A time manager for MPAS

MPAS Development Team

April 21, 2011

Contents

Chapter 1

Summary

In the current MPAS code, time is simply tracked as the number of time steps since the beginning of the simulation; with the time step length also known, the code can multiply the time step length by the number of steps taken to find the elapsed time. However, because there is no calendar date and time associated with the start of the model simulation, there is no concept of 'real time' in a simulation — that is, a simulation has no way of distinguishing whether it is currently, say, January or July.

When performing real-data simulations of the atmosphere (and perhaps of other physical domains, too), certain physics schemes must know the calendar date and time at each step in the model integration. For example, a short-wave radiation scheme needs to know the solar zenith angle, which it computes from the date and time of day. To someone looking at model output, it would also be important to know the date and time for which a particular set of fields is valid; without such information, validating model simulations becomes nearly impossible.

Besides lacking the date and time at any point in a model simulation, the current MPAS time-keeping method provides no unified mechanism for identifying when arbitrarily defined points in the simulation have been reached, at which time some action may be taken. For example, the work of determining whether it is time to perform I/O currently falls on the driver code, which uses the current time step modulo some output interval specified in time steps; adjustments to the model time step are therefore usually accompanied by changes to the output interval. Using a sophisticated, future I/O subsystem in MPAS, we may like to perform I/O on multiple streams (e.g., files) at different intervals specified in natural time units rather than model time steps; in such cases, the simple-minded approach that is currently used becomes cumbersome, especially if the future I/O implementation is to be run-time configurable in terms of the number of streams.

To permit progress on the addition of physics schemes and the re-implementation of the MPAS I/O sub-system in a more general and robust way, this document proposes the addition of a new time management system, which will be general enough to be used throughout the MPAS code. This new system should be useful for tracking the date and time, providing notifications when pre-specified points in time have been reached, and tracking the progress through time of a model simulation (i.e., model time-stepping).

Chapter 2

Requirements

Date last modified: 2011/03/28
Contributors: Michael Duda, Phil Jones

2.1 General date/time tracking

The most fundamental requirement for the new MPAS time management system is the ability to track the current date/time through the course of a model simulation.

2.1.1 Set the starting and ending date/time

Recognizing that there is always a finite amount of wallclock time that can be devoted to any simulation, it follows that every simulation is associated with a starting and ending *simulation* date/time. The MPAS time manager is required to maintain these two points explicitly, since their relation to the current simulation time determines whether a simulation has completed or not. In the case of 'cycled' simulations, which in principle could be extended indefinitely given an infinite amount of computer time, the ending date/time represents the end of the simulation cycle currently being run. It must be possible for the user to reset the starting or ending time at any point during the model simulation.

2.1.2 Determine whether the starting or ending time has been reached

Determining whether the ending time has been reached or exceeded is a requirement for knowing when integration should stop in MPAS. For integration backward in time, the same requirement holds, except the role of the ending time is fulfilled by the starting time.

2.1.3 Set the default time increment

An MPAS simulation generally makes use of a fixed, default time step length, and the time manager must be able to record this time step so that the date/time can be incremented without the need to specify the increment explicitly each time step. It must also be possible for the user to reset the default time step at any point during the model simulation. The user must be able to specify this default time increment as a time interval in any supported unit (see Units).

2.1.4 Advance the date/time by an arbitrary time increment

In future, we may like to implement the ability in MPAS to vary the time step over the course of a simulation, according to criteria such as the maximum Courant number in the previous time step; therefore, the time manager must be able to advance the date/time by an arbitrary time increment in addition to incrementing by the default time step.

2.1.5 Handle both forward and backward time increments

The ability to 'advance' the date/time by a negative time increment — that is, to step backward through time — may be useful for further model development. For example, the addition of a digital filtering initialization scheme would require the ability to integrate both forward and backward in time, and, therefore, to increment the date/time with both positive and negative increments. The MPAS time manager must support both forward and backward increments.

2.1.6 Perform time computation with no round-off using exact arithmetic

The ability of the time manager to track time using exact arithmetic is necessary so that, for example, events take place at the proper time, and not at the time step before or after due to rounding errors in the date/time computation. Simulations of up to 100,000 years without roundoff accumulation are required.

2.2 Notifications

Upcoming developments in MPAS will require the ability to determine when pre-specified points in the model simulation have been reached. For example, we may want to update tendencies from physics parameterizations periodically at an interval longer than the time step. Another planned development that would require the ability to perform actions at pre-determined points is in the I/O system, where we may want to read (in the case of boundary conditions, for example) or write fields to any of several I/O streams periodically. In general, we require the time manager to have the ability to provide notifications to code when specified points in time have been reached.

2.2.1 Add notifications

Clearly, the ability to specify the instants in time where notifications are to be provided is required of the time manager. It must be possible to add notifications at any point in program execution, so that not all notifications need to be added at initialization time.

2.2.2 One-time notification

The time manager will support notifications that only ring once for a particular time.

2.2.3 Recurring notification

The time manager must support the setting of notifications which will recur with a fixed period, since some events (e.g., boundary updates, diagnostics, I/O) occur periodically.

2.2.4 Query for notifications

The user must be able to query whether a notification or alarm is "ringing". It should be possible to query whether any alarms are currently ringing, or whether any alarms will begin to ring within a specified time interval from the current time.

2.2.5 Clear a notification

In order that a particular notification not be provided by the time manager in subsequent queries, it must be possible to clear a notification. Clearing an instance of a periodically recurring notification will only clear that particular instance.

2.2.6 Remove a notification

For both periodic and 'one-off' notifications, it must be possible to remove those notifications from the time manager to prevent them from being provided in future queries for notifications.

2.3 Time instants and time intervals

The concepts of 'time instants' and 'time intervals' are fundamental to the task of tracking time and providing notifications, so it is necessary that the MPAS time manager be able to perform basic manipulation of them.

2.3.1 Time arithmetic

The time manager will be able to:

- compute a new time instant given a time instant and a time interval (either negative or positive);
- compute a new time interval as the sum or difference of two other time intervals;
- compute the (positive or negative) time interval between two time instants;
- multiply a time interval by an integer constant;
- divide a time interval by an integer constant; and
- compute the remainder (as a time interval) when dividing one time interval by another.

2.4 Calendars and units

The time manager must be able to track time in a variety of units and calendars.

2.4.1 Calendars

The time manager must, at minimum, support a Gregorian calendar, and must support the ability to enable or disable leap years. Other calendars that may need to be supported are 360-day (30-day equal month) calendars or Julian Day calendars.

2.4.2 Units

The time manager must support time intervals and time instants in units of hours, minutes, seconds, days, months and years (and fractions thereof). The underlying representation can be in arbitrary units (to support the roundoff requirements above), but user queries and arguments must support these units.

2.4.3 Unit conversion

A set of utilities must be available to convert units for time-related quantities (e.g. days to seconds, etc.). Some conversions may require knowledge of the calendar choice.

Chapter 3

Design

In order to make the MPAS time manager as re-usable as possible, and to facilitate incremental implementation of the time manager in various parts of the MPAS code, we envision the MPAS time manager as a stand-alone module that provides opaque data types for clocks (instantiations of the time manager), time instants, and time intervals; additionally, the module will provide a set of routines for manipulating these types. Since the internal representation of clocks, time instants, and time intervals should remain unknown to code outside the module, we only describe the routines to be provided by the module.

3.1 General date/time tracking

Date last modified: 2011/04/21
Contributors: Michael Duda

The routines used to create, destroy, and manipulate instantiations of the MPAS time manager (clocks) are described below. There is no limit to the number of concurrent clocks that can be created and used by code that employs the MPAS time manager.

---

subroutine

MPAS_createClock(clock, calendar, startTime, timeStep, stopTime, runDuration, ierr)

Creates a clock object given a calendar type, a starting time, a default time step, and either a stop time or a run duration.

Input

integer :: calendar — *The calendar to use for the clock, either GREGORIAN or GREGORIAN_NOLEAP*

MPAS_Time_type :: startTime — *The start time of the clock*

MPAS_TimeInterval_type :: timeStep — *The default time step of the clock*

MPAS_Time_type, optional :: stopTime — *The end time of the clock, which equals (startTime + runDuration)*

MPAS_TimeInterval_type, optional :: runDuration — *The run duration of the clock, defined as (stopTime - startTime)*

Output

MPAS_Clock_type :: clock — *An initialized MPAS clock*

integer :: ierr — *The return error code*

---

subroutine

MPAS_destroyClock(clock, ierr)

Destroys a clock object.

Input

MPAS_Clock_type :: clock — *The clock to be destroyed*

Output

integer :: ierr — *The return error code*

---

function

MPAS_isClockStartTime(clock, ierr)

Tells whether the current time on the clock is at or before the clock's start time.

Return value

A logical TRUE if the current time is at or before the start time, and FALSE otherwise.

Input

MPAS_Clock_type :: clock — *The clock to be queried*

Output

integer :: ierr — *The return error code*

---

function

MPAS_isClockEndTime(clock, ierr)

Tells whether the current time on the clock is at or past the clock's stop time.

Return value

A logical TRUE if the current time is at or past the stop time, and FALSE otherwise.

Input

MPAS_Clock_type :: clock — *The clock to be queried*

Output

integer :: ierr — *The return error code*

---

subroutine

MPAS_setClockDirection(clock, direction, ierr)

Sets the clock's direction of advance to be either positive/forward or negative/backward.

Input

MPAS_Clock_type :: clock — *The clock whose direction is to be set*

integer :: direction — *Either of the module-defined constants FORWARD or BACKWARD*

Output

integer :: ierr — *The return error code*

---

subroutine

MPAS_getClockDirection(clock, direction, ierr)

Retrieves the clock's direction of advance, either positive/forward or negative/backward.

Input

MPAS_Clock_type :: clock — *A clock*

Output

integer :: direction — *Either of the module-defined constants FORWARD or BACKWARD*

integer :: ierr — *The return error code*

---

subroutine

MPAS_setClockTimeStep(clock, timeStep, ierr)

Sets the clock's default time step.

Input

MPAS_Clock_type :: clock — *The clock whose time step is to be set*

MPAS_TimeInterval_type :: timeStep — *The new default time increment for the clock*

Output

integer :: ierr — *The return error code*

---

subroutine

MPAS_getClockTimeStep(clock, timeStep, ierr)

Retrieves the clock's default time step.

Input

MPAS_Clock_type :: clock — *A clock*

Output

MPAS_TimeInterval_type :: timeStep — *The clock's default time increment*

integer :: ierr — *The return error code*

---

subroutine

MPAS_advanceClock(clock, timeStep, ierr)

Advances the clock's current time by the specified time step, or by the clock's default time step if no explicit step is specified.

Input

MPAS_Clock_type :: clock — *The clock to be advanced*

MPAS_TimeInterval_type, optional :: timeStep — *A time increment to be used in place of the clock's default increment*

Output

integer :: ierr — *The return error code*

---

subroutine

MPAS_setClockTime(clock, clock_time, whichTime, ierr)

Sets the clock's start, stop, or current time depending on the whichTime argument.

Input

MPAS_Clock_type :: clock — *The clock whose time is to be set*

MPAS_Time_type :: clock_time — *The time instant to become the start, stop, or current time of the clock*

integer :: whichTime — *Which time to set, either START_TIME, STOP_TIME, or NOW*

Output

integer :: ierr — *The return error code*

---

subroutine

MPAS_getClockTime(clock, whichTime, clock_time, ierr)

Gets the clock's start, stop, or current time, depending on the whichTime argument.

Input

MPAS_Clock_type :: clock — *The clock whose time is to be retrieved*

integer :: whichTime — *Which time to get, either START_TIME, STOP_TIME, or NOW*

Output

MPAS_Time_type :: clock_time — *The start, stop, or current time of the clock*

integer :: ierr — *The return error code*

3.2 Notifications

Date last modified: 2011/04/21
Contributors: Michael Duda

Rather than using non-intrinsic data types to represent notifications, the MPAS time manager will identify notifications as user-specified integer ID numbers relative to clocks (i.e., the same ID can refer to different notifications in different clocks).

---

subroutine

MPAS_addClockAlarm(clock, alarmID, alarmTime, alarmTimeInterval, relativeToTime, recurring, ierr)

Adds an alarm to the clock, with the nature of the alarm — either 'one-off' or recurring — depending on the combination of arguments.

Input

MPAS_Clock_type :: clock — *The clock to which the alarm will be added*

integer :: alarmID — *The alarmID that will be used to identify the alarm*

MPAS_Time_type :: alarmTime — *The time of the alarm*

MPAS_TimeInterval_type :: alarmTimeInterval — *The interval of the alarm*

MPAS_Time_type :: relativeToTime — *The the base time that the interval is relative to*

logical :: recurring — *Whether or not the alarm is a recurring alarm*

Output

integer :: ierr — *The return error code*

---

function

MPAS_isAlarmRinging(clock, alarmID, interval, ierr)

Tells whether the specified alarm is ringing on the clock, or whether it will begin ringing within the optional time interval from the present.

Return value

A logical TRUE if the alarm is ringing and FALSE otherwise.

Input

MPAS_Clock_type :: clock — *The clock holding the alarm to be queried*

integer :: alarmID — *The alarmID to be queried*

MPAS_TimeInterval_type, optional :: interval — *The interval over which to consider the alarm*

Output

integer :: ierr — *The return error code*

---

subroutine

MPAS_removeClockAlarm(clock, alarmID, ierr)

Removes the specified alarm from the clock.

Input

MPAS_Clock_type :: clock — *The clock from which the alarm will be removed*

integer :: alarmID — *The alarmID to remove from the clock*

Output

integer :: ierr — *The return error code*

---

subroutine

MPAS_resetClockAlarm(clock, alarmID, ierr)

Resets the specified alarm if it is ringing.

Input

MPAS_Clock_type :: clock — *The clock holding the alarm to be reset*

integer :: alarmID — *The alarmID to be reset*

Output

integer :: ierr — *The return error code*

---

subroutine

MPAS_getClockRingingAlarms(clock, nAlarms, alarmList, interval, ierr)

Returns a list of all the alarms that are currently ringing, or will begin ringing within the optional time interval, including those that were ringing before the current time but had not been reset.

Input

MPAS_Clock_type :: clock — *A clock*

MPAS_TimeInterval_type :: interval — *The interval over which to consider alarms*

Output

integer :: nAlarms — *The number of alarms returned in the alarmList*

integer, dimension(MAX_ALARMS) :: alarmList — *A list of IDs for alarms that are ringing*

integer :: ierr — *The return error code*

3.3 Time instants and time intervals

Date last modified: 2011/03/28
Contributors: Michael Duda

Time intervals and time instants in the MPAS time manager are 'shallow' types, and require no *create* or *destroy* routines. In this section, the routines used to set, get, and format time instants and time intervals are described.

---

subroutine

MPAS_setTime(curr_time, YYYY, MM, DD, H, M, S, S_n, S_d, dateString, ierr)

Sets the date and time of a time instant. If a dateString is given, the integer arguments are ignored.

Input

integer, optional :: YYYY — *The year*

integer, optional :: MM — *The month*

integer, optional :: DD — *The day*

integer, optional :: H — *The hour*

integer, optional :: M — *The minute*

integer, optional :: S — *The second*

integer, optional :: S_n — *The numerator of a rational fraction of a second*

integer, optional :: S_d — *The denominator of a rational fraction of a second*

character(len=*), optional :: dateString — *The date/time in the form YYYY-MM-DD[_H[:M[:S]]]*

Output

MPAS_Time_type :: curr_time — *The time instant described by the input arguments*

integer :: ierr — *The return error code*

---

subroutine

MPAS_getTime(curr_time, YYYY, MM, DD, H, M, S, S_n, S_d, dateString, ierr)

Gets the date and time of a time instant. If a dateString is given, none of the integer arguments will be set.

Input

MPAS_Time_type :: curr_time — *A time instant*

Output

integer, optional :: YYYY — *The year*

integer, optional :: MM — *The month*

integer, optional :: DD — *The day*

integer, optional :: H — *The hour*

integer, optional :: M — *The minute*

integer, optional :: S — *The second*

integer, optional :: S_n — *The numerator of a rational fraction of a second*

integer, optional :: S_d — *The denominator of a rational fraction of a second*

character(len=*), optional :: dateString — *The date/time in the form YYYY-MM-DD_H:M:S.SSS*

integer :: ierr — *The return error code*

---

subroutine

MPAS_setTimeInterval(interval, YYYY, MM, DD, H, M, S, S_n, S_d, dateString, ierr)

Sets the length of a time interval. The integer arguments are cumulative; if a date string is given, the integer arguments are ignored.

Input

integer, optional :: YYYY — *The number of years in the interval*

integer, optional :: MM — *The number of months*

integer, optional :: DD — *The number of days*

integer, optional :: H — *The number of hours*

integer, optional :: M — *The number of minutes*

integer, optional :: S — *The number of whole seconds*

integer, optional :: S_n — *The numerator of a rational fraction of a seconds*

integer, optional :: S_d — *The denominator of a rational fraction of a seconds*

character(len=*), optional :: dateString — *The time interval in the form YYYY-MM-DD[_H[:M[:S]]]*

Output

MPAS_TimeInterval_type :: interval — *The time interval described by the input arguments*

integer :: ierr — *The return error code*

---

subroutine

MPAS_getTimeInterval(interval, YYYY, MM, DD, H, M, S, S_n, S_d, dateString, ierr)

Gets the length of a time interval. If a dateString is given, none of the integer arguments will be set.

Input

MPAS_TimeInterval_type :: interval — *A time interval*

Output

integer, optional :: YYYY — *The number of years in the interval*

integer, optional :: MM — *The number of months*

integer, optional :: DD — *The number of days*

integer, optional :: H — *The number of hours*

integer, optional :: M — *The number of minutes*

integer, optional :: S — *The number of whole seconds*

integer, optional :: S_n — *The numerator of a rational fraction of a seconds*

integer, optional :: S_d — *The denominator of a rational fraction of a seconds*

character(len=*), optional :: dateString — *The time interval in the form YYYY-MM-DD_H:M:S.SSS*

integer :: ierr — *The return error code*

In addition to the above routines, operators will be defined for the following operations:

- time = time + interval
- time = time - interval
- interval = interval + interval
- interval = interval - interval
- interval = interval % interval
- interval = time - time
- interval = interval * n
- interval = interval / n
- interval = -interval

3.4 Calendars and units

Date last modified: 2011/03/28
Contributors: Michael Duda

Perhaps there is nothing much to say about units?

Unit conversions are handled by the design of the interfaces for setting and getting time instants and time intervals.

Chapter 4

Proposed Plan for Implementation

Date last modified: 2011/04/21
Contributors: Michael Duda

Roughly speaking, the implementation of the time manager will follow these steps:

1. Implement the routines and operators described in Chapter 3 in a single, independent module to be located in the framework directory.
2. Add the necessary code throughout the framework so that a 'master' clock for a simulation will be initialized based on namelist parameters for the starting time, ending time, and time step.
3. Add code so that the master clock is updated during the course of model simulation by the mpas_run() routines of individual MPAS cores.
4. Write and read date/time information from the clock to input, output, and restart files in the same way as the current xtime variable.

Chapter 5

Testing and Validation

5.1 Testing and Validation: XXX

Date last modified: 2011/01/05
Contributors: (add your name to this list if it does not appear)

TBD