CSU – RAMS

Procedures for Common User Customizations of the Model

This document contains a list of instructions and examples of how to customize this model. This includes the addition of new scalar and scratch array variables and how to implement these for sequential and parallel-processing simulations. It also discusses how to add new subroutines, initialization of new variables, and adding source terms to atmospheric variables.

Updated by:

Stephen Saleeby
Department of Atmospheric Science
Colorado State University

Last updated: 22 Jan, 2016

Common Customizations for RAMS v6.0

It is often necessary for a user to modify one or more sections of the atmospheric model code to customize it for a particular application. While the namelist variables described in the Model Input Namelist Parameters document control the vast majority of options available in the model, it is simply not practical to place all foreseeable options under their control. Therefore, we will describe here some of the most commonly encountered reasons for requiring code modifications, and present the most straightforward means of accomplishing each.

This document assumes not only that the user understands FORTRAN programming, but also various more recent concepts. In particular, the Fortran 90 user-defined datatypes must be understood. If you are not very familiar with the usage of the datatypes, please consult a Fortran 90 programming manual first.

The following discussion does not address the concept of “thread-safe” programming and shared-memory parallelism. RAMS does not use this yet, but we are moving in this direction. So if you intend to be developing code for the future, you want to consider these issues.

1) Adding Array Space to the Model

RAMS is mostly programmed in standard Fortran 90. If you wanted to, you could bypass all the existing memory structures and simply add new arrays with standard declaration and dimension statements. However, it is usually extremely desirable to take advantage of the existing structure to have the new arrays be included in file I/O and/or the parallel processing.

In some applications, a user needs to dimension a new array in the atmospheric model for various reasons. These reasons may include:

· adding one or more new prognostic fields
· adding one or more new diagnostic fields for output

V6.0 is significantly easier to program than previous versions that used the “A” array to configure memory. However, a completely different method of memory configuration is used which may involve learning new concepts. We will first look at one of the existing memory modules as an example.

1) src/memory/mem_turb.f90: An Example
All significant model memory in RAMS is dynamically-allocated. The primary 3D and 2D variables are allocated in the *mem_** modules in the *src/memory* subdirectory. These modules will be called by *subroutine rams_mem_alloc* in *src/memory/alloc.f90* to allocate the main memory.

Let's look at the main components of one of the memory modules as an example. Using *mem_turb.f90* as the example, we see the various parts of the module:

1. Declaration statements – the primary declaration is that of the *turb_g* user-defined datatype (which can be loosely viewed similar to a C structure). Once all the allocations are performed, there will be, for example, the TKE variable that can be accessed by *turb_g(ngrid)%tkep(k,i,j)*. *tkep* is declared as a pointer, which is a member of the *turb_g* datatype.

2. *alloc_turb* – routine which actually allocates the memory. Note some options may be checked for conditional allocation.

3. *nullify_turb* – routine which makes sure the datatype member pointers are disassociated with any memory.

4. *dealloc_turb* – routine which deallocates the pointer memory. Note that these routines have been removed since they are not necessary during runtime. These could be easily created as being similar to *nullify_turb* but with use of the deallocate statement rather than nullify.

5. *filltab_turb* – for the arrays that are allocated, enter the variable into various variable tables. This is accomplished by passing a character string, delineated by colons (:), which contains a number of parameters. As you can see, this is where the VTABLE information was put. We determined that if you could figure out VTABLE file, you could easily make the same modifications in code!

a. **Name** – name assigned to the variable for the analysis files
b. **Dimensionality** – an integer code defining the subscript type
    i. 2 – 2 dimensions (nxp,nyp)
    ii. 3 – 3 atmospheric dimensions (nzp,nxp,nyp)
    iii. 4 – 4 soil dimensions (nxp,nyp,nzg,npatch)
    iv. 5 – 4 snow dimensions (nxp,nyp,nzs,npatch)
    v. 6 – 3 LEAF dimensions (nxp,nyp,npatch)
c. Variable tables
    i. *anal* – write variable to analysis files
    ii. *mpti* – parallel table – send variable from master to node during initialization
    iii. *mpt3* – parallel table – send variable from node to master at file write times
    iv. *mpt1* – parallel table – exchange variable boundaries among subdomains during the timestep

To make the array space that is configured in a module available to a Fortran subroutine, use the Fortran use statement directly after the subroutine statement. So for example:

```
subroutine mysub(n1,n2,n3)
use mem_turb
```

Note the *use* statement specifies the *module* name, not the file name (as in an *include* statement). This is rather similar, in practice, to declaring a variable in a *common* block and using an *include* statement. However, the details are substantially different.

It is left as an exercise to the reader to compare a few of the mem_*.f90 modules with *mem_turb.f90* to recognize the similarity of structure.

2) I want to add…

…a local scratch array for computations within the routine.

If the new array space is only required within one subroutine, it is simplest to add the appropriate declaration statement to that subroutine. Note that all routines use *implicit none*, so make sure to use the appropriate *integer, real*, etc. declaration. It is your choice whether to use static, dynamically-allocated, or automatic arrays.

If you need several scratch variables and/or you don't want to be bothered making sure you allocate the array big enough, you can use the global scratch arrays. These arrays are listed in mem_scratch.f90. You can either *use* the module:

```
use mem_scratch
```

then reference the arrays such as:

```
scratch%vt3da(k,i,j) = scratch%vt3db(k,i,j)
```

or you can pass them through the call statement (usually more efficient on current compilers):

```
call mysub2(nzp,nxp,nyp,scratch%vt3da)

subroutine mysub2 (n1,n2,n3,scr_arr_a)
implicit none
integer :: n1,n2,n3
real, dimension(n1,n2,n3) :: scr_arr_a
```

…a scratch array to pass computations among routines.

In this situation, you probably want to use the global scratch arrays. However, you must be careful that no other routines use the scratch array between the time you fill it and the time you want to use the results.

…an array to store values for writing to output files.

Sometimes you may want to save values that the model computes in order to write them to the output analysis files. In this case, it is easiest to add the variable to the

existing memory modules. Therefore, you do not need deal with the actual output yourself; the model will take care of it. Let's say you wanted to add the turbulent vertical temperature flux to the analysis files. Let's again use the example of *mem_turb.f90*. Here are the steps for editing the module:

1) Determine the dimensionality of the variable. Add the pointer name to the appropriate list of 2-D or 3-D variables. Let us assume we will add a 3-D variable which we will name *tflux*. The declaration statement will then be modified to:

```
! Variables to be dimensioned by (nzp,nxp,nyp)
real, pointer, dimension(:,:,:) :: tkep,epsp,hkm,vkm,vkh,cdrag,tflux
```

2) Add the allocation to the *alloc_turb* routine:

```
subroutine alloc_turb (turb,n1,n2,n3,ng)
…
allocate (turb%tflux(n1,n2,n3))
```

Include the proper conditional statements if the allocation is dependent on namelist options.

3) Add the deallocate and nullify calls to *nullify_turb* and *dealloc_turb* routines. Note the *associated* intrinsic function will check to see if the array exists before performing the functions. As previously mentioned, the deallocate routines have been removed since they are not necessary during runtime. These could be easily created as being similar to *nullify_turb* but with use of the deallocate statement rather than nullify.

4) Add a *tflux* call to *filltab_turb*:

Notes:
· The *turbm* datatype is used for storing the time-averaged fields. Even if you have no plans on using them, include it in the call (memory is not actually allocated if you aren't using them).
· Add the *anal* table option if you want this field output to analysis files.
· You need to determine which parallel table(s) to include. In this example, most likely only *mpt3* is needed, unless you do want this variable to be read on a history restart. Then *mpti* is required also.

All that is left, then, is to add the new array to the appropriate routines to actually fill in the values.

```
subroutine filltab_turb (turb,turbm,imean,n1,n2,n3,ng)
…
    if (associated(turb%tflux)) &
      call vtables2 (turb%tflux(1,1,1),turbm%tflux(1,1,1),ng, npts, imean, &
            'TFLUX :3:hist:anal:mpti:mpt3:mpt1')
```

There would be REVU modifications if you wanted to access the variable…

…a new prognostic variable.
In v6.0, it is easy to simply add the variable to an existing module, or if it is a new parameterization, to add a new module.

The modifications to add a new prognostic variable start out exactly the same as above. Some additional comments:

1) The RAMS naming conventions have a “P” as the last character of prognostic variable name. This “P” is replaced by a “T” for the tendency (see below).

2) If the variable needs to be communicated across parallel subdomain boundaries, the mpt1 table is needed to be specified. If you are not concerned with efficiency and you are not sure, you can specify this table anyway.

3) Edit *mem_tend.f90*. The declaration statements designate the names of the time tendency arrays (rate of change of the variable for a timestep). As in the comment for step 1, add a tendency array for your variable, and consider the naming convention.

4) In *subroutine alloc_tend*, add the allocate statement for your tendency array.

5) Add the appropriate statements to *nullify_tend* and *delloc_tend*.

6) If the variable is to be considered a “***scalar***” variable, in *filltab_tend*, we must add a call to *vtables_scalar* to:

a. Associate the tendency array with the prognostic variable array

b. Give the scalar an internal name. This does not need to be the same as the name for the analysis files, but it can be.

What is meant by a “***scalar***” variable? If you specify a prognostic variable to be a ***scalar***, it will be defined at the thermodynamic point in the stagger and it will be automatically advected, diffused (using the eddy viscosity for heat), participate in the two-way nesting algorithms, and updated in time with other scalars in the timestep cycle. Unless of course, you can add code to change this automatic behavior where you do not want it to occur (example: TKE is a scalar, but a different eddy viscosity coefficient is used.)

If you do not want the variable to be a scalar, it does not need to be included in *filltab_tend*.

That is all that needs to be done for the memory configuration. There still, of course, are things to consider. You may want to define an initial field for your variable. We will discuss that below. And you may want to add some source/sink terms to the model code. A simple example of specifying a tendency will be mentioned below also.

Developing an entirely new memory module which includes prognostic variables is just as easy. One of the existing modules can be used as a template. The only additional step is that a section in *src/memory/alloc.f90* needs to be added. Again, an existing section there can be used as a template.

3. Customization of Initial Atmospheric Prognostic Fields

It is often desired to initialize certain atmospheric fields, such as potential temperature and moisture mixing ratio, with specific horizontally inhomogeneous perturbations. For example, a high resolution simulation of a thunderstorm may be initialized with a local region of enhanced temperature and moisture to trigger the onset of convection at a specific location.

Following is a simple example to modify or specify initial fields for RAMS. The following template subroutine, called subroutine bubble, is provided in the file *ruser.f90* in order to customize one or more prognostic fields in the model.

The first step is to determine where to call the perturbation routine. The usual place to call the routine is during the initialization procedure, before you need to worry about parallelization. This way, we have the full arrays available for all grids.

Note that the included subroutine bubble was intended to add a warm, moist perturbation to initiate convection. However, any type of values can be set for any field in a similar routine.

Take a look at *src/init/rdint.f90*. Find the following section (we will not specify line numbers, since they go out of date easily!):

```
! Initialize past time level velocity and perturbation
!Exner function on all grids.

do ifm=1,ngrids

   call newgrid(ifm)
   call fldinit(1)
   call negadj1(nzp,nxp,nyp)
   call thermo(nzp,nxp,nyp,1,nxp,1,nyp,'THRM_ONLY')

   If (level == 3) then
     call initqin(nzp,nxp,nyp,micro_g(ifm)%q2 (1,1,1),micro_g(ifm)%q6 (1,1,1) &
       ,micro_g(ifm)%q7 (1,1,1),basic_g(ifm)%pi0 (1,1,1),basic_g(ifm)%pp (1,1,1) &
       ,basic_g(ifm)%theta (1,1,1),basic_g(ifm)%dn0 (1,1,1) )
   endif

enddo
```

This section performs various initialization functions and is executed for all RUNTYPE = ’INITIAL’ types. It is also the last grid loop in this section.

So here is one good place to add a call to bubble.

```
! Initialize past time level velocity and perturbation
!Exner function on all grids.

do ifm=1,ngrids
   call newgrid(ifm)
   call fldinit(1)
   call negadj1(nzp,nxp,nyp)
   call thermo(nzp,nxp,nyp,1,nxp,1,nyp,'THRM_ONLY')
   call bubble (n1,n2,n3,basic_g(ifm)%thp(1,1,1), basic_g(ifm)%rtp(1,1,1))
….
```

```
subroutine bubble(m1,m2,m3,thp,rtp)
implicit none
integer :: m1,m2,m3,i,j,k
real, dimension(m1,m2,m3) :: thp,rtp

do j = 1,1
  do i = 17,26
    do k = 2,7
      thp(k,i,j) = thp(k,i,j) + 5.
      rtp(k,i,j) = rtp(k,i,j) * 1.2
    enddo
  enddo
enddo
return
end
```

In this example, two 3-D arrays, potential temperature (thp) and total water mixing ratio (rtp) are passed to the subroutine, as are the dimensions (m1,m2,m3) of the arrays. They illustrate an example where **thp** (theta il) is increased by 5 K and **rtp** (total water mixing ratio) is increased by 20% in the lowest 6 model levels (k=2,7) and over 10 different locations in the x-direction (i=17,26).

Note that because of where we placed the call statement, this routine will be called for all grids. If you only wanted to give the perturbation to grid 2, you can place the call outside of the grid loop, then specify the grid index and explicit grid point dimensions:

```
! Initialize past time level velocity and perturbation
!Exner function on all grids.
do ifm=1,ngrids
   call newgrid(ifm)
   call fldinit(1)
   call negadj1(nzp,nxp,nyp)
   call thermo(nzp,nxp,nyp,1,nxp,1,nyp,'THRM_ONLY')
….
enddo

call bubble (nnzp(2),nnxp(2),nnyp(2) &
     ,basic_g(2)%thp(1,1,1), basic_g(2)%rtp(1,1,1))
```

Or, you could leave the call in the original location and pass in the grid number, then check it in the bubble routine:

```
! Initialize past time level velocity and perturbation Exner function on all grids.
do ifm=1,ngrids
  call newgrid(ifm)
  call fldinit(1)
  call negadj1(nzp,nxp,nyp)
  call thermo(nzp,nxp,nyp,1,nxp,1,nyp,'THRM_ONLY')

  call bubble (n1,n2,n3, ,ifm,basic_g(ifm)%thp(1,1,1), basic_g(ifm)%rtp(1,1,1))
….
```

```
subroutine bubble(m1,m2,m3,ng,thp,rtp)
implicit none
integer :: m1,m2,m3,ng,i,j,k
real, dimension(m1,m2,m3) :: thp,rtp

if (ng == 2) then

 do j = 1,1
  do i = 17,26
   do k = 2,7
     thp(k,i,j) = thp(k,i,j) + 5.
     rtp(k,i,j) = rtp(k,i,j) * 1.2
   enddo
  enddo
 enddo

endif

return
end
```

And please note: you can change the name of the routine. It does NOT have to be called *bubble*!

4 Adding a New Source Term to Model Prognostic Fields

Another need which sometimes arises in the use of the atmospheric model is to add an artificial forcing term to one or more prognostic fields. For example, as an alternative to placing a warm, moist perturbation in the initial fields to trigger convection as described in the previous section, a heat and moisture *source* may instead be imposed in a region of the domain. This source adds the heat and moisture gradually over a period of time. The following template subroutine, called *subroutine force*, is a simple example which customizes a prognostic field in the model.

```
subroutine force(n1,n2,n3,tht)
implicit none
integer :: n1,n2,n3
real :: tht(n1,n2,n3)
integer :: i,j,k

do j = 10,12
  do i = 17,26
    do k = 2,7
      tht(k,i,j) = tht(k,i,j) + 0.001
    enddo
  enddo
enddo

return
end
```

In this example, the 3-D array **tht**, which is the potential temperature tendency in [K/s], is incremented by a value of 0.001 K/s.

A routine such as *force* should be called from subroutine TIMESTEP (*rtimh.f90*) by adding a call such as the following after the call to TEND0 (which will zero out all tendency arrays), but before the call to PREDTR:

```
if(time < 1800.) call force(nzp,nxp,nyp,tend%tht(1))
```

In this example, the artificial forcing term is applied to **tht** only during the first 1800 seconds of the simulation. As you should have seen in *mem_tend.f90*, the tendency arrays are one dimensional, since they are used for multiple grids.

A BIG caveat: the preceding code is only correct for a run on a single processor. So it might be appropriate to include here a few words about distributed memory parallelism.

When you run in parallel, RAMS, as most codes of this kind, uses domain decomposition to achieve parallelism. This means that portions of each grid are executed on different processors. Each processor is only given a portion of the total domain. When subroutine timestep is executed, there are numerous copies of the routine that are running, each with its own domain portion. Therefore, when calling a routine such as force, you need to make sure that the perturbation is applied in the proper location relative to the whole domain.

When code is executed with a “grids” loop, which includes a *call newgrid* (*subroutine timestep* qualifies), there are several variables set up for you to help locate this subdomain:

* *mxp, myp, mzp – mxp* and *myp* are the total number of horizontal grid points of the subdomain on the current processor. *mzp* is the same as *nnzp(ngrid)*. For all 3-D, 2-D, soil, snow, etc. variables, these are the array bounds that are actually allocated on a processor.

* *ia, iz, ja, jz* – the bounds of the subdomain relative to *mxp* and *myp* which this processor is responsible for updating the variables. A do loop then, such as:

```
do j = ja,jz
  do i = ia,iz
.
..
  enddo
enddo
```

will loop over all the columns which this processor will update.

* *i0, j0* – the "offsets" for this processor. For example, when you add the *i* value from the previous loop to *i0*, you will get the *i* grid point value relative to the **whole** grid, not just the subdomain portion.

* *ibcon* – not used here, but is a 4-byte flag designating whether the subdomain boundaries are full domain boundaries.

So let's rewrite subroutine force taking into account the parallelism. We will apply the perturbation over the same points relative to the entire grid.

```
if(time < 1800.) call force(mzp,mxp,myp,ia,iz,ja,jz,i0,j0,tend%tht(1))
```

The routine will be called from the same location.

```
subroutine force(m1,m2,m3,ia,iz,ja,jz,i0,j0,tht)

implicit none

integer :: m1,m2,m3,ia,iz,ja,jz,i0,j0
real :: tht(m1,m2,m3)
integer :: i,j,k

do j = ja,jz
  do i = ia,iz
    do k = 2,7

       if (i+i0 >= 17 .and. i+i0 <= 26  .and.  j+j0 >= 10  .and.  j+j0 <=12) &
            tht(k,i,j) = tht(k,i,j) + 0.001

    enddo
  enddo
enddo

return
end
```

There are obviously more efficient ways to write this, but it provides an example.

5 Adding a variable to the RAMSIN namelist in the distributed memory version of RAMS (_DM)

Step 1: Place your new namelist variable in the appropriate group section of the RAMSIN namelist. These group sections are MODEL_GRIDS, MODEL_FILE_INFO, MODEL_OPTIONS, MODEL_SOUND, ISAN_CONTROL, ISAN_ISENTROPIC. Decide if is grid dependent or not. Separate all values with commas and end the input values with a comma (ie. ICLOUD = 5,). If the variable is grid dependent this would include multiple values separated by commas. The values are in order of the grids.

Note: the following steps are for inclusion of namelist variables that are not ISAN_CONTROL or ISAN_ISENTROPIC namelist variables. If you need to include an ISAN variable, follow similar procedures as below but in the appropriate sections of io/rname.f90, mpi/mpass_init.f90, isan/isan_coms.f90, and isan/isan_name.f90. I would suggest doing a search on one of the ISAN variables as an example. Then follow that example and a similar approach to the instructions below.

Step 2: Declare your variable in one of the memory modules. For example, ICLOUD is placed in micphys.f90 in the module “micphys”.

Step 3: Within file mpi/mpass_init.f90 subroutine “broadcast_config” you will enter the new variable within a “par_put” AND a “par_get” statement. These will either be integers or floats, so use “par_put_int” and “par_put_float” appropriately. If the variable is a single integer value, an input sample statement might look like
“CALL par_put_int (ICLOUD,1)”.
If it is multi-valued float variable it might look like
“CALL par_put_float (AERO_MEDRAD,9)”.
The customization depends upon the dimensions of the variable. Lastly, you need to increment “nwords” at the top of the routine by the number of variables being input. If this is a single dimension variable, just increment the main number by 1. If it is multidimensional then increment the appropriate number (ie. maxgrd, nzpmax, nzgmax, etc.) in the multipliers of “nwords”.

Step 4: In io/rname.f90 you need to enter the variable for outputting to standard output that is printed when the model is first executed. Your variable should go into an appropriate DATA statement (either GRIDS, START, INDAT, or SOUND), and increment the associated counter variable (either nvgrid, nvstrt, nvindat, or nvsound). Next, scroll down and enter your variable in a statement similar to:
“IF(VR.EQ.'ICLOUD') CALL varseti (VR,ICLOUD,NV,1,II,0,5)” or
“IF(VR.EQ.'GNU') CALL varsetf (VR,GNU(NV),NV,8,FF,0.,20.)”.
Note the difference here for integers vs. floats. Also note the difference between single value variable “ICLOUD” vs. multi valued variable “GNU”. The final 2 numbers in the calls to varset are the upper and lower numerical bounds. Finally scroll down in the same file and enter a print statement in the appropriate section.