I/O API: New Stuff in Versions 3.x

Table of Contents

I/O API: New Stuff in Versions 3.x

Contents / Agenda

On Github: I/O API-3.2

Version 3.2 of the I/O API library code is available here for download in *gzip*ped-*tar* source code form from the CMAS Center web-site, or is available on GitHub from ***https://github.com/cjcoats/ioapi-3.2***. To install a distribution-copy using *git* directly from GitHub, go to the directory under which do the installation, and then do the command

```
git clone https://github.com/cjcoats/ioapi-3.2
```

Sub-directories and files are:

- ***ioapi***: library source directory
- ***M3Tools***: tools-program source directory
- ***LICENSE***: GPL-2 text
- ***Makefile***: Top-level *make* file
- ***Makefile.template***: master copy of top-level *make* file
- ***README.txt***: ASCII "readme"
- ***README.md***: markup-language "readme"
- ***VERSION.txt***: tarball-release date file
- ***exclude***: file used in *make gtar* tarball-creation

The full set of documentation is also git-configuration-managed as HTML pages, in a manner consistent with the release-tarballs and the CMAS Center I/O API web-pages. It can be found

at ***https://cjcoats.github.io/ioapi/AA.html***

Back to ***Contents***

MS-Windows support

I/O API-3.2:

Support for 32-bit and 64-bit windows builds of the I/O API under *CygWin*, using *gfortran* and *gcc*. Note that *Makefile*s must be changed slightly by adding an extra `.dll` suffixes to the library-directives for the netCDF libraries:

```
... -lnetcdff.dll -lnetcdf.dll ...
```

See ***https://cjcoats.github.io/ioapi/AVAIL.html#win64***

Back to ***Contents***

CF-compliant geospatial metadata option

CF (Climate and Forecast) netCDF metadata conventions are used by a number of global climate and meteorology models. This allows the direct import of model data directly into a number of GIS, analysis, and graphics packages, including GRASS and ARC-INFO. These conventions define metadata that provide a definitive description of what the data in each variable represents, and the spatial and temporal properties of the data. This enables users of data from different sources to decide which quantities are comparable, and facilitates building applications with powerful extraction, regridding, and display capabilities.
See ***http://cfconventions.org/*** and
https://en.wikipedia.org/wiki/Climate_and_Forecast_Metadata_Conventions

Automatic creation of the relevant CF-compliant geospatial metadata during file-creation can be turned on by environment variable `IOAPI_CFMETA`:

```
setenv IOAPI_CFMETA  Y
```

or by calling subroutine `INITCF()` in `MODULE MODATTS3`: see below.

(There are portions of the CF standard that are not reasonable in this context, particularly as regards standardization of variable names and units in forms that are not well-suited for atmospheric chemistry modeling; these are not supported, and in fact production of them cannot be automated...)

Back to ***Contents***

PnetCDF/MPI distributed I/O option

The I/O API can now be configured to use PnetCDF to provide distributed I/O on a file-by-file basis for CMAQ (turning this on is a build-time option for the I/O API). Distributed I/O is only supported for gridded files on the cross-point grid; to turn it on for a

particular file in a CMAQ run, prefix the path-name by a *MPI:* prefix, as in the example below:

```
setenv  CHEMCONC3D  MPI:/mydir/cmaq.conc.US36_CRO.2015233.ncf
```

See this description.

I/O API libraries and builds using PnetCDF are not link compatible with ordinary builds, and should be kept carefully separate from them. You can build the I/O API to use PnetCDF/MPI distributed I/O using *make -f Makefile.pncf* and the following binary types and `Makeinclude` files (or you can use them as templates to build your own custom binary type):

```
Makeinclude.Linux2_x86_64gfortmpi
Makeinclude.Linux2_x86_64ifortmpi
Makeinclude.Linux2_x86_64pgmpi
Makeinclude.Linux2_x86_64sunmpi
```

When performing the link-step to create model-executables, you will need to put the PnetCDF libraries in the library-build directory, and add the PnetCDF libraries to the link-step command line:

```
... -lpnetcdf -lnetcdff -lnetcdf ...
```

See also `MODULE MODNCFIO` below.

Back to ***Contents***

I/O API Versions 3.2 and 3.2-large

I/O API Version 3.2 as of Aug. 1, 2019, has been enhanced to support up to 256 files (each with up to 2048 variables. There should be no compatibility issues with this change, since the maximum-file parameter `MXFILE3` is only used internally by the I/O API, and should not be needed by any external program.

For use in CMAQ's DDM and ISAM versions, there is an additional version, **I/O API-3.2-large** that supports up to 512 files, each with up to 16384 variables; it may be downloaded from the CMAS web-site as ***ioapi-3.2-large.tar.gz***. There is an accompanying set of *M3Tools* programs.

Note that executable programs built using 3.2-large will use substantially more memory than programs built with normal 3.2, and may encounter performance degradation due to the extra strain that large-mode places on your computer's memory system.
Version 3.2-large should be kept carefully separate from normal 3.2: object-files and libraries built with the one are *not* compatible with object-files or libraries built with the other. Mixing the two versions can lead to hard-to-diagnose errors.

Back to ***Contents***

Snoop-Mode option for read-operations

The I/O API can now be configured (e.g., for forecast-modeling-system operations) to enable "model-pipelining" by responding to end-of-file conditions with multiple re-tries, controlled by environment variables `SNOOPTRY3, SNOOPSECS3`:

> If `SNOOPTRY3, SNOOPSECS3 > 0`: when read-operations `READ3(), XTRACT3(), INTERP3(), CHECK3(), or DDTVAR3()` encounter end-of-file, they will re-try for up to `SNOOPTRY3` attempts, with delay `SNOOPSECS3` seconds in between attempts.
>
> If `SNOOPTRY3 = 0, SNOOPSECS3 > 0` then the number of re-tries as above is (almost) unlimited.
>
> If `SNOOPTRY3 < 0` or `SNOOPSECS3 ≤ 0`, then *Snoop Mode* is turned off.

This not only allows the generation of early-hour forecast products well before the entire forecast is complete, it also enables the operating system to make better use of its internal I/O-buffers, further increasing modeling system efficiency.

Unfortunately, both CMAQ and SMOKE do not allow easy use of this capability because their program-codes insist (inappropriately!) that all time steps of all input data be available before model-start.

Back to ***Contents***

New Module `MODATTS3`

New `MODULE MODATTS3` replaces `MODULE MATXATTS` (so that all `USE MATXATTS` statements need to become `USE MODATTS3` in your codes):

- Matrix-attribute routines (add extra *netCDF* file attributes to describe input and output grids for matrix-files):
  - `INITMTXATT()`
  - `GETMTXATT()`
  - `SETMTXATT()`
  - `CHKMTXATT()`
  - `ENDMTXATT()`
- CF-convention geospatial-metadata routines, turned on by environment variable `IOAPI_CFMETA`, make newly-created files CF compliant:

  setenv IOAPI_CF Y

  - `INITCF()`
  - `SETCF()`
  - `ENDCF()`
- "Standard" CMAQ data structures and routines for CMAQ metadata, turned on by environment variable `IOAPI_CMAQMETA`:

setenv IOAPI_CMAQMETA Y

- ◊ INITCMAQ()
- ◊ ISCMAQ()
- ◊ GETCMAQ()
- ◊ LOGCMAQ()
- ◊ SETCMAQ()
- ◊ ENDCMAQ()

♦ Place-holder for "standard" SMOKE metadata data structures and routines (what this metadata contains needs to be determined...), turned on by environment variable IOAPI_SMOKEMETA:

setenv IOAPI_SMOKEMETA Y

- ◊ INITSMOKE()
- ◊ ISSMOKE()
- ◊ GETSMOKE()
- ◊ LOGSMOKE()
- ◊ SETSMOKE()
- ◊ ENDSMOKE()

Developing the data dictionary for standard SMOKE metadata is an open issue below, that is needed before the "guts" of these routines can be filled in.

Back to ***Contents***

New Module MODGCTP

New MODULE MODGCTP with all the coordinate-transform related routines and INTERFACEs:
Old but modified: (moved from MODULE M3UTILIO)

- ♦ INTERFACE for GTPZ0()
- ♦ INTERFACEs and new implementations for GTPZ0()-spheroid initialization procedures SETSPHERE(), INITSPHERES(), SPHEREDAT()
- ♦ INTERFACEs and new implementations for single-precision/single-point/specific-projection coordinate-transform functions INITPROJ(), ..., LAMBERT(), ..., ALB2EQM()

New:

- ♦ Generic (single-point/scattered-point-array) coordinate-transform procedure XY2XY()
- ♦ Grid-node-array generic coordinate-transform procedure GRID2XY()
- ♦ High-performance OpenMP-Parallel generic bilinear interpolation package GRID2INDX(), PNTS2INDX(), INDXMULT()
- ♦ PARAMETERs STDSPHERES(0:21) and SPHERENAMES(0:21) for GCTP spheroid names and indices;
- ♦ GTPZ0()-argument initialization procedure M3TOGTPZ()

Back to ***Contents***

New Module MODNCFIO

This module exists for two reasons: to reconcile problems between netCDF and pnetCDF for use in distributed I/O (above), and to provide high level I/O facilities for "Raw netCDF".

Due to inconsistencies between them, it was difficult to use the vendor supplied INCLUDE files *netcdf.inc* (or *NETCDF.EXT*) and *pnetcdf.inc* for the I/O API Distributed-I/O Mode: the latter defined some but not all of the netCDF definitions that were needed, so it was not possible to use #ifdefs to select between them. New MODULE MODNCFIO puts all the definitions in a consistent form in one place (with conditional compilation, to determine whether or not PnetCDF is active).

This module also supplies new routines for use with a specified "raw netCDF" file:

- CREATENC() creates a new "raw netCDF" file, according to the supplied file definition.
- DESCNCVAR() returns the list of variables and their units, types, and dimensions,
- READNCVAR() is a generic routine that reads a specified 1-D, 2-D, 3-D, or 4-D variable or time step of a variable of type INTEGER, REAL, REAL*8, INTEGER*1, or INTEGER*2, with both fully-dimensioned and single-indexed forms for the output-buffers from a "raw netCDF" file
- WRITENCVAR() is a generic routine that writes a specified variable or time step of a variable to a specified "raw netCDF" file

Note that directly manipulating netCDF files using the low level access routines provided by netCDF, while doing proper error checking and logging is quite tedious (which is reflected in the fact that MODULE MODNCFIO requires about 20,000 lines of code to do these tasks (You *don't* want to have to write all that code for yourself from scratch!>

The latest version of SMOKE uses READNCVAR() to read global gridded emissions data. Many other uses (e.g., to read INTEGER*2 gridded NLDAS land-cover for CONUS) can easily be imagined.

Back to ***Contents***

New Module MODMPASFIO and related *M3Tools* Programs

MPAS is a potentially-global unstructured-grid model for meteorology, land surface, and atmospheric chemistry modeling that has been proposed for the next generation of EPA air quality modeling (among other things). See https://mpas-dev.github.io/files/documents/MPAS-MeshSpec.pdf for the MPAS grid and netCDF-file specifications. Note that MPAS grid definitions and MPAS-netCDF-format file specifications are very complex.

New MODULE MODMPASFIO provides a copy of the MPAS required grid related and indexing variables, together with routines to initialize them, perform various grid related tasks, and to read and writhe time independent and time stepped MAPS-format-netCDF files. This is a really massive chunk of code, managing the hugely-complex detail that is required for MPAS related programming.

There are also four related ***M3Tools*** programs:

- ***mpasdiff***: compute per-variable and per-layer-range-of-variable statistical comparisons between MPAS-format-netCDF files
- ***mpasstat***: compute statistics for variables and layer-ranges-of-variable in a MPAS-format-netCDF file
- ***mpastom3***: interpolate variables from MPAS-format-netCDF files to GRIDDED I/O API files; and
- ***mpaswtest***: sample program for MPAS grid manipulation, that does MPAS-grid allocation of emissions line sources.

Back to ***Contents***

New Module `MODWRFIO` and related *M3Tools* Programs

This module contains routines and INTERFACEs for reading and writing WRF format netCDF files:

- **`OPENWRF()`**: opens the indicated WRF-netCDF format file for read or read/write.
- **`CRTWRF()`**: Opens/creates a new WRF-netCDF format file for read/write.
- **`READWRF()`**: Reads the indicated time step for the indicated variable from the current WRF-netCDF format file. Generic, for 1-D, 2-D, and 3-D variables of types INTEGER, REAL, or DOUBLE
- **`WRITEWRF()`**: Writes the indicated time step for the indicated variable to the current WRF-netCDF format file (also generic).
- **`CLOSEWRF()`**: Close/flush the indicated WRF-netCDF format file

There are also two related ***M3Tools*** programs; these are much more generic (not tied to specific WRF versions nor CMAQ-variable output) than MCIP:

- ***wrftom3***: Convert/extract/window variables from WRF-format netCDF files to GRIDDED I/O API files
- ***wrfgriddesc***: Create a GRIDDESC file from the header-data in a WRF-format netCDF file.

Back to ***Contents***

Input File Lists: Easy Processing for Sequences of Files

`READ3()` and `INTERP3()` can handle lists of consecutive files, to cover multi-run/multi-file studies. For example if you have a month-long set of 31 single-day files, here is how an I/O API based program can process all of them as though they were the single file `FOO`

```
setenv  FOO      "LIST:NAME_1,...,NAME_31"
setenv  NAME_1   <path>
...
setenv  NAME_31  <path>
```

provided that files NAME_1 through NAME_31 have the same grid, variable-sets, and time-steps.

READ3() and INTERP3() will find the first file in the list containing the indicated date&time, and read the data from it. Note that this is automatically part of the I/O API, so that, for example, ***M3Tools*** program ***m3stat*** can be used to process an entire month's single-day model data without having to post-process it into a single file. Or the CMAQ CCTM could do an entire month-long run using a sequence of single-day emissions and meteorology input files.

Back to ***Contents***

New Fortran-90 Generic Routines

Many of these are what's needed to provide explicit INTERFACEs where arguments may have been previously single-indexed or not, or to provide for both REAL and REAL8 arguments, etc. Transform routines in MODULE MODGCTP exist in both same-sphere and sphere-to-sphere forms. In addition to the many generic routines now found in MODULE M3ATTS and MODULE MODGCTP, the following are now defined in MODULE M3UTILIO

Note that these generic INTERFACEs require USE M3UTILIO.

```
SUBROUTINE BILIN():
      BILIN11L(), BILIN12L(), BILIN21L(), BILIN22L(),
      BILIN11(), BILIN12(), BILIN2L(), BILIN22()
SUBROUTINE BMATVEC():
      BMATVEC11(), BMATVEC12(), BMATVEC21(), BMATVEC22(),
      BMATVEC01(), BMATVEC02(), BMATVEC021(), BMATVEC022()
LOGICAL FUNCTION ENVLIST():
      INTLIST(), REALIST(), DBLLIST(), STRLIST()
<type> FUNCTION ENVGET():
      BENVINT(), BENVDBLE(), BENVREAL(), ENVINT(), ENVDBLE(),
      ENVREAL(), ENVSTR(), ENVYN()
INTEGER FUNCTION FINDKEY():
      FINDC(), FIND1(), FIND2(), FIND3(), FIND4(), FINDL1(),
      FINDL2(), FINDL3(), FINDL4(), FINDR1(), FINDR2(),
      FINDR3(), FINDR4()
<type> FUNCTION GETVAL():
      GETDBLE(), GETDBLE1(), GETMENU(), GETNUM(), GETNUM1(),
      GETREAL(), GETREAL1(), GETYN()
INTEGER FUNCTION LOCATE():
      LOCAT1(), LOCAT2(), LOCAT3(), LOCAT4(), LOCATC(),
      LOCATL1(), LOCATL2(), LOCATL3(), LOCATL4() LOCATR1(),
      LOCATR2(), LOCATR3(), LOCATR4()
SUBROUTINE PMATVEC():
      PMATVEC11(), PMATVEC12(), PMATVEC21(), PMATVEC22()
SUBROUTINE SORTI():
      SORTIC4(), SORTIC8(), SORTINC4(), SORTINC8(), SORTI1(),
      SORTI2(), SORTI3(), SORTI4(), SORTL1(), SORTL2(),
```

```
      SORTL3(), SORTL4(), SORTR1(),SORTR2(), SORTR3(), SORTR4()
SUBROUTINE UNGRIDB():
      UNGRIDBS1(), UNGRIDBS2(), UNGRIDBD1(), UNGRIDBD2()
SUBROUTINE UNGRIDI():
      UNGRIDIS1(), UNGRIDIS2(), UNGRIDID1(), UNGRIDID2()
```

Back to ***Contents***

M3Tools Programs

Date-and-Time Manipulation for Scripting: *datshift, greg2jul, jul2greg, juldiff, julshift, timeshift*
These echo the program's result (without any extraneous stuff). Here are some sample uses (where note that "enclose in back-quotes" means to take the result of the program and use it, e.g., as the right-hand side of *set ... =* , and that the *:r* "root" and *:e* "extension" operators take the left and right hand sides of the period in shell-variable values):

```
set jdate = `greg2jul today`
set kdate = `greg2jul 20150103`             # converts Jan. 3, 2015 to YYYYDDD format
set idate = `julshift ${jdate} 7`           # Julian date for this day next week
set days  = `juldiff ${kdate} ${jdate}`     # is the number of days from 2015003 to to
timeshift 2014029.120000 183000             # echoes 2014030.063000
set foo = `timeshift 2014029.120000 183000`
echo $foo:r                                 # echoes 2014030
echo $foo:e                                 # echoes 063000
echo `jul2greg 2015133`                     # echoes 20150513 (for May 13, 2015)
```

New *M3Tools* Programs:
bcwndw, camxtom3, dayagg,
vertimeproc, vertintegral, findwndw, gridprobe, insertgrid, m3mask, m3probe, m3totxt, wrfgriddesc, wrftom3

OpenMP-Parallel *M3Tools* Programs:
bcwndw, dayagg, m3agmask, m3agmax, m3combo, m3cple, m3interp, m3mask, m3tproc, mtxcalc mtxcple, presz, vertimeproc, vertintegral, vertot

Fortran-90 "Free" (*.f90*) source format:
bcwndw.f90, camxtom3.f90, datshift.f90, dayagg.f90, factor.f90, fakestep.f90, fills.f90, greg2jul.f90, gregdate.f90, jul2greg.f90, juldate.f90, juldiff.f90, julshift.f90, latlon.f90, m3combo.f90, m3cple.f90, m3fake.f90, m3mask.f90, m3pair.f90, m3probe.f90, m3totxt.f90, m3tproc.f90, m3tshift.f90, m3wndw.f90, mtxcalc.f90, pairstep.f90, presz.f90, timeshift.f90, vertimeproc.f90, vertintegral.f90, vertot.f90

New SAMPLE PROGRAMS page: updated to demonstrate I/O API-3.x capabilities and `MODULE`s, *.f90* source format and coding.

Removed *M3Tools* program *utmtool* for I/O API-3.2(deprecated in I/O API-3.0), in favor of program *projtool* which has greatly extended capabilities.

Back to ***Contents***

New SUBROUTINEs and FUNCTIONs

INTERFACE-specific forms:

needed for MODULE M3UTILIO to provide INTERFACEs for multiple argument-rank cases, as described in the section on Fortran-90 Generics, above.

SUBROUTINE LASTTIME(SDATE,STIME,TSTEP,NRECS, EDATE,ETIME): *I/O API-3.1*

computes the last date and time for a time step sequence, robustly avoiding INTEGER-overflow problems (Limit: NRECS overflows beyond about 68 years with 1-sec timestep, or 8947 years with 1-hour timestep. I've done 33-year runs with no problems, with I/O API-3.1 codes).

GETNUM(), GETREAL(), GETDBLE():

Have only arguments for DEFAULT and PROMPT, but not LO, HI as in GETNUM(), GETREAL(), GETDBLE()

BENVINT(), BENVDBLE(), BENVREAL():

accept bounded ranges of inputs: *do* have LO, HI arguments

M3TOGTPZ(),XY2XY(),GRID2XY(), GRID2INDX(), PNTS2INDX(), INDXMULT():

in MODULE MODGCTP

FINDL1(), FINDL2(), FINDL3(), FINDL4(), LOCATL1(), LOCATL2(), LOCATL3(), LOCATL4(), SORTL1(), SORTL2(), SORTL3(), SORTL4():

...for INTEGER*8 key-tuples

SORTINC4(), SORTINC8(): *I/O API-3.1*

...only the first N characters are significant.

SORTIC8(), SORTINC8():

...use INTEGER*8 table-subscripting, and require compilation for the Medium-64 memory model to be useful (*-mcmodel=medium* for *ifort*). [SMOKE-CARB request where they want to have more than 2G "sources" for program *smkreport*)

Back to ***Contents***

---

gfortran hacks

Recent versions of *gfortran* insist on breaking compatibility with all other compilers we use (and with earlier versions of *gfortran*), not accepting command-line-argument routines IARGC() and GETARG() (which are an industry standard Fortran extension), and accepting only Fortran-2003 routines GET_COMMAND_ARGUMENT() and COMMAND_ARGUMENT_COUNT(). This version conditionally implements IARGC() and GETARG() internally in *init3.F*, depending upon preprocessor-definition -DNEED_ARGS=1 which is now set in *Makeinclude.Linux2_x86*, *Makeinclude.Linux2_x86gfort*, *Makeinclude.Linux2_x86_64,* and *Makeinclude.Linux2_x86_64gfort*. Depending upon *gfortran* version, you either need to have it, or need *not* to have it.

Back to ***Contents***

Climatology-Year Versions

You can build the I/O API to use a 360-day or 365-day climatology year, using the following binary types and `Makeinclude` files:

Makeinclude.Linux2_x86_64_360
Makeinclude.Linux2_x86_64_365
Makeinclude.Linux2_x86_64gfort_360
Makeinclude.Linux2_x86_64gfort_365
Makeinclude.Linux2_x86_64ifort_360
Makeinclude.Linux2_x86_64ifort_365
Makeinclude.Linux2_x86_64pg_360
Makeinclude.Linux2_x86_64pg_365

Note that you need to keep these builds distinct from the (not-compatible) "normal" builds.

Back to ***Contents***

Memory-Model Issues

See Notes on `Linux2_x86_64` memory models
There are three different memory-models for programs on 64-bit `x86_64` Linux machines, which offer differing degrees of support for huge (>2 GB) arrays and huge object-files. Most compilers default to the "small" memory model, which suffices for the vast majority of uses. You can build the I/O API to use the "medium" memory model (at some cost in performance), using the following binary types and `Makeinclude` files:

Makeinclude.Linux2_x86_64gfort_medium
Makeinclude.Linux2_x86_64ifort_medium
Makeinclude.Linux2_x86_64pg_medium

Compiler-flags for memory model tend to take one of the following forms:

```
-mcmodel=small
-mcmodel=medium
-mcmodel=large
```

For CMAQ, you will need the medium memory model if your full `CONC` field is larger than 2 GB, i.e., if your grid dimensions are much larger than about
300 rows × 400 columns × 40 layers × 100 species (or equivalent).

Note that you need to build all model-components with the same set of memory-model flags and to keep these builds distinct from the (not-compatible) "normal" builds.

Back to ***Contents***

Other Issues

`INTEGER` Overflow Issues

Note that time intervals coded `HHMMSS` will suffer INTEGER-overflow after a little more than 24.5 years; however, a number of I/O API routines—*CURREC(), CURRSTEP(), JSTEP3(), LASTTIME(), NEXTTIME()*—are carefully coded so as to avoid such overflow, and can be used for time periods as long as several thousand years.

Programs that use starting date and time and run-duration (formatted `YYYYDDD`, `HHMMSS`, and `HHMMSS`) as run-control parameters will necessarily have the matching overflow problems; it is recommended that starting and ending dates and times be used instead. This latter style is safe for multi-century runs...

Environment variables

...of length up to 64K are now supported internally for I/O API-3.1 and later. Note that your operating system may well barf over this kind of thing — POSIX only mandates up to 512 bytes ;-(

GET_ENVLIST()

Why re-invent the wheel? ***un-safely?***
...and repeatedly? (there being 5 different copies of this routine in the CMAQ-5.0.1 source!)
There was *already* a well-written routine `STRLIST()` with the same functionality, except that it did so safely, with bounds-checking: by putting a too-long list into a CMAQ script, I can crash CMAQ any time I want! This would *not* be true if it used the already-written I/O API routines.

CHECKMEM()

Note that this is mostly found in SMOKE-descended codes, and in SMOKE itself...
Versions of this routine prior to the one found in SMOKE-4.0 *violate Models-3 standards* by suppressing logging of the I/O-status argument, which when non-zero is is the actual reason for program failure — in one case I know of it cost more than two person-weeks of debugging a development-version of SMOKE program *movesmrg*, before I replaced this routine by one which did report the I/O status, at which point it took me about ten minutes to re-compile and do a test-run, five more minutes to look up the failure-status in Intel's web-docs, and then a final ten minutes to find and fix the problem.

Moreover, `CHECKMEM()` use both suppresses optimization opportunities and at the same time makes the code less readable. For example, compare this example taken from *BDSNP_MOD.F* (which hides its actual content among a forest of `CHECKMEM`s):

```
ALLOCATE( SOILM( NCOLS,NROWS ), STAT=IOS )
CALL CHECKMEM( IOS, 'SOILM', PNAME )

ALLOCATE( SOILMPREV( NCOLS,NROWS ), STAT=IOS )
CALL CHECKMEM( IOS, 'SOILMPREV', PNAME )

ALLOCATE( SOILT( NCOLS,NROWS ), STAT=IOS )
CALL CHECKMEM( IOS, 'SOILT', PNAME )

ALLOCATE( ISLTYP( NCOLS,NROWS ), STAT=IOS )
CALL CHECKMEM( IOS, 'ISLTYP', PNAME )
```

```
            ALLOCATE( DRYPERIOD( NCOLS,NROWS ), STAT=IOS )
            CALL CHECKMEM( IOS, 'PTYPE', PNAME )

            ALLOCATE( NDEPRES( NCOLS,NROWS ), STAT=IOS )
            CALL CHECKMEM( IOS, 'PTYPE', PNAME )

            ALLOCATE( NDEPRATE( NCOLS,NROWS ), STAT=IOS )
            CALL CHECKMEM( IOS, 'PTYPE', PNAME )

            NDEPRATE = 0.0

            ALLOCATE( PFACTOR( NCOLS,NROWS ), STAT=IOS )
            CALL CHECKMEM( IOS, 'PFACTOR', PNAME )

            ALLOCATE( ARID( NCOLS,NROWS ), STAT=IOS )
            CALL CHECKMEM( IOS, 'ARID', PNAME )

            ALLOCATE( NONARID( NCOLS,NROWS ), STAT=IOS )
            CALL CHECKMEM( IOS, 'NONARID', PNAME )

            ALLOCATE( LANDFRAC( NCOLS,NROWS ), STAT=IOS )
            CALL CHECKMEM( IOS, 'LANDFRAC', PNAME )

            ALLOCATE( FERT( NCOLS,NROWS ), STAT=IOS )
            CALL CHECKMEM( IOS, 'FERT', PNAME )

            ALLOCATE( T1_NH3( NCOLS,NROWS ), STAT=IOS )
            CALL CHECKMEM( IOS, 'T1_NH3', PNAME )

            ALLOCATE( T1_NO3( NCOLS,NROWS ), STAT=IOS )
            CALL CHECKMEM( IOS, 'T1_NO3', PNAME )

            ALLOCATE( T1_ON( NCOLS,NROWS ), STAT=IOS )
            CALL CHECKMEM( IOS, 'T1_ON', PNAME )

            ALLOCATE( EPICN( NCOLS,NROWS ), STAT=IOS )
            CALL CHECKMEM( IOS, 'EPICN', PNAME )

            ALLOCATE( CRF( NCOLS,NROWS ), STAT=IOS )
            CALL CHECKMEM( IOS, 'CRF', PNAME )

            ALLOCATE( CRFAVG( NCOLS,NROWS ), STAT=IOS )
            CALL CHECKMEM( IOS, 'CRF', PNAME )

            ALLOCATE( PULSEAVG( NCOLS,NROWS ), STAT=IOS )
            CALL CHECKMEM( IOS, 'PULSEAVG', PNAME )

            ALLOCATE( BASESUM( NCOLS,NROWS ), STAT=IOS )
            CALL CHECKMEM( IOS, 'BASESUM', PNAME )

        C ------ Diagnostics ----------------------------------
            ALLOCATE( THETA_DIAG( NCOLS,NROWS ), STAT=IOS )
            CALL CHECKMEM( IOS, 'THETA_DIAG', PNAME )

            ALLOCATE( WET_DIAG( NCOLS,NROWS ), STAT=IOS )
            CALL CHECKMEM( IOS, 'WET_DIAG', PNAME )

            ALLOCATE( TEMP_DIAG( NCOLS,NROWS ), STAT=IOS )
            CALL CHECKMEM( IOS, 'TEMP_DIAG', PNAME )

            ALLOCATE( A_DIAG( NCOLS,NROWS ), STAT=IOS )
```

```
    CALL CHECKMEM( IOS, 'A_DIAG(', PNAME )

    ALLOCATE( AFERT_DIAG( NCOLS,NROWS ), STAT=IOS )
    CALL CHECKMEM( IOS, 'AFERT_DIAG', PNAME )

    ALLOCATE( NRES_FERT_DIAG( NCOLS,NROWS ), STAT=IOS )
    CALL CHECKMEM( IOS, 'NRES_FERT_DIAG', PNAME )

    ALLOCATE( NRES_DEP_DIAG( NCOLS,NROWS ), STAT=IOS )
    CALL CHECKMEM( IOS, 'NRES_DEP_DIAG', PNAME )
```

with the following equivalent code, which I find much more readable:

```
    ALLOCATE( SOILM( NCOLS,NROWS ),
&         SOILMPREV( NCOLS,NROWS ),
&             SOILT( NCOLS,NROWS ),
&            ISLTYP( NCOLS,NROWS ),
&         DRYPERIOD( NCOLS,NROWS ),
&           NDEPRES( NCOLS,NROWS ),
&          NDEPRATE( NCOLS,NROWS ),
&           PFACTOR( NCOLS,NROWS ),
&              ARID( NCOLS,NROWS ),
&           NONARID( NCOLS,NROWS ),
&          LANDFRAC( NCOLS,NROWS ),
&              FERT( NCOLS,NROWS ),
&            T1_NH3( NCOLS,NROWS ),
&            T1_NO3( NCOLS,NROWS ),
&             T1_ON( NCOLS,NROWS ),
&             EPICN( NCOLS,NROWS ),
&               CRF( NCOLS,NROWS ),
&            CRFAVG( NCOLS,NROWS ),
&          PULSEAVG( NCOLS,NROWS ),
&           BASESUM( NCOLS,NROWS ),
&        THETA_DIAG( NCOLS,NROWS ),    !!  diagnostic variables:
&          WET_DIAG( NCOLS,NROWS ),
&         TEMP_DIAG( NCOLS,NROWS ),
&            A_DIAG( NCOLS,NROWS ),
&        AFERT_DIAG( NCOLS,NROWS ),
&     NRES_FERT_DIAG( NCOLS,NROWS ),
&      NRES_DEP_DIAG( NCOLS,NROWS ), STAT=IOS )
    IF ( IOS .NE. 0 ) THEN
        WRITE( MESG, '(A, I9)' )
&          'ERROR allocating SOILM...NRES_DEP_DIAG.  STATUS=', IOS
        CALL M3EXIT( PNAME, 0,0, MESG, 2 )
    END IF

    NDEPRATE = 0.0
```

Array-of-*TYPE* code architecture

If you take nVidia's courses on GPU programming, one of the very first things they will say (using C-programmer language, *struct* instead of Fortran-90 *TYPE*) is:

> Many codes can be structured either as *array-of-structs* or as *struct-of-arrays* (or even as parallel arrays). ***Array-of-structs* code will kill GPU-computing performance.**

And actually the same is true for for killing performance on the upcoming Intel PHI Xeons and, though to a lesser degree, for current cache based microprocessors...

Module names

With a very few exceptions (Absoft, PathScale, Cray), Fortran-90 compiler behavior for generating MODULE-file names is

> Downcase the MODULE-name
> Add a trailing *.mod*

Therefore, if we name MODULE-source files by a similar pair of rules:

> One MODULE per source-file
> *Optionally, name all modules "mod<something>" so that module-source files are immediately obvious*
> Downcase the MODULE-name for the base of the source-file name, and add a trailing *.f* or *.f90*, as appropriate.
> For example MODULE MODQUX should live in source-file *modqux.f90* or...

then we can put rule based—and correct!— MODULE and precise dependency handling into *Makefiles*, and avoid a number of problems caused by the hacks found in the current systems. Here is an example of some of the rules:

```
.SUFFIXES: .m4 .c .F .f ..f90 .mod
...
##########  Rules:
%.o : %.mod        #  Disable "gmake"s obnoxious implicit Modula-2 rule !!
%.f : %.F          #  Hack for some versions of  "gmake" + "gfortran"
.f.mod:
        cd ${OBJDIR}; $(FC) -c $(FFLAGS) ${SRCDIR}/$<
.f90.mod:
        cd ${OBJDIR}; $(FC) -c $(FFLAGS) ${SRCDIR}/$<
...
##########  Dependencies:

foo.o : modbar.mod m3utilio.mod
...
```

By way of further example, the *Makefile* for one of my hydrology models has the following precise set of MODULE-related dependency rules:

```
mod_noah.mod  mod_noah.o  : mod_calibparms.mod
mod_route.mod mod_route.o : mod_noah.mod mod_rteparms.mod mod_calibparms.mod
chanadj.o    : mod_rteparms.mod
chaninit.o   : mod_rteparms.mod
gis2route.o  : mod_rteparms.mod
gridinit.o   : mod_route.mod mod_noah.mod mod_rteparms.mod mod_calibparms.mod
mpoolinit.o  : mod_rteparms.mod
nldastomet.o : ${LIBDIR}/modgribio.mod
nldasmask.o  : ${LIBDIR}/modgribio.mod
qbaseinit.o  : mod_noah.mod mod_rteparms.mod mod_calibparms.mod
reflex.o     : mod_noah.mod mod_route.mod libnoah_phys.a
...
```

Back to ***Contents***

To-Do List/Discussion

Just to get things started:

- Define standard *SMOKE* metadata and the interface(s) by which it is provided. Then finish the relevant routines in `MODULE MODATTS3` accordingly.
- Add a `MODULE` that supports routines for GIS-raster-format I/O (`INTEGER*[1,2,4]` BIL, GRIDFLOAT, ARC-ASCII, ASC-ASCII, etc., as well as *gzip*ped forms of the same)?
- Tension-spline interpolation module and tools-programs?
- High-performance production-graphics programs (*m3plot, mtxplot, ncfplot, gisplot*)?
- *M3Tools* development—new codes with standardized, overflow-proof (starting date&time; ending date&time) interfaces, free-format F90, OpenMP-parallel where reasonable?

Back to ***Contents***

Send comments to Carlie J. Coats, Jr.
cjcoats@email.unc.edu