NAME

M_time - [M_time] Fortran module for manipulating and presenting time and date values

DESCRIPTION

The M_time(3f) Fortran module and associated utility programs provide date and time-related procedures. Both a procedural and OOP (Object Oriented Programming) interface are provided. Each routine is accompanied by a man(1) page which includes a sample program for that procedure. This manual, the source and example programs are included in the download.

The M_time(3f) module

- provides for formatting dates.
- facilitates simple computations using time and date values in the recent era.
- allow for macro-level timing of code.

The M_TIME(3f) module complements the DATE_AND_TIME(3f) procedure, which is the standard intrinsic subroutine that returns the current date and time in the Gregorian calendar. That is, the primary way this module represents dates is as an integer array with the same meaning for elements as defined by the DATE_AND_TIME(3f) routine. In addition it can calculate or read many other date representations such as ...

- Julian Dates
- Unix Epoch Dates
- High-level date formatting
- Ordinal days of the year
- days of the week
- ISO-8601 week numbers
- month and weekday names

Julian and Unix Epoch Dates are particularly useful for manipulating dates in simple numeric expressions.

The extensive formatting options include showing SYSTEM_CLOCK(3f) and CPU_USAGE(3f) information along with Gregorian date information, allowing for the easy incorporation of timing information into program messages. In addition to conventional Civilian Calendar dates, the module supports the ISO-8601 standard methods of displaying dates.

SYNOPSIS

UNIX EPOCH		
date_to_unix(dat,UNIXTIME,IERR)	%epoch()	Convert date array to Unix Time
unix_to_date(unixtime,DAT,IERR)		Convert Unix Time to date array
d2u(dat) result (UNIXTIME)		Convert date array to Unix Time
u2d(unixtime) result (DAT)		

		Convert Unix Time to date array
JULIAN		
julian_to_date(julian,DAT,IERR)		Convert Julian Date to date array
date_to_julian(dat,JULIAN,IERR)	%julian()	Converts date array to Julian Date
d2j(dat) result (JULIAN)		Convert date array to Julian Date
j2d(julian) result (DAT)		Convert Julian Date to date array
DAY OF WEEK		
dow(dat,[WEEKDAY],[DAY],IERR)	%weekday()	Convert date array to day of the week as number(Mon=1) and name
WEEK OF YEAR		
d2w(dat,ISO_YEAR,ISO_WEEK,ISO_WEEKDAY,ISO_NAME)		calculate iso-8601 Week-numbering year date yyyy-Www-d
w2d(iso_year,iso_week,iso_weekday,DAT)		calculate date given iso-8601 Week date yyyy-Www-d
ORDINAL DAY		
d2o(dat) result(ORDINAL)	%ordinal()	given date array return ordinal day of year, Jan 1st=1
o2d(ordinal,[year]) result(DAT)		given ordinal day of year return date array, Jan 1st=1
ordinal_to_date(ordinal,year,DAT)		given ordinal day of year return date array, Jan 1st=1
ordinal_seconds()		return seconds since beginning of year
PRINTING DATES		
fmtdate(dat,format) result (TIMESTR)	%format([STRING])	

		Convert date array to string using format
fmtdate_usage(indent)		display macros recognized by fmtdate(3f)
now(format) result (NOW)		return string representing current time given format
box_month(dat,CALEN)		print specified month into character array
MONTH NAME		
mo2v(month_name) result (MONTH_NUMBER)		given month name return month number
v2mo(month_number) result (MONTH_NAME)		given month number return month name
mo2d(month_name) result (DAT)		return date array for first day of given month name in specified year
ASTROLOGICAL		
easter(year,dat)		calculate month and day Easter falls on for given year
moon_fullness(DAT) result(FULLNESS)		percentage of moon phase from new to full
phase_of_moon(DAT) result(PHASE)		return name for phase of moon for given date
DURATION		
sec2days(seconds) result(dhms)		converts seconds to string D-HH:MM:SS
days2sec(string) result(seconds)		converts string D-HH:MM:SS to seconds
READING DATES		
guessdate(anot,dat)		Converts a date string to a date array, in various formats

FORMATTING OPTIONS IN FMTDATE

You can easily use Julian Ephemeris Dates and Unix Epoch Times to add and subtract times from dates or to calculate the interval between dates. But JEDs and UETs and even the Gregorian Calendar arrays in the DAT arrays are not the way we typically describe a date on the Civilian Calendar. So the fmtdate(3f) routine lets us print a DAT array in a variety of familiar styles.

The fmtdate(3f) and now(3f) procedures let you display a Gregorian date using either keywords for standard formats or using macros in a user-specified formatting string. A formatting string may contain the following macros:

```
Description                                        Example

 Base time array:
  (1) %Y -- year, yyyy                                2016
  (2) %M -- month of year, 01 to 12                   07
  (3) %D -- day of month, 01 to 31                    27
      %d -- day of month, with suffix (1st, 2nd,...)  27th
  (4) %Z -- minutes from UTC                          -0240
      %z -- -+hh:mm from UTC                          -04:00
      %T -- -+hhmm  from UTC                          -0400
  (5) %h -- hours, 00 to 23                           21
      %H -- hour (1 to 12, or twelve-hour clock)      09
      %N -- midnight< AM <=noon; noon<= PM <midnight  PM
  (6) %m -- minutes, 00 to 59                         24
  (7) %s -- sec, 00 to 59                             22
  (8) %x -- milliseconds 000 to 999                   512
 Conversions:
      %E -- Unix Epoch time                           1469669062.5129952
      %e -- integer value of Unix Epoch time          1469669063
      %J -- Julian  date                              2457597.559
      %j -- integer value of Julian Date(Julian Day)  2457597
      %O -- Ordinal day (day of year)                 209
      %o -- whole days since Unix Epoch date          17009
      %U -- day of week, 1..7 Sunday=1                4
      %u -- day of week, 1..7 Monday=1                3
      %i -- ISO week of year 1..53                    30
      %I -- iso-8601 week-numbering date(yyyy-Www-d)  2016-W30-3
  Names:
      %l -- abbreviated month name                    Jul
      %L -- full month name                           July
      %w -- first three characters of weekday         Wed
      %W -- weekday name                              Wednesday
      %p -- phase of moon                             New
      %P -- percent of way from new to full moon      -1%
  Literals:
      %% -- a literal %                               %
      %t -- tab character
      %b -- blank character
      %B -- exclamation(bang) character
      %n -- new line (system dependent)
      %q -- single quote (apostrophe)
      %Q -- double quote
  Program timing:
      %c -- CPU_TIME(3f) output                       .78125000000000000E-001
```

```
        %C -- number of times this routine is used      1
        %S -- seconds since last use of this format     .0000000000000000
        %k -- time in seconds from SYSTEM_CLOCK(3f)     588272.750
        %K -- time in clicks from SYSTEM_CLOCK(3f)      588272750

  If no percent (%) is found in the format one of several
  alternate substitutions occurs.

  If the format is composed entirely of one of the following
  keywords the following substitution occurs:

    "iso-8601",
    "iso"        ==> %Y-%M-%DT%h:%m:%s%z  ! Example: 2017-08-26T18:56:33,510912700-04:

    "iso-8601W",
    "isoweek"    ==> %I
    "sql"        ==> "%Y-%M-%D %h:%m:%s.%x"
    "sqlday"     ==> "%Y-%M-%D"
    "sqltime"    ==> "%h:%m:%s.%x"
    "rfc-2822"   ==> %w, %D %l %Y %h:%m:%s %T  !  Example: Mon, 14 Aug 2006 02:34:56 -

    "rfc-3339"   ==> %Y-%M-%DT%h:%m:%s%z  !  Example: 2006-08-14 02:34:56-06:00
    "date"       ==> %w %l %D %h:%m:%s UTC%z %Y
    "short"      ==> %w, %l %d, %Y %H:%m:%s %N UTC%z
    "long"," "   ==> %W, %L %d, %Y %H:%m:%s %N UTC%z
    "suffix"     ==> %Y%D%M%h%m%s
    "formal"     ==> The %d of %L %Y
    "lord"       ==> the %d day of %L in the year of our Lord %Y
    "easter"     ==> FOR THE YEAR OF THE CURRENT DATE:
                       Easter day: the %d day of %L in the year of our Lord %Y
    "all"        ==> A SAMPLE OF DATE FORMATS
  otherwise the following words are replaced with the most
  common macros:

     STRING   MACRO  EXAMPLE
     year     %Y     2016
     month    %M     07
     day      %D     27
     hour     %h     21
     minute   %m     24
     second   %s     22
     epoch    %e     1469669063
     julian   %j     2457597
     ordinal  %O     209
     weekday  %u     3

  if none of these keywords are found then every letter that
  is a macro is assumed to have an implied percent in front
  of it. For example:

     YMDhms ==> %Y%M%D%h%m%s ==> 20160727212422
```

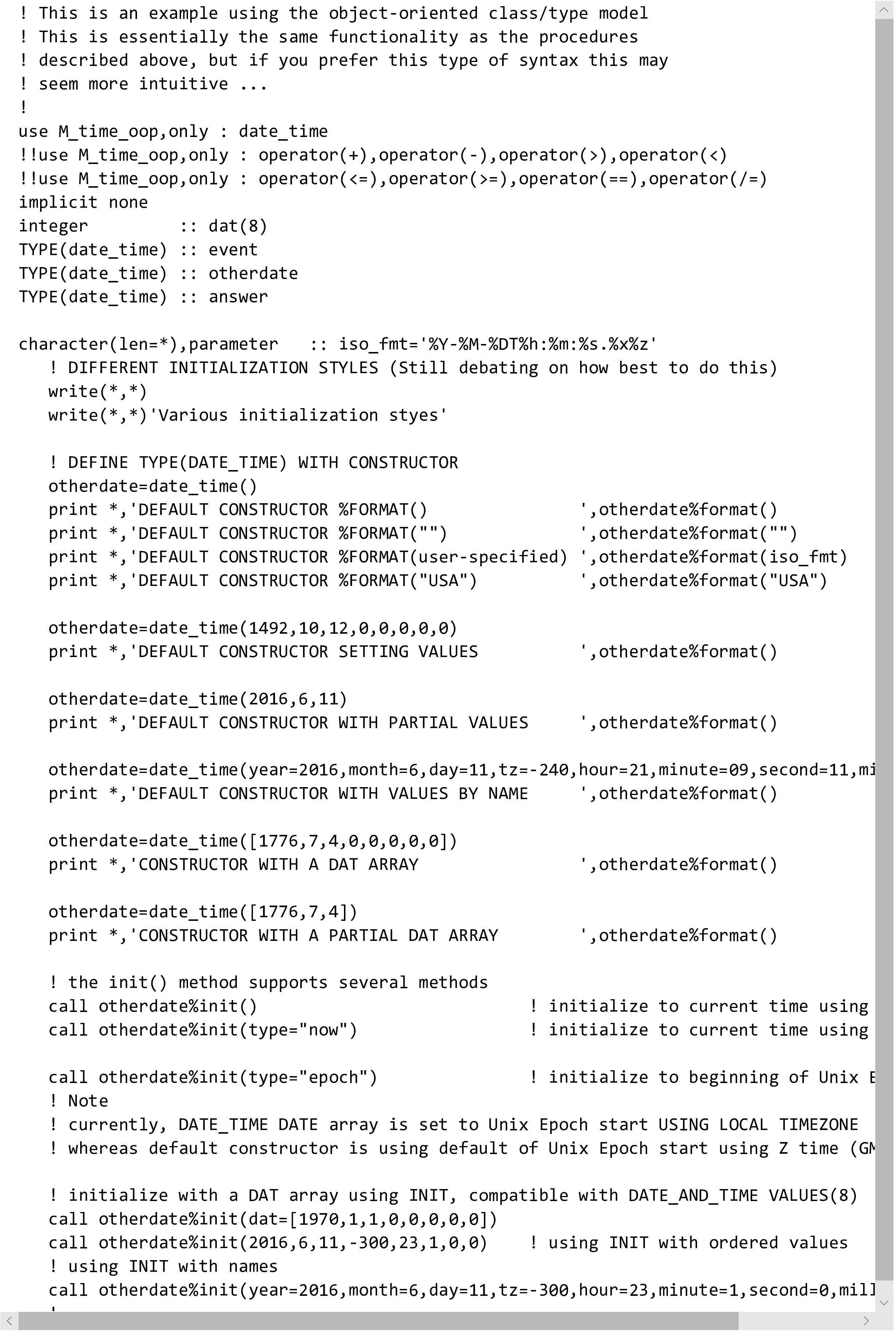
OOPS INTERFACE

If you prefer an Object-oriented interface the M_time_oop module (included with the M_time module source) provides an OOP interface to the M_time module; as described in the subroutine OBJECT_ORIENTED() in the example section.

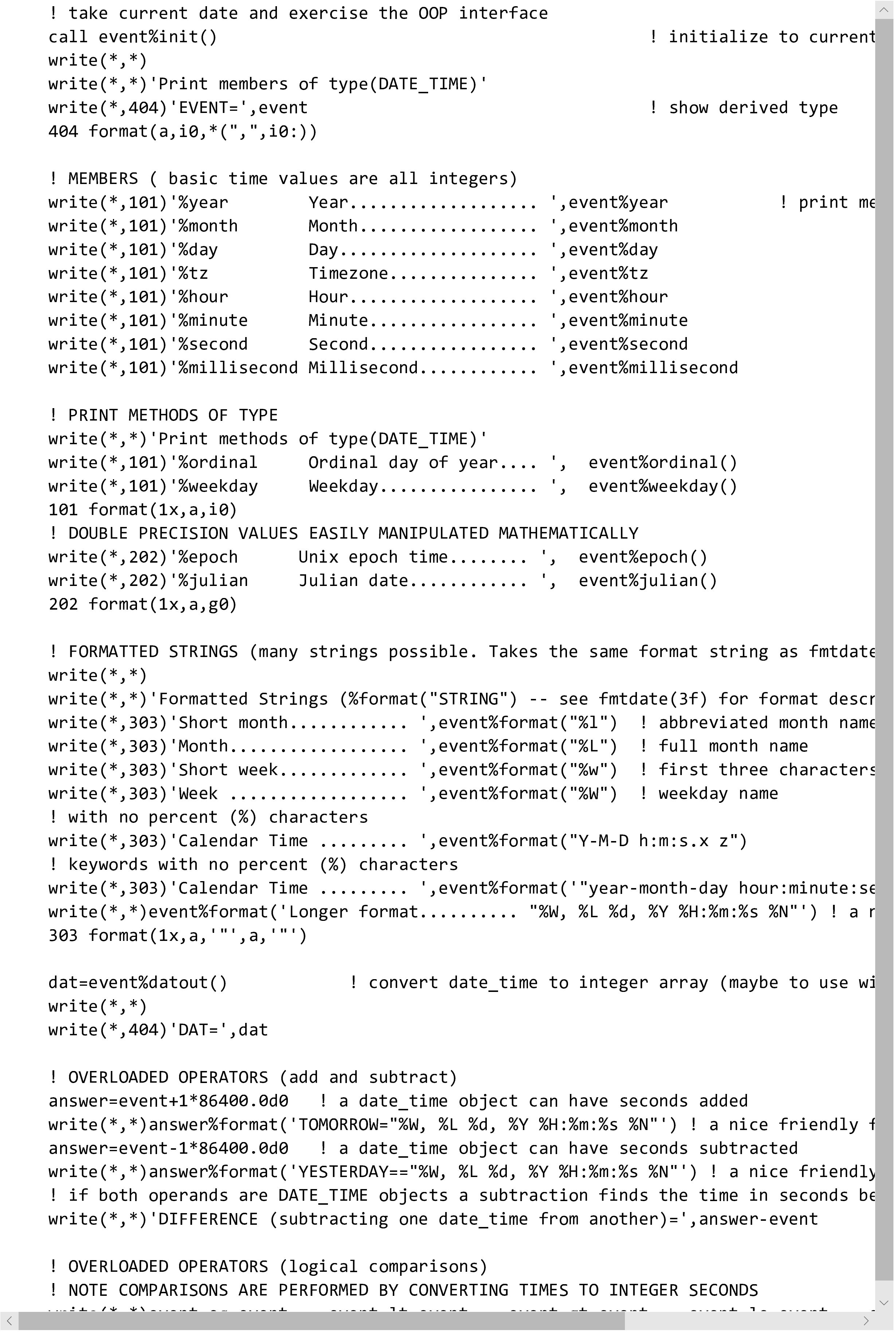
EXAMPLES

The following example program demonstrates the extensive options available for formatting a date as well as how to use the module to calculate dates such as "Yesterday" and "Tomorrow", as well as how to use the Object Oriented interface to the conventional procedures found in the M_time(3fm) module.

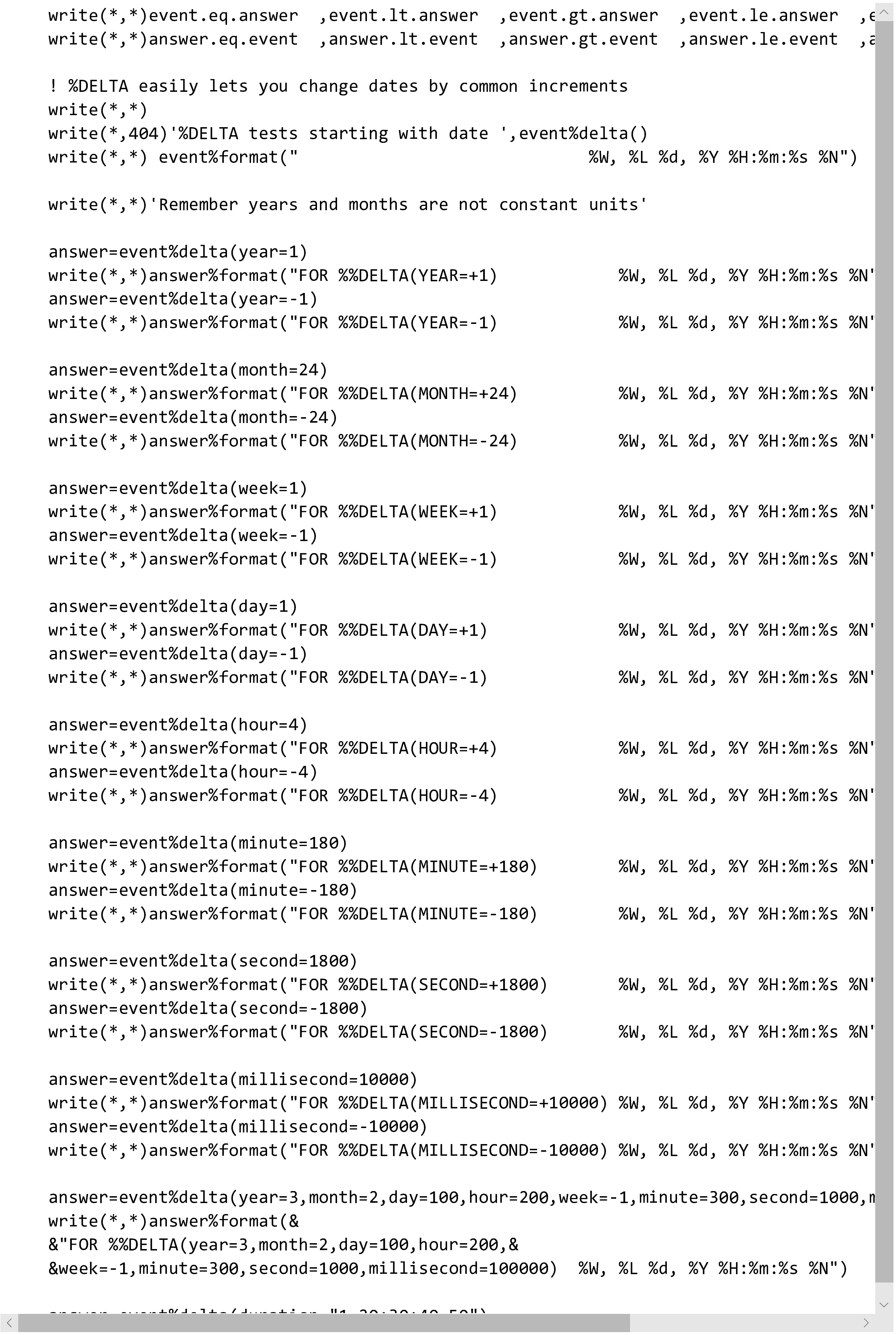
```
program demo_M_time
   call procedural()
   call object_oriented()
!================================================================================
contains
!================================================================================
subroutine procedural()
use M_time, only:  j2d, d2j, u2d, d2u, fmtdate, realtime
integer                      :: dat(8)
real(kind=realtime)          :: julian, unixtime
character(len=*),parameter   :: iso_fmt='%Y-%M-%DT%h:%m:%s.%x%z'
character(len=:),allocatable :: friendly

   friendly='%W, %L %d, %Y %H:%m:%s %N' ! a nice friendly format

   call date_and_time(values=dat)  ! current time is placed in array

   write(*,*)'Today'
   write(*,*)'ISO       ',fmtdate(dat,iso_fmt)
   write(*,*)'Friendly  ',fmtdate(dat,friendly)
   write(*,*)'ISO week  ',fmtdate(dat,'%I')

   julian=d2j(dat)
   unixtime=d2u(dat)

   write(*,*)'Yesterday' ! subtract a day from scalar time and print
   write(*,*)'          ',fmtdate(u2d(unixtime-86400),iso_fmt)
   write(*,*)'          ',fmtdate(j2d(julian-1.0),friendly)
   write(*,*)'          ',fmtdate(j2d(julian-1.0),'%I')

   write(*,*)'Tomorrow'  ! add a day to scalar time and print
   write(*,*)'          ',fmtdate(u2d(unixtime+86400),iso_fmt)
   write(*,*)'          ',fmtdate(j2d(julian+1.0),friendly)
   write(*,*)'          ',fmtdate(j2d(julian+1.0),'%I')

   write(*,*)'Next Week'  ! add a week to scalar time and print
   write(*,*)'          ',fmtdate(u2d(unixtime+7*86400),iso_fmt)
   write(*,*)'          ',fmtdate(j2d(julian+7.0),friendly)
   write(*,*)'          ',fmtdate(j2d(julian+7.0),'%I')

end subroutine procedural
!=====================================================================================
subroutine object_oriented()
!
```



```
! This is an example using the object-oriented class/type model
! This is essentially the same functionality as the procedures
! described above, but if you prefer this type of syntax this may
! seem more intuitive ...
!
use M_time_oop,only : date_time
!!use M_time_oop,only : operator(+),operator(-),operator(>),operator(<)
!!use M_time_oop,only : operator(<=),operator(>=),operator(==),operator(/=)
implicit none
integer         :: dat(8)
TYPE(date_time) :: event
TYPE(date_time) :: otherdate
TYPE(date_time) :: answer

character(len=*),parameter   :: iso_fmt='%Y-%M-%DT%h:%m:%s.%x%z'
   ! DIFFERENT INITIALIZATION STYLES (Still debating on how best to do this)
   write(*,*)
   write(*,*)'Various initialization styes'

   ! DEFINE TYPE(DATE_TIME) WITH CONSTRUCTOR
   otherdate=date_time()
   print *,'DEFAULT CONSTRUCTOR %FORMAT()               ',otherdate%format()
   print *,'DEFAULT CONSTRUCTOR %FORMAT("")             ',otherdate%format("")
   print *,'DEFAULT CONSTRUCTOR %FORMAT(user-specified) ',otherdate%format(iso_fmt)
   print *,'DEFAULT CONSTRUCTOR %FORMAT("USA")          ',otherdate%format("USA")

   otherdate=date_time(1492,10,12,0,0,0,0,0)
   print *,'DEFAULT CONSTRUCTOR SETTING VALUES          ',otherdate%format()

   otherdate=date_time(2016,6,11)
   print *,'DEFAULT CONSTRUCTOR WITH PARTIAL VALUES     ',otherdate%format()

   otherdate=date_time(year=2016,month=6,day=11,tz=-240,hour=21,minute=09,second=11,mi
   print *,'DEFAULT CONSTRUCTOR WITH VALUES BY NAME     ',otherdate%format()

   otherdate=date_time([1776,7,4,0,0,0,0,0])
   print *,'CONSTRUCTOR WITH A DAT ARRAY                ',otherdate%format()

   otherdate=date_time([1776,7,4])
   print *,'CONSTRUCTOR WITH A PARTIAL DAT ARRAY        ',otherdate%format()

   ! the init() method supports several methods
   call otherdate%init()                           ! initialize to current time using
   call otherdate%init(type="now")                 ! initialize to current time using

   call otherdate%init(type="epoch")               ! initialize to beginning of Unix E
   ! Note
   ! currently, DATE_TIME DATE array is set to Unix Epoch start USING LOCAL TIMEZONE
   ! whereas default constructor is using default of Unix Epoch start using Z time (GM

   ! initialize with a DAT array using INIT, compatible with DATE_AND_TIME VALUES(8)
   call otherdate%init(dat=[1970,1,1,0,0,0,0,0])
   call otherdate%init(2016,6,11,-300,23,1,0,0)    ! using INIT with ordered values
   ! using INIT with names
   call otherdate%init(year=2016,month=6,day=11,tz=-300,hour=23,minute=1,second=0,mill
   !
```



```
! take current date and exercise the OOP interface
call event%init()                                           ! initialize to current
write(*,*)
write(*,*)'Print members of type(DATE_TIME)'
write(*,404)'EVENT=',event                                  ! show derived type
404 format(a,i0,*(",",i0:))

! MEMBERS ( basic time values are all integers)
write(*,101)'%year        Year................... ',event%year           ! print me
write(*,101)'%month       Month.................. ',event%month
write(*,101)'%day         Day.................... ',event%day
write(*,101)'%tz          Timezone............... ',event%tz
write(*,101)'%hour        Hour................... ',event%hour
write(*,101)'%minute      Minute................. ',event%minute
write(*,101)'%second      Second................. ',event%second
write(*,101)'%millisecond Millisecond............ ',event%millisecond

! PRINT METHODS OF TYPE
write(*,*)'Print methods of type(DATE_TIME)'
write(*,101)'%ordinal     Ordinal day of year.... ',  event%ordinal()
write(*,101)'%weekday     Weekday................ ',  event%weekday()
101 format(1x,a,i0)
! DOUBLE PRECISION VALUES EASILY MANIPULATED MATHEMATICALLY
write(*,202)'%epoch      Unix epoch time........ ',  event%epoch()
write(*,202)'%julian     Julian date............ ',  event%julian()
202 format(1x,a,g0)

! FORMATTED STRINGS (many strings possible. Takes the same format string as fmtdate
write(*,*)
write(*,*)'Formatted Strings (%format("STRING") -- see fmtdate(3f) for format descr
write(*,303)'Short month............ ',event%format("%l")  ! abbreviated month name
write(*,303)'Month.................. ',event%format("%L")  ! full month name
write(*,303)'Short week............. ',event%format("%w")  ! first three characters
write(*,303)'Week .................. ',event%format("%W")  ! weekday name
! with no percent (%) characters
write(*,303)'Calendar Time ......... ',event%format("Y-M-D h:m:s.x z")
! keywords with no percent (%) characters
write(*,303)'Calendar Time ......... ',event%format('"year-month-day hour:minute:se
write(*,*)event%format('Longer format.......... "%W, %L %d, %Y %H:%m:%s %N"') ! a n
303 format(1x,a,'"',a,'"')

dat=event%datout()            ! convert date_time to integer array (maybe to use wi
write(*,*)
write(*,404)'DAT=',dat

! OVERLOADED OPERATORS (add and subtract)
answer=event+1*86400.0d0   ! a date_time object can have seconds added
write(*,*)answer%format('TOMORROW="%W, %L %d, %Y %H:%m:%s %N"') ! a nice friendly f
answer=event-1*86400.0d0   ! a date_time object can have seconds subtracted
write(*,*)answer%format('YESTERDAY=="%W, %L %d, %Y %H:%m:%s %N"') ! a nice friendly
! if both operands are DATE_TIME objects a subtraction finds the time in seconds be
write(*,*)'DIFFERENCE (subtracting one date_time from another)=',answer-event

! OVERLOADED OPERATORS (logical comparisons)
! NOTE COMPARISONS ARE PERFORMED BY CONVERTING TIMES TO INTEGER SECONDS
write(*,*)event.eq.event, event.lt.event, event.gt.event, event.le.event, e
```



```
write(*,*)event.eq.answer  ,event.lt.answer  ,event.gt.answer  ,event.le.answer  ,e
write(*,*)answer.eq.event  ,answer.lt.event  ,answer.gt.event  ,answer.le.event  ,a

! %DELTA easily lets you change dates by common increments
write(*,*)
write(*,404)'%DELTA tests starting with date ',event%delta()
write(*,*) event%format("                             %W, %L %d, %Y %H:%m:%s %N")

write(*,*)'Remember years and months are not constant units'

answer=event%delta(year=1)
write(*,*)answer%format("FOR %%DELTA(YEAR=+1)            %W, %L %d, %Y %H:%m:%s %N'
answer=event%delta(year=-1)
write(*,*)answer%format("FOR %%DELTA(YEAR=-1)            %W, %L %d, %Y %H:%m:%s %N'

answer=event%delta(month=24)
write(*,*)answer%format("FOR %%DELTA(MONTH=+24)          %W, %L %d, %Y %H:%m:%s %N'
answer=event%delta(month=-24)
write(*,*)answer%format("FOR %%DELTA(MONTH=-24)          %W, %L %d, %Y %H:%m:%s %N'

answer=event%delta(week=1)
write(*,*)answer%format("FOR %%DELTA(WEEK=+1)            %W, %L %d, %Y %H:%m:%s %N'
answer=event%delta(week=-1)
write(*,*)answer%format("FOR %%DELTA(WEEK=-1)            %W, %L %d, %Y %H:%m:%s %N'

answer=event%delta(day=1)
write(*,*)answer%format("FOR %%DELTA(DAY=+1)             %W, %L %d, %Y %H:%m:%s %N'
answer=event%delta(day=-1)
write(*,*)answer%format("FOR %%DELTA(DAY=-1)             %W, %L %d, %Y %H:%m:%s %N'

answer=event%delta(hour=4)
write(*,*)answer%format("FOR %%DELTA(HOUR=+4)            %W, %L %d, %Y %H:%m:%s %N'
answer=event%delta(hour=-4)
write(*,*)answer%format("FOR %%DELTA(HOUR=-4)            %W, %L %d, %Y %H:%m:%s %N'

answer=event%delta(minute=180)
write(*,*)answer%format("FOR %%DELTA(MINUTE=+180)        %W, %L %d, %Y %H:%m:%s %N'
answer=event%delta(minute=-180)
write(*,*)answer%format("FOR %%DELTA(MINUTE=-180)        %W, %L %d, %Y %H:%m:%s %N'

answer=event%delta(second=1800)
write(*,*)answer%format("FOR %%DELTA(SECOND=+1800)       %W, %L %d, %Y %H:%m:%s %N'
answer=event%delta(second=-1800)
write(*,*)answer%format("FOR %%DELTA(SECOND=-1800)       %W, %L %d, %Y %H:%m:%s %N'

answer=event%delta(millisecond=10000)
write(*,*)answer%format("FOR %%DELTA(MILLISECOND=+10000) %W, %L %d, %Y %H:%m:%s %N'
answer=event%delta(millisecond=-10000)
write(*,*)answer%format("FOR %%DELTA(MILLISECOND=-10000) %W, %L %d, %Y %H:%m:%s %N'

answer=event%delta(year=3,month=2,day=100,hour=200,week=-1,minute=300,second=1000,m
write(*,*)answer%format(&
&"FOR %%DELTA(year=3,month=2,day=100,hour=200,&
&week=-1,minute=300,second=1000,millisecond=100000)  %W, %L %d, %Y %H:%m:%s %N")

answer=event%delta(duration="1-20:30:40.50")
```

```
     write(*,*)answer%format("FOR %%DELTA(DURATION='1-20:30:40.50')      %W, %L %d, %Y %H:

  end subroutine object_oriented
  end program demo_M_time
```

Sample output of example program ...

The example from the conventional calls looks like this ...

```
   Today
   ISO       2015-12-22T08:07:34.025-0300
   Friendly  Tuesday, December 22nd, 2015 08:07:34 AM
   ISO week  2015-W52-2
   Yesterday
             2015-12-21T08:07:34.025-0300
             Monday, December 21st, 2015 08:07:34 AM
             2015-W52-1
   Tomorrow
             2015-12-23T08:07:34.025-0300
             Wednesday, December 23rd, 2015 08:07:34 AM
             2015-W52-3
   Next Week
             2015-12-29T08:07:34.025-0300
             Tuesday, December 29th, 2015 08:07:34 AM
             2015-W53-2
```

The example from the object-oriented calls looks like this ...

```
   Various initialization styles
   DEFAULT CONSTRUCTOR %FORMAT()            1970-01-01T00:00:00.000+0000
   DEFAULT CONSTRUCTOR %FORMAT("")          1970-01-01T00:00:00.000+0000
   DEFAULT CONSTRUCTOR %FORMAT(user-specified) 1970-01-01T00:00:00.000+0000
   DEFAULT CONSTRUCTOR %FORMAT("USA")       Thursday, January 1st, 1970 12:00:00 AM
   DEFAULT CONSTRUCTOR SETTING VALUES       1492-10-12T00:00:00.000+0000
   DEFAULT CONSTRUCTOR WITH PARTIAL VALUES  2016-06-11T00:00:00.000+0000
   DEFAULT CONSTRUCTOR WITH VALUES BY NAME  2016-06-11T21:09:11.500-0240
   CONSTRUCTOR WITH A DAT ARRAY             1776-07-04T00:00:00.000+0000
   CONSTRUCTOR WITH A PARTIAL DAT ARRAY     1776-07-04T20:00:00.000-0240

   Print members of type(DATE_TIME)
  EVENT=2016,6,14,-240,22,22,31,253
   Year................... 2016
   Month.................. 6
   Day.................... 14
   Timezone............... -240
   Hour................... 22
   Minute................. 22
   Second................. 31
   Millisecond............ 253
   Print methods of type(DATE_TIME)
```

```
 Ordinal day of year.... 166
 Weekday................ 3
 Unix epoch time........ 1465957351.2529941
 Julian date............ 2457554.5989728356

 Formatted Strings
 Short month............ "Jun"
 Month.................. "June"
 Short week............. "Tue"
 Week .................. "Tuesday"
 Longer format.......... "Tuesday, June 14th, 2016 10:22:31 PM"

DAT=2016,6,14,-240,22,22,31,253
 TOMORROW="Wednesday, June 15th, 2016 10:22:31 PM"
 YESTERDAY=="Wednesday, June 13th, 2016 10:22:31 PM"
 DIFFERENCE (subtracting one date_time from another)=   86400.000000000000
 T F F T T F
 F T F T F T
 F F T F T T

%DELTA tests starting with date 2016,6,14,-240,22,22,31,253
                              Tuesday, June 14th, 2016 10:22:31 PM
 Remember years and months are not constant units
 FOR DELTA YEAR=+1            Wednesday, June 14th, 2017 10:22:31 PM
 FOR DELTA YEAR=-1            Sunday, June 14th, 2015 10:22:31 PM
 FOR DELTA MONTH=+24          Saturday, June 16th, 2018 10:22:31 PM
 FOR DELTA MONTH=-24          Saturday, June 14th, 2014 10:22:31 PM
 FOR DELTA WEEK=+1            Tuesday, June 21st, 2016 10:22:31 PM
 FOR DELTA WEEK=-1            Tuesday, June 7th, 2016 10:22:31 PM
 FOR DELTA DAY=+1             Wednesday, June 15th, 2016 10:22:31 PM
 FOR DELTA DAY=+1             Monday, June 13th, 2016 10:22:31 PM
 FOR DELTA HOUR=+4            Wednesday, June 15th, 2016 02:22:31 AM
 FOR DELTA HOUR=-4            Tuesday, June 14th, 2016 06:22:31 PM
 FOR DELTA MINUTE=+180        Wednesday, June 15th, 2016 01:22:31 AM
 FOR DELTA MINUTE=-180        Tuesday, June 14th, 2016 07:22:31 PM
 FOR DELTA SECOND=+1800       Tuesday, June 14th, 2016 10:52:31 PM
 FOR DELTA SECOND=-1800       Tuesday, June 14th, 2016 09:52:31 PM
 FOR DELTA MILLISECOND=+10000 Tuesday, June 14th, 2016 10:22:41 PM
 FOR DELTA MILLISECOND=-10000 Tuesday, June 14th, 2016 10:22:21 PM
 FOR DELTA ONE-OF-EACH        Sunday, November 24th, 2019 11:39:01 AM
```

DEFINITIONS

A "date_and_time" array **"DAT"** has the same format as the array of values generated by the Fortran intrinsic DATE_AND_TIME(3f). That is, it is an 8-element integer array containing year, month, day, Time zone difference from UTC in minutes, hour, minutes, seconds, and milliseconds of the second. This array represents a date on the Proleptic Gregorian Calendar.

The **Proleptic Gregorian Calendar** assumes the Gregorian Calendar existed back to the beginning of the Julian Day calendar (4713 BC). This means historic dates will often be confused, as the Julian Calendar was used in the USA until 1752-09-03, for example. The Gregorian Calendar was formally decreed on 1582-10-15 but was not adapted in many countries. The Julian Calendar was first used around 45 BC. Note that the Proleptic Gregorian Calendar includes a year zero (0). It is frequently used in computer software to simplify the handling of older dates. For example, it is the calendar used by MySQL, SQLite, PHP, CIM, Delphi, Python and COBOL. The Proleptic Gregorian Calendar is explicitly required for all dates before 1582 by ISO 8601:2004 (clause 4.3.2.1 The Gregorian calendar) if the partners to information exchange agree.

Unix Epoch Time (UET) is defined as the number of seconds since 00:00:00 on January 1st. 1970, UTC.

A **JED** is defined as a **Julian Ephemeris Date**. JED days start at noon (not at midnight). 4713-01-01 BC at noon is defined as JED 0.0.

If you are not familiar with them, in this context Julian Dates and Unix Epoch Times are scalar numbers that allow for easy computations using dates (to go back one day just subtract one from a Julian Date, for example). Since these values are generally not considered intelligible, routines are included to convert between these scalar values and the date array so human-readable results can be obtained.

Coordinated Universal Time (French: Temps universel coordonn'e), abbreviated as **UTC**, is the primary time standard by which the world regulates clocks and time. It is within about 1 second of mean solar time at 0o longitude;[1] it does not observe daylight saving time. It is one of several closely related successors to Greenwich Mean Time (GMT). For most purposes, UTC is considered interchangeable with GMT, but GMT is no longer precisely defined by the scientific community.

LIMITATIONS

Like most collections of date and time procedures M_time is *not* a high-precision library that accounts internally for leap seconds and relativistic effects.

M_time(3f) is intended for use in the recent era and is not appropriate for use with historical dates that used some other calendar scheme such as the Julian Calendar. That is, you have to remember to account for conversions to other calendar systems when using historical dates.

When Daylight Savings is in effect calculations will generally be correct, as the date model includes a timezone value; but you are responsible for ensuring dates you create use the correct timezone value or otherwise account for Daylight Savings Time as needed.

Currently, dates are manipulated using the current system timezone, which can typically be set using the environment variable TZ. So if you desire to set the default timezone you generally set the environment variable *before* executing your program. This is compatible with current observed behavior for the intrinsic procedure DATE_AND_TIME(3f) with compilers I have tested with, but does not seem to be a specified behavior as far as the standard is concerned. That is, DATE_AND_TIME(3f) returns a vector that contains a current time zone, but does not specify how a current time zone can be explicitly set. Since this library is intentionally designed to complement DATE_AND_TIME(3f) it adopts the same behavior. A routine to let you set a default time zone could be added in the future.

Note the environment variable can be set using put_environment_variable(3f) from the libGPF library:

```
use M_system, only : put_environment_variable
call put_environment_variable('TZ','America/New_York',ierr)
```

There is no warranty on this code, and it is certain to change.

SEE ALSO

The ISO-8601 standard is often used for business-related transactions.

There are (of course) the C/C++ intrinsics which provide much of the same functionality that should be bindable to Fortran via the ISO_C_BINDING module.

If you are looking for a high-precision Fortran library that is well tested for manipulating dates I would suggest looking at the NASA SPICElib library. If you care about Leap Seconds, Orbital Mechanics, GPS/Satellite communications, and Astronomy it is worth a look.

The Fortran Wiki fortranwiki.org contains information on other libraries and modules that provide date-time procedures.

NAME

box_month(3f) - [M_time] create specified month in a character array **(LICENSE:PD)**

SYNOPSIS

```
subroutine box_month(dat,calen)

    integer,intent(in)     :: dat(8)
    character(len=21)      :: calen(8)
```

DESCRIPTION

box_month(3f) uses a year and month from a date array to populate a small character array with a calendar representing the month.

OPTIONS

dat "DAT" array (an integer array of the same format as the array returned by the intrinsic **DATE_AND_TIME**(3f)) describing the date to be used to specify what calendar month to produce.

```
dat=[year,month,day,timezone,hour,minutes,seconds,milliseconds]
```

RETURNS

calen returned character array holding a display of the specified month

EXAMPLE

Sample program:

```
program demo_box_month
use M_time, only : box_month
implicit none
integer           :: dat(8)
character(len=21) :: calendar(8)
   call date_and_time(values=dat)
   call box_month(dat,calendar)
   write(*,'(a)')calendar
end program demo_box_month
```

results:

```
 >     July 2016
 >Mo Tu We Th Fr Sa Su
 >              1  2  3
 > 4  5  6  7  8  9 10
 >11 12 13 14 15 16 17
 >18 19 20 21 22 23 24
 >25 26 27 28 29 30 31
```

AUTHOR

John S. Urban, 2015

LICENSE

Public Domain

NAME

d2j(3f) - [M_time] given DAT date-time array returns Julian Date **(LICENSE:PD)**

SYNOPSIS

```
function d2j(dat) result (julian)

    integer,intent(in)   :: dat(8)
    real(kind=realtime) :: julian
```

DESCRIPTION

OPTIONS

dat Integer array holding a "DAT" array, similar in structure to the array returned by the intrinsic **DATE_AND_TIME**(3f). If not present, use current time.

dat=[year,month,day,timezone,hour,minutes,seconds,milliseconds]

RETURNS

julian The Julian Date.

EXAMPLE

Sample program:

```
program demo_d2j
use M_time, only : d2j
implicit none
integer :: dat(8)
   call date_and_time(values=dat)
   write(*,'(" Today is:",*(i0:,":"))')dat
   write(*,*)'Julian Date is ',d2j(dat)
end program demo_d2j
```

results:

```
Today is:2016:7:19:-240:2:11:50:885
Julian Date is    2457588.7582278359
```

AUTHOR

John S. Urban, 2015

LICENSE

Public Domain

NAME

d2o(3f) - [M_time] converts DAT date-time array to Ordinal day **(LICENSE:PD)**

SYNOPSIS

```
function d2o(dat) result (ordinal)

    integer,intent(in),optional :: dat(8)    ! date time array
    integer                     :: ordinal   ! the returned day of the year
```

DESCRIPTION

Given a date in the form of a "DAT" array return the Ordinal Day, (ie. "the day of the year").

OPTIONS

dat Integer array holding a "DAT" array, similar in structure to the array returned by the intrinsic **DATE_AND_TIME**(3f).

```
dat=[year,month,day,timezone,hour,minutes,seconds,milliseconds]
```

RETURNS

ordinal The day of the year calculated for the given input date, where Jan 1st=1.

EXAMPLE

Sample program:

```
program demo_d2o
use M_time, only : d2o
implicit none
integer :: dat(8)
   call date_and_time(values=dat)
   write(*,'(" Today is:",*(i0:,":"))')dat
   write(*,*)'Day of year is:',d2o(dat)

   ! year, month, day, timezone, hour, minute, seconds, milliseconds
   dat=[2020,12,31,-240,12,0,0,0]
   write(*,*)dat(1),' Days in year is:',d2o(dat)

   dat=[2021,12,31,-240,12,0,0,0]
   write(*,*)dat(1),' Days in year is:',d2o(dat)

   dat=[2022,12,31,-240,12,0,0,0]
   write(*,*)dat(1),' Days in year is:',d2o(dat)

   dat=[2023,12,31,-240,12,0,0,0]
   write(*,*)dat(1),' Days in year is:',d2o(dat)

   dat=[2024,12,31,-240,12,0,0,0]
   write(*,*)dat(1),' Days in year is:',d2o(dat)

end program demo_d2o
```

results:

```
Today is:2016:7:19:-240:20:1:19:829
Day of year is:         201
       2020  Days in year is:         366
       2021  Days in year is:         365
       2022  Days in year is:         365
       2023  Days in year is:         365
       2024  Days in year is:         366
```

AUTHOR

John S. Urban, 2015

LICENSE

Public Domain

NAME

d2u(3f) - [M_time] given DAT date-time array returns Unix Epoch Time (UET starts at 0000 on 1 Jan. 1970, UTC) **(LICENSE:PD)**

SYNOPSIS

```
function d2u(dat) result (unixtime)

    integer,intent(in),optional :: dat(8)
    real(kind=realtime)         :: unixtime
```

DESCRIPTION

Converts a DAT date-time array to a Unix Epoch Time value. Typically mathematical operations such as sums, sorting and comparison are performed with simple UET numeric values, and then they are converted back.

OPTIONS

dat Integer array holding a "DAT" array, similar in structure to the array returned by the intrinsic **DATE_AND_TIME**(3f). If not present the current time is used

```
dat=[year,month,day,timezone,hour,minutes,seconds,milliseconds]
```

RETURNS

unixtime

The "Unix Epoch" time, or the number of seconds since 00:00:00 on January 1st, 1970, UTC.

EXAMPLE

Sample program:

```
program demo_d2u
use M_time, only : d2u
implicit none
integer           :: dat(8)
   call date_and_time(values=dat)
   write(*,'(" Today is:",*(i0:,":"))')dat
   write(*,*)'Unix Epoch time is ',d2u(dat)
end program demo_d2u
```

results:

```
Today is:2016:7:19:-240:2:0:48:561
Unix Epoch time is    1468908048.5610321
```

AUTHOR

John S. Urban, 2015

LICENSE

Public Domain

NAME

d2w(3f) - [M_time] calculate iso-8601 Week-numbering year date yyyy-Www-d given DAT date-time array **(LICENSE:PD)**

SYNOPSIS

```
subroutine d2w(dat,iso_year,iso_week,iso_weekday,iso_name)

    integer,intent(in)              :: dat(8)     ! input date array
    integer,intent(out)             :: iso_year, iso_week, iso_weekday
    character(len=10),intent(out)   :: iso_name
```

DESCRIPTION

Given a "DAT" array defining a date and time, return the ISO-8601 Week in two formats -- as three integer values defining the ISO year, week of year and weekday; and as a string of the form "yyyy-Www-d".

OPTIONS

dat "DAT" array (an integer array of the same format as the array returned by the intrinsic **DATE_AND_TIME**(3f)) describing the date, which is the basic time description used by the other **M_time**(3fm) module procedures.

RETURNS

iso_year
ISO-8601 year number for the given date

iso_week
ISO-8601 week number for the given date

iso_weekday
ISO-8601 weekday number for the given date

iso_name
ISO-8601 Week string for the data in the form "yyyy-Www-d".

EXAMPLE

Sample program:

```
program demo_d2w
use M_time, only : d2w
implicit none
integer           :: dat(8)     ! input date array
integer           :: iso_year, iso_week, iso_weekday
character(len=10) :: iso_name

   call date_and_time(values=dat)
   call d2w(dat,iso_year,iso_week,iso_weekday,iso_name)
   write(*,'("ISO-8601 Week:   ",a)')iso_name
   write(*,'(a,i0)')'ISO-8601 year    ',iso_year
   write(*,'(a,i0)')'ISO-8601 week    ',iso_week
   write(*,'(a,i0)')'ISO-8601 weekday ',iso_weekday
```

end program demo_d2w

results:

```
ISO-8601 Week:   2016-W29-1
ISO-8601 year    2016
ISO-8601 week    29
ISO-8601 weekday 1
```

DEFINITION

The ISO-8601 date and time standard was issued by the International Organization for Standardization (ISO). It is used (mainly) in government and business for fiscal years, as well as in timekeeping. The system specifies a week year atop the Gregorian calendar by defining a notation for ordinal weeks of the year.

An ISO week-numbering year (also called ISO year informally) has 52 or 53 full weeks. That is 364 or 371 days instead of the usual 365 or 366 days. The extra week is referred to here as a leap week, although ISO-8601 does not use this term. Weeks start with Monday. The first week of a year is the week that contains the first Thursday of the year (and, hence, always contains 4 January). ISO week year numbering therefore slightly deviates from the Gregorian for some days close to January 1st.

CALCULATION

The ISO-8601 week number of any date can be calculated, given its ordinal date (i.e. position within the year) and its day of the week.

METHOD

Using ISO weekday numbers (running from 1 for Monday to 7 for Sunday), subtract the weekday from the ordinal date, then add 10. Divide the result by 7. Ignore the remainder; the quotient equals the week number. If the week number thus obtained equals 0, it means that the given date belongs to the preceding (week-based) year. If a week number of 53 is obtained, one must check that the date is not actually in week 1 of the following year.

These two statements are assumed true when correcting the dates around January 1st:

- The number of weeks in a given year is equal to the corresponding week number of 28 December.
- January 4th is always in the first week.

ISO_NAME

Week date representations are in the format YYYYWww-D.

- [YYYY] indicates the ISO week-numbering year which is slightly different from the traditional Gregorian calendar year.
- [Www] is the week number prefixed by the letter W, from W01 through W53.
- [D] is the weekday number, from 1 through 7, beginning with Monday and ending with Sunday.

For example, the Gregorian date 31 December 2006 corresponds to the Sunday of the 52nd week of 2006, and is written

```
2006-W52-7 (extended form)
or
2006W527 (compact form).
```

REFERENCE

From Wikipedia, the free encyclopedia 2015-12-19

AUTHOR

John S. Urban, 2015-12-19

LICENSE

Public Domain

NAME

date_to_julian(3f) - [M_time] converts DAT date-time array to Julian Date **(LICENSE:PD)**

SYNOPSIS

```
subroutine date_to_julian(dat,juliandate,ierr)

    integer,intent(in)                 :: dat(8)
    real(kind=realtime),intent(out)    :: juliandate
    integer,intent(out)                :: ierr
```

DESCRIPTION

Converts a DAT date-time array to a Unix Epoch Time (UET) value. UET is the number of seconds since 00:00 on January 1st, 1970, UTC.

OPTIONS

dat Integer array holding a "DAT" array, similar in structure to the array returned by the intrinsic **DATE_AND_TIME**(3f).

```
dat=[year,month,day,timezone,hour,minutes,seconds,milliseconds]
```

RETURNS

juliandate
A Julian Ephemeris Date (JED) is the number of days since noon (not midnight) on January 1st, 4713 BC.

ierr Error code. If 0 no error occurred.

EXAMPLE

Sample Program:

```
program demo_date_to_julian
use M_time, only : date_to_julian,realtime
implicit none
integer             :: dat(8)
real(kind=realtime) :: juliandate
integer             :: ierr
   ! generate DAT array
   call date_and_time(values=dat)
   ! show DAT array
   write(*,'(" Today is:",*(i0:,":"))')dat
   ! convert DAT to Julian Date
   call date_to_julian(dat,juliandate,ierr)
   write(*,*)'Julian Date is ',juliandate
   write(*,*)'ierr is ',ierr
end program demo_date_to_julian
```

results:

```
Today is:2016:7:19:-240:11:3:13:821
Julian Date is    2457589.1272432986
ierr is            0
```

AUTHOR

John S. Urban, 2015

LICENSE

Public Domain

NAME

date_to_unix(3f) - [M_time] converts DAT date-time array to Unix Epoch Time **(LICENSE:PD)**

SYNOPSIS

```
subroutine date_to_unix(dat,unixtime,ierr)

    integer,intent(in)                 :: dat(8)
    real(kind=realtime),intent(out)    :: unixtime
    integer,intent(out)                :: ierr
```

DESCRIPTION

Converts a DAT date-time array to a UET (Unix Epoch Time).

OPTIONS

dat Integer array holding a "DAT" array, similar in structure to the array returned by the intrinsic **DATE_AND_TIME**(3f).

```
dat=[year,month,day,timezone,hour,minutes,seconds,milliseconds]
```

RETURNS

unixtime
The "Unix Epoch" time, or the number of seconds since 00:00:00 on January 1st, 1970, UTC.

ierr Error code. If 0 no error occurred.

EXAMPLE

Sample program:

```
program demo_date_to_unix
use M_time, only : date_to_unix, realtime
implicit none
integer             :: dat(8)
real(kind=realtime) :: unixtime
integer             :: ierr
   call date_and_time(values=dat)
   write(*,'(" Today is:",*(i0:,":"))')dat
   call date_to_unix(dat,unixtime,ierr)
   write(*,*)'Unix Epoch time is ',unixtime
   write(*,*)'ierr is ',ierr
end program demo_date_to_unix
```

results:

```
Today is:2016:7:18:-240:23:44:20:434
Unix Epoch time is    1468899860.4340105
ierr is           0
```

AUTHOR

John S. Urban, 2015

LICENSE

Public Domain

NAME

days2sec(3f) - [M_time] convert string of form [[-]dd-]hh:mm:ss.nn to seconds **(LICENSE:PD)**

SYNOPSIS

```
function days2sec(str) result(time)

    character(len=*),intent(in)       :: str
    real(kind=realtime)               :: time
```

DESCRIPTION

Given a string representing a duration of the form

[-][[[dd-]hh:]mm:]ss

or NNdNNhNNmNNsNNw return a value representing seconds

If "dd-" is present, units for the numbers are assumed to proceed from day to hour to minute to second. But if no day is present, the units are assumed to proceed from second to minutes to hour from left to right. That is ...

```
[-]dd-hh:mm:ss
[-]dd-hh:mm
[-]dd-hh

hh:mm:ss
mm:ss
ss

 Where dd is days, hh hours, mm minutes and ss seconds.

 A decimal fraction is supported on the seconds (Actually,
 any of the numeric values may represent positive floating
 point numbers). Spaces are ignored.
```

NNdNNhNNmNNs Simple numeric values may also be used with unit suffixes; where s,m,h, or d represents seconds, minutes, hours or days and w represents a week. Allowed aliases for w,d,h,m, and s units are

```
d -  days,day
m -  minutes,minute,min,mins
h -  hours,hour,hr,hrs
s -  seconds,second,sec,secs
w -  week, weeks, wk, wks
```

The numeric values may represent floating point numbers.

Spaces, commas and case are ignored.

OPTIONS

str string of the general form dd-hh:mm:ss.nn

RETURNS

time the number of seconds represented by the input string

EXAMPLE

Sample program:

```
program demo_days2sec
use M_time, only : days2sec
implicit none
   write(*,*)days2sec('1-12:04:20')
   write(*,*)'one second ',days2sec('1')
```

```
      write(*,*)'one minute ',days2sec('1:00')
      write(*,*)'one hour ',days2sec('1:00:00')
      write(*,*)'one day ',days2sec('1-00:00:00')
      write(*,*)nint(days2sec(' 1-12:04:20              ')) .eq. 129860
      write(*,*)nint(days2sec(' 1.5 days                ')) .eq. 129600
      write(*,*)nint(days2sec(' 1.5 days 4hrs 30minutes ')) .eq. 145800
      write(*,*)nint(days2sec(' 1.5d                    ')) .eq. 129600
      write(*,*)nint(days2sec(' 1d2h3m4s                ')) .eq. 93784
      ! duplicates
      write(*,*)nint(days2sec(' 1d1d1d                  ')) .eq. 259200
      ! negative values
      write(*,*)nint(days2sec(' 4d-12h                  ')) .eq. 302400
   end program demo_days2sec
```

Results:

```
   129860.00000000000
   one second    1.0000000000000000
   one minute    60.000000000000000
   one hour    3600.0000000000000
   one day    86400.000000000000
```

T
T
T
T
T
T
T

AUTHOR

John S. Urban, 2015

LICENSE

Public Domain

NAME

dow(3f) - [M_time] given a date-time array DAT return the day of the week **(LICENSE:PD)**

SYNOPSIS

```
subroutine dow(values, weekday, day, ierr)

    integer,intent(in) :: values(8)
    integer,intent(out),optional :: weekday
    character(len=*),intent(out),optional :: day
    integer,intent(out),optional :: ierr
```

DESCRIPTION

Given a date array DAT return the *day* of the week as a number and a name, Mon=1.

OPTIONS

values "DAT" array (an integer array of the same format as the array returned by the intrinsic **DATE_AND_TIME**(3f)) describing the date to be used to calculate the *day* of the week.

RETURNS

weekday
The numeric *day* of the week, starting with Monday=1. Optional.

day The name of the *day* of the week. Optional.

ierr Error code

- [0] correct
- [-1] invalid input date
- [-2] neither *day* nor *weekday* return *values* were requested.

If the error code is not returned and an error occurs, the program is stopped.

EXAMPLE

Sample program:

```
program demo_dow
use M_time, only : dow
implicit none
integer          :: dat(8)      ! input date array
integer          :: weekday
character(len=9) :: day
integer          :: ierr

  call date_and_time(values=dat)
  call dow(dat, weekday, day, ierr)
  write(*,'(a,i0)')'weekday=',weekday
  write(*,'(a,a)')'day=',trim(day)
  write(*,'(a,i0)')'ierr=',ierr

  end program demo_dow
```

results:

```
weekday=1
day=Monday
ierr=0
```

AUTHOR

John S. Urban, 2015-12-19

LICENSE

Public Domain

NAME

easter(3f) - [M_time] calculate date for Easter given a year **(LICENSE:PD)**

SYNOPSIS

```
subroutine easter(year,dat)

    integer, intent(in)   :: year
    integer, intent(out)  :: dat
```

DESCRIPTION

The Date of Easter (Sunday)

The algorithm is due to **J.-M**. Oudin (1940) and is reprinted in the Explanatory Supplement to the Astronomical Almanac, ed. P. K. Seidelmann (1992). See Chapter 12, "Calendars", by L. E. Doggett.

The following are dates of Easter from 1980 to 2024:

```
1980  April  6        1995  April 16        2010  April  4
1981  April 19        1996  April  7        2011  April 24
1982  April 11        1997  March 30        2012  April  8
1983  April  3        1998  April 12        2013  March 31
1984  April 22        1999  April  4        2014  April 20
1985  April  7        2000  April 23        2015  April  5
1986  March 30        2001  April 15        2016  March 27
1987  April 19        2002  March 31        2017  April 16
1988  April  3        2003  April 20        2018  April  1
1989  March 26        2004  April 11        2019  April 21
1990  April 15        2005  March 27        2020  April 12
1991  March 31        2006  April 16        2021  April  4
1992  April 19        2007  April  8        2022  April 17
1993  April 11        2008  March 23        2023  April  9
1994  April  3        2009  April 12        2024  March 31
```

N.B. The date of Easter for the Eastern Orthodox Church may be different.

OPTIONS

year Year for which to calculate day that Easter falls on

RESULTS

dat Date array for noon on Easter for the specified year

EXAMPLE

Sample program:

```
program demo_easter
use M_time, only : easter, fmtdate
implicit none
integer :: year
integer :: dat(8) ! year,month,day,tz,hour,minute,second,millisecond
  call date_and_time(values=dat)  ! get current year
  year=dat(1)
  call easter(year, dat)
  write(*,*)fmtdate(dat,&
  "Easter day: the %d day of %L in the year of our Lord %Y")
end program demo_easter
```

Sample output:

```
Easter day: the 16th day of April in the year of our Lord 2017
```

NAME

fmtdate(3f) - [M_time] given DAT date-time array return date as string using specified format **(LICENSE:PD)**

SYNOPSIS

```
function fmtdate(values,format) RESULT (timestr)

    integer,dimension(8),intent(in)       :: values
    character(len=*),intent(in),optional :: format
    character(len=:),allocatable         :: timestr
```

DESCRIPTION

The **fmtdate**(3f) procedure lets you reformat a DAT array in many common formats using a special string containing macro names beginning with '%'. To see the allowable macros call or see the **fmtdate_usage**(3f) routine.

OPTIONS

values date in a "DAT" array, which is the same format as the values returned by the intrinsic **DATE_AND_TIME**(3f).

dat=[year,month,day,timezone,hour,minutes,seconds,milliseconds]

format string describing how to format the "DAT" array. For a complete description of the formatting macros supported see **fmtdate_usage**(3f).

RETURNS

timestr formatted output string representing date

EXAMPLE

Sample program:

```
program demo_fmtdate
use M_time, only : fmtdate
implicit none
integer :: dat(8)
   call date_and_time(values=dat)
   write(*,*)fmtdate(dat,"current date: %w, %l %d, %Y %H:%m:%s %N")
   call showme()
contains
subroutine showme()
   use M_time, only : fmtdate_usage
   call fmtdate_usage() ! see all formatting options
end subroutine showme
end program demo_fmtdate
```

results:

```
 The current date is Sun, Jul 17th, 2016 01:21:35 PM
  ::
  :: An up-to-date description of all the
  :: formatting options will appear here
  ::
```

AUTHOR

John S. Urban, 2015-12-19

LICENSE

Public Domain

NAME

fmtdate_usage(3f) - [M_time] display macros recognized by **fmtdate**(3f) and **now**(3f) **(LICENSE:PD)**

SYNOPSIS

```
subroutine fmtdate_usage(indent)

    integer,intent(in),optional      :: indent
```

DESCRIPTION

The **fmtdate_usage**(3f) subroutine displays the formatting options available for use in procedures such as **fmtdate**(3f) and **now**(3f). It is typically used to produce up-to-date help text in commands that use the **M_time**(3fm) module, so that the formatting information only needs maintained in one place (this routine) and is easily displayed so users can quickly obtain a description of the formatting macros.

OPTIONS

indent how many spaces to prefix the output with, so that calling programs can position the output. Default for this optional parameter is three (3).

EXAMPLE

Sample Program:

```
program demo_fmtdate_usage
use M_time, only : fmtdate_usage
implicit none
   call fmtdate_usage() ! see all formatting options
end program demo_fmtdate_usage
```

results (actually call the routine to ensure this is up to date):

```
Description                                        Example

Base time array:
(1) %Y -- year, yyyy                                2016
(2) %M -- month of year, 01 to 12                   07
(3) %D -- day of month, 01 to 31                    29
    %d -- day of month, with suffix (1st, 2nd,...)  29th
(4) %Z -- minutes from UTC                          -0240
    %z -- -+hh:mm from UTC                          -04:00
    %T -- -+hhmm  from UTC                          -0400
(5) %h -- hours, 00 to 23                           10
    %H -- hour (1 to 12, or twelve-hour clock)      10
    %N -- midnight< AM <=noon; noon<= PM <midnight  AM
(6) %m -- minutes, 00 to 59                         54
(7) %s -- sec, 00 to 59                             08
(8) %x -- milliseconds 000 to 999                   521
Conversions:
    %E -- Unix Epoch time                           1469804048.5220029
    %e -- integer value of Unix Epoch time          1469804049
    %J -- Julian  date                              2457599.121
    %j -- integer value of Julian Date(Julian Day)  2457599
    %O -- Ordinal day (day of year)                 211
    %o -- Whole days since Unix Epoch date          17011
    %U -- day of week, 1..7 Sunday=1                6
    %u -- day of week, 1..7 Monday=1                5
    %i -- ISO week of year 1..53                    30
    %I -- iso-8601 week-numbering date(yyyy-Www-d)  2016-W30-5
 Names:
    %l -- abbreviated month name                    Jul
```

```
      %L -- full month name                             July
      %w -- first three characters of weekday           Fri
      %W -- weekday name                                Friday
      %p -- phase of moon                               New
      %P -- percent of way from new to full moon        -1%
   Literals:
      %% -- a literal %                                 %
      %t -- tab character
      %b -- blank character
      %B -- exclamation(bang) character
      %n -- new line (system dependent)
      %q -- single quote (apostrophe)
      %Q -- double quote
   Program timing:
      %c -- CPU_TIME(3f) output                         .21875000000000000
      %C -- number of times this routine is used        1
      %S -- seconds since last use of this format       .0000000000000000
      %k -- time in seconds from SYSTEM_CLOCK(3f)       723258.812
      %K -- time in clicks from SYSTEM_CLOCK(3f)        723258812
```

If no percent (%) is found in the format one of several alternate substitutions occurs.

If the format is composed entirely of one of the following keywords the following substitutions occur:

```
   "iso-8601",
   "iso"         ==> %Y-%M-%DT%h:%m:%s%z
   "iso-8601W",
   "isoweek"     ==> %I 2016-W30-5
   "sql"         ==> "%Y-%M-%D %h:%m:%s.%x"
   "sqlday"      ==> "%Y-%M-%D"
   "sqltime"     ==> "%h:%m:%s.%x"
   "rfc-2822"    ==> %w, %D %l %Y %h:%m:%s %T
   "rfc-3339"    ==> %Y-%M-%DT%h:%m:%s%z
   "date"        ==> %w %l %D %h:%m:%s UTC%z %Y
   "short"       ==> %w, %l %d, %Y %H:%m:%s %N UTC%z
   "long"," "    ==> %W, %L %d, %Y %H:%m:%s %N UTC%z
   "suffix"      ==> %Y%D%M%h%m%s
   "formal"      ==> The %d of %L %Y
   "lord"        ==> the %d day of %L in the year of our Lord %Y
   "easter"      ==> FOR THE YEAR OF THE CURRENT DATE:
                     Easter day: the %d day of %L in the year of our Lord %Y
   "all"         ==> A SAMPLE OF DATE FORMATS
```

otherwise the following words are replaced with the most common macros:

```
    year     %Y  2016
    month    %M  07
    day      %D  29
    hour     %h  10
    minute   %m  54
    second   %s  08
    epoch    %e  1469804049
    julian   %j  2457599
    ordinal  %O  211
    weekday  %u  5
```

if none of these keywords are found then every letter that is a macro is assumed to have an implied percent in front of it. For example:

```
YMDhms ==> %Y%M%D%h%m%s ==> 20160729105408
```

AUTHOR

John S. Urban, 2015-10-24

LICENSE

Public Domain

NAME

guessdate(3f) - [M_time] reads in a date, in various formats **(LICENSE:PD)**

SYNOPSIS

```
subroutine guessdate(anot,dat)

    character(len=*),intent(in) :: anot
    integer,intent(out)         :: dat(8)
```

DESCRIPTION

Read in strings and except for looking for month names remove non-numeric characters and try to convert a string assumed to represent a date to a date-time array.

Years should always be expressed as four-digit numbers, and except for the special format yyyy-mm-dd the day should come after the year. Named months are preferred. If ambiguous the order is assumed to be day - month - year. Times are assumed to be of the form HH:MM:SS

It is planned that this routine will be superseded. As an alternative, a C routine exists in the standard C libraries that allows for expansive features when reading dates that can be called via the ISO_C_BINDING interface.

OPTIONS

anot A string assumed to represent a date including a year, month and day.

dat Integer array holding a "DAT" array, similar in structure to the array returned by the intrinsic **DATE_AND_TIME**(3f).

```
dat=[year,month,day,timezone,hour,minutes,seconds,milliseconds]
```

EXAMPLE

Sample program:

```
program demo_guessdate
use M_time, only : guessdate, fmtdate
implicit none
character(len=20),allocatable :: datestrings(:)
character(len=:),allocatable  :: answer
integer                       :: dat(8)
integer                       :: i
   datestrings=[ &
   & 'January 9th, 2001   ',&
   & ' Tue Jul 19 2016    ',&
   & ' 21/12/2016         ',&
   & ' 4th of Jul 2004    ' ]
   do i=1,size(datestrings)
      write(*,'(a)')repeat('-',80)
      write(*,*)'TRYING ',datestrings(i)
      call guessdate(datestrings(i),dat)
      write(*,*)'DAT ARRAY ',dat
      answer=fmtdate(dat)
      write(*,*)'FOR '//datestrings(i)//' GOT '//trim(answer)
   enddo
end program demo_guessdate
```

results:

```
---------------------------------------------------------------------
TRYING January 9th, 2001
DAT ARRAY         2001  1  9   -240    0   0   0    0
```

```
FOR January 9th, 2001    GOT Tuesday, January 9th, 2001 12:00:00 AM
---------------------------------------------------------------------
TRYING  Tue Jul 19 2016
DAT ARRAY         2016  7  19  -240    0   0   0    0
FOR  Tue Jul 19 2016     GOT Tuesday, July 19th, 2016 12:00:00 AM
---------------------------------------------------------------------
TRYING  21/12/2016
DAT ARRAY         2016  12 21  -240    0   0   0    0
FOR  21/12/2016          GOT Wednesday, December 21st, 2016 12:00:00 AM
---------------------------------------------------------------------
TRYING  4th of Jul 2004
DAT ARRAY         2004  7  4   -240    0   0   0    0
FOR  4th of Jul 2004     GOT Sunday, July 4th, 2004 12:00:00 AM
```

LICENSE

Public Domain

NAME

j2d(3f) - [M_time] given a JED (Julian Ephemeris Date) returns a date-time array DAT. **(LICENSE:PD)**

SYNOPSIS

```
function j2d(julian) result (dat)

    real(kind=realtime),intent(in),optional :: julian
    integer                                 :: dat(8)
```

DESCRIPTION

Converts a Julian Ephemeris Date to a DAT date-time array.

OPTIONS

julian A Julian Ephemeris Date (JED) is the number of days since noon (not midnight) on January 1st, 4713 BC. If not present, use current time.

RETURNS

dat Integer array holding a "DAT" array, similar in structure to the array returned by the intrinsic **DATE_AND_TIME**(3f).

```
dat=[year,month,day,timezone,hour,minutes,seconds,milliseconds]
```

EXAMPLE

Sample program:

```
program demo_j2d
use M_time, only : j2d, d2j, fmtdate, realtime
implicit none
real(kind=realtime) :: today
integer :: dat(8)
   call date_and_time(values=dat) ! get the date using intrinsic
   today=d2j(dat)                  ! convert today to Julian Date
   write(*,*)'Today=',fmtdate(j2d(today))
   ! math is easy with Julian Days and Julian Dates
   write(*,*)'Yesterday=',fmtdate(j2d(today-1.0d0))
   write(*,*)'Tomorrow=',fmtdate(j2d(today+1.0d0))
end program demo_j2d
```

results:

```
Today=Tuesday, July 19th, 2016 08:48:20 AM
Yesterday=Monday, July 18th, 2016 08:48:20 AM
Tomorrow=Wednesday, July 20th, 2016 08:48:20 AM
```

AUTHOR

John S. Urban, 2015

LICENSE

Public Domain

NAME

julian_to_date(3f) - [M_time] converts a **JED**(Julian Ephemeris Date) to a DAT date-time array. **(LICENSE:PD)**

SYNOPSIS

```
subroutine julian_to_date(julian,dat,ierr)

    real(kind=realtime),intent(in) :: julian
    integer,intent(out)            :: dat(8)
    integer,intent(out)            :: ierr
```

DESCRIPTION

Converts a Unix Epoch Time (UET) value to a DAT date-time array. UET is the number of seconds since 00:00 on January 1st, 1970, UTC.

OPTIONS

julian Julian Date (days)

dat Integer array holding a "DAT" array, similar in structure to the array returned by the intrinsic **DATE_AND_TIME**(3f).

ier 0 for successful execution

```
dat=[year,month,day,timezone,hour,minutes,seconds,milliseconds]
```

RETURNS

unixtime
The "Unix Epoch" time, or the number of seconds since 00:00:00 on January 1st, 1970, UTC.

ierr Error code. If 0 no error occurred.

EXAMPLE

Sample program:

```
program demo_julian_to_date
use M_time, only : julian_to_date, fmtdate, realtime
implicit none
real(kind=realtime)     :: juliandate
integer                 :: dat(8)
integer                 :: ierr
   ! set sample Julian Date
   juliandate=2457589.129d0
   ! create DAT array for this date
   call julian_to_date(juliandate,dat,ierr)
   write(*,*)'Sample Date=',fmtdate(dat)
   ! go back one day
   call julian_to_date(juliandate-1.0d0,dat,ierr)
   write(*,*)'Day Before =',fmtdate(dat)
   ! go forward one day
   call julian_to_date(juliandate+1.0d0,dat,ierr)
   write(*,*)'Day After  =',fmtdate(dat)
end program demo_julian_to_date
```

results:

```
Sample Date=Tuesday, July 19th, 2016 11:05:45 AM UTC-04:00
Day Before =Monday, July 18th, 2016 11:05:45 AM UTC-04:00
Day After  =Wednesday, July 20th, 2016 11:05:45 AM UTC-04:00
```

AUTHOR

John S. Urban, 2015

LICENSE

Public Domain

NAME

mo2d(3f) - [M_time] given month name return DAT date-time array for beginning of that month in specified year **(LICENSE:PD)**

SYNOPSIS

```
function mo2d(month_name) result (dat)

        character(len=*),intent(in) :: month_name
        integer                     :: dat(8)
```

DESCRIPTION

Given a Common Calendar month name, return the date as a "DAT" array for the 1st day of the month. An optional year may be specified. The year defaults to the current year.

OPTIONS

month_name
A string representing a Common Calendar month name.

year Optional year. Defaults to current year

RETURNS

dat An integer array that has the same structure as the array returned by the Fortran intrinsic **DATE_AND_TIME**(3f).

EXAMPLE

Sample program:

```
    program demo_mo2d
    use M_time, only : mo2d
    implicit none
       write(*,'(*(i0:,":"))')mo2d('March')
    end program demo_mo2d
```

results:

```
      2016:3:1:-240:0:0:0:0
```

AUTHOR

John S. Urban, 2015

LICENSE

Public Domain

NAME

mo2v(3f) - [M_time] given month name return month number (1-12) of that month **(LICENSE:PD)**

SYNOPSIS

```
function mo2v(month_name) result(imonth)

     character(len=*),intent(in):: month_name ! month name
     integer                     :: imonth     ! month number
```

DESCRIPTION

Given a string representing the name or abbreviation of a Gregorian Calendar month return a number representing the position of the month in the calendar starting with 1 for January and ending with 12 for December.

OPTIONS

month_name

name or abbreviation of month. Case is ignored Once enough characters are found to uniquely identify a month the rest of the name is ignored.

RETURNS

imonth month number returned. If the name is not recognized a **-1** is returned.

EXAMPLE

Sample program:

```
program demo_mo2v
use M_time, only : mo2v
implicit none
   write(*,*)mo2v("April")
   write(*,*)mo2v('Apr')
   ! NOTE: still matches September, as "SE" was enough
   write(*,*)mo2v('sexember')
   write(*,*)mo2v('unknown')  ! returns -1
end program demo_mo2v
```

results:

```
>   4
>   4
>   9
>  -1
```

AUTHOR

John S. Urban, 2015

LICENSE

Public Domain

NAME

moon_fullness(3f) - [M_time] return percentage of moon phase from new to full **(LICENSE:PD)**

SYNOPSIS

```
function moon_fullness(datin)

   integer,intent(in)            :: datin(8)
   integer                       :: moon_fullness
```

DESCRIPTION

This procedure is used to support the %P field descriptor for the **fmtdate**(3f) routine.

The moon circles the earth every 29.530588853 days on average, so pick a starting point and count. A new moon occurred at January 6, 2000, 18:14 UTC. Then it is easy to count the number of days since the last new moon. This is an approximate calculation.

OPTIONS

dat DAT Date array describing input date

RESULTS

moon_fullness
0 is a new or dark moon, 100 is a full moon, + for waxing and - for waning.

EXAMPLES

Sample:

```
program demo_moon_fullness
use M_time, only : now
use M_time, only : phase_of_moon
use M_time, only : moon_fullness
implicit none
integer             :: dat(8)
   ! generate DAT array
   call date_and_time(values=dat)
   ! show DAT array
   write(*,'(" Today is:",*(i0:,":"))')dat
   ! the %p and %P fields are supported by fmtdate(3f)
   write(*,*)now('The phase of the moon is %p, with a fullness of %P')
   write(*,'(1x,*(a))',advance='no')'The phase of the moon is ',trim( phase_of_mo
   write(*,'(1x,a,i0,a)')'with a fullness of ', moon_fullness(dat),'%'
end program demo_moon_fullness
```

Sample output:

```
 Today is:2018:11:3:-240:20:18:44:245
 The phase of the moon is Waning crescent, with a fullness of -30%
 The phase of the moon is Waning crescent, with a fullness of -30%
```

AUTHOR

John S. Urban, 2015

LICENSE

Public Domain

NAME

now(3f) - [M_time] return string representing current time given format **(LICENSE:PD)**

SYNOPSIS

```
function now(format) RESULT (timestr)

    character(len=*),intent(in)     :: format  ! input format string
    character(len=:),allocatable    :: timestr ! formatted date
```

DESCRIPTION

The **now**(3f) function is a call to the **fmtdate**(3f) function using the current date and time. That is, it is a convenient way to print the current date and time.

OPTIONS

format string describing how to *format* the current date and time. For a complete description of the formatting macros supported see **fmtdate_usage**(3f).

RETURNS

timestr formatted output string representing date

EXAMPLE

Sample Program:

```
    program demo_now
    use M_time, only : now
    implicit none
       write(*,*)now("The current date is %w, %l %d, %Y %H:%m:%s %N")
       call showme()
    contains
    subroutine showme() ! see all formatting options
    use M_time, only : fmtdate_usage
       call fmtdate_usage() ! see all formatting options
    end subroutine showme

    end program demo_now
```

results:

```
      The current date is Sun, Jul 17th, 2016 01:21:35 PM
       ::
       :: description of all formatting options will appear here
       ::
```

AUTHOR

John S. Urban, 2015

LICENSE

Public Domain

NAME

o2d(3f) - [M_time] converts Ordinal day to DAT date-time array **(LICENSE:PD)**

SYNOPSIS

```
function o2d(ordinal,[year]) result (dat)

    integer,intent(in) :: ordinal  ! the day of the year
    integer,optional   :: year     ! year
    integer            :: dat(8)   ! date time array
```

DESCRIPTION

Given an Ordinal day of the year return a date in the form of a "DAT" array.

OPTIONS

ordinal

The day of the year for the given year, where Jan 1st=1.

year An optional year for the ordinal day. If not present the current year is assumed.

RETURNS

dat Integer array holding a "DAT" array, similar in structure to the array returned by the intrinsic **DATE_AND_TIME**(3f). The timezone value is from the current time on the current platform.

```
dat=[year,month,day,timezone,hour,minutes,seconds,milliseconds]
```

EXAMPLE

Sample program:

```
program demo_o2d
use M_time, only : o2d,fmtdate
implicit none
integer :: year
   do year=2004,2008
      write(*,*)'100th day of ',year,' is ',fmtdate(o2d(100,year))
   enddo
   write(*,*)'100th day of this year is ',fmtdate(o2d(100))
end program demo_o2d
```

results:

```
100th day of 2004 is Friday, April 9th, 2004 00:00:00 PM UTC-02:40
100th day of 2005 is Sunday, April 10th, 2005 00:00:00 PM UTC-02:40
100th day of 2006 is Monday, April 10th, 2006 00:00:00 PM UTC-02:40
100th day of 2007 is Tuesday, April 10th, 2007 00:00:00 PM UTC-02:40
100th day of 2008 is Wednesday, April 9th, 2008 00:00:00 PM UTC-02:40
100th day of this year is Saturday, April 9th, 2016 00:00:00 PM UTC-02:40
```

AUTHOR

John S. Urban, 2015

LICENSE

Public Domain

NAME

ordinal_seconds(3f) - [M_time] seconds since beginning of year **(LICENSE:PD)**

SYNOPSIS

```
function ordinal_seconds()

    integer :: ordinal_seconds
```

DESCRIPTION

Return number of seconds since beginning of current year.

Before using this routine consider the consequences if the application is running at the moment a new year begins.

2 147 483 647 / 31 536 000 ==> 68.09625973490613901572 years

EXAMPLE

sample program

```
program demo_ordinal_seconds
use M_time, only : ordinal_seconds
implicit none
character(len=1) :: paws
integer          :: ios
integer          :: istart, iend
istart=ordinal_seconds()
write(*,'(a)',advance='no')'now pause. Enter return to continue ...'
read(*,'(a)',iostat=ios) paws
iend=ordinal_seconds()
write(*,*)'that took ',iend-istart,'seconds'
write(*,*)istart,iend
end program demo_ordinal_seconds
```

AUTHOR

John S. Urban, 2015

LICENSE

Public Domain

NAME

ordinal_to_date(3f) - [M_time] when given a valid year and day of the year returns the DAT array for the date **(LICENSE:PD)**

SYNOPSIS

```
subroutine ordinal_to_date(yyyy, ddd, dat)

      integer, intent(in)   :: yyyy
      integer, intent(in)   :: ddd
      integer, intent(out)  :: dat
```

DESCRIPTION

When given a valid year, YYYY, and day of the year, DDD, returns the date as a DAT date array

OPTIONS

yyyy known year

ddd known ordinal day of the year

RETURNS

dat DAT array describing the date

EXAMPLE

Sample program:

```
   program demo_datesub
   use M_time, only : ordinal_to_date
   implicit none
   INTEGER            :: yyyy, ddd, mm, dd
   integer            :: dat(8)
   integer            :: ios

     INFINITE: do
        write(*,'(a)',advance='no')'Enter year YYYY and ordinal day of year DD '
        read(*,*,iostat=ios)yyyy,ddd
        if(ios.ne.0)exit INFINITE
        ! recover month and day from year and day number.
        call ordinal_to_date(yyyy, ddd, dat)
        mm=dat(2)
        dd=dat(3)
      enddo INFINITE

      end program demo_datesub
```

NAME

phase_of_moon(3f) - [M_time] return name for phase of moon for given date **(LICENSE:PD)**

SYNOPSIS

```
function phase_of_moon(datin)

   integer,intent(in)             :: datin(8)
   character(len=:),allocatable   :: phase_of_moon
```

DESCRIPTION

Phases Of The Moon

This procedure is used to support the %p field descriptor for the **fmtdate**(3f) routine.

The moon circles the earth every 29.530588853 days on average, so pick a starting point and count. A new moon occurred at Julian date 2451550.1 (January 6, 2000, 18:14 UTC). Then it is easy to count the number of days since the last new moon. This is an approximate calculation.

There are eight generally recognized phases of the moon in common use

- new or dark
- waxing crescent
- first quarter
- waxing gibbous
- full
- waning gibbous
- laster quarter
- waning crescent

To calculate the phase of the moon simply divide the days since the last new moon by eight and select the appropriate phase.

Note that technically the four states (new, first quarter, full, third quarter) are events not phases. That is to say, the moon is technically only new for an instant.

EXAMPLES

Sample:

```
program demo_phase_of_moon
use M_time, only : now
use M_time, only : phase_of_moon
use M_time, only : moon_fullness
implicit none
integer             :: dat(8)
   ! generate DAT array
   call date_and_time(values=dat)
   ! show DAT array
   write(*,'(" Today is:",*(i0:,":"))')dat
   ! the %p and %P fields are supported by fmtdate(3f)
   write(*,*)now('The phase of the moon is %p, with a fullness of %P')
   write(*,'(1x,*(a))',advance='no')'The phase of the moon is ',trim( phase_of_mo
   write(*,'(1x,a,i0,a)')'with a fullness of ', moon_fullness(dat),'%'
end program demo_phase_of_moon
```

Sample output:

```
 Today is:2018:11:3:-240:20:18:44:245
 The phase of the moon is Waning crescent, with a fullness of -30%
```

```
      The phase of the moon is Waning crescent, with a fullness of -30%
```

AUTHOR

John S. Urban, 2015

LICENSE

Public Domain

NAME

sec2days(3f) - [M_time] convert seconds to string of form dd-hh:mm:ss **(LICENSE:PD)**

SYNOPSIS

```
function sec2days(seconds,crop) result(dhms)

    real(kind=realtime),intent(in) :: seconds
      or
    integer,intent(in)             :: seconds
      or
    real,intent(in)                :: seconds
      or
    character(len=*)               :: seconds

    logical,intent(in),optional    :: crop
    character(len=:),allocatable   :: dhms
```

DESCRIPTION

Given a number of seconds convert it to a string of the form

```
dd-hh:mm:ss
```

where dd is days, hh hours, mm minutes and ss seconds.

OPTIONS

seconds

number of seconds to convert to string of form dd-hh:mm:ss. May be of type INTEGER, REAL, **REAL**(KIND=REALTIME), or CHARACTER.

CHARACTER strings may be of the form NNdNNhNNmNNs. Case,spaces and underscores are ignored. Allowed aliases for d,h,m, and s units are

```
d -  days,day
m -  minutes,minute,min
h -  hours,hour,hrs,hr
s -  seconds,second,sec
```

The numeric values may represent floating point numbers.

crop if .true., remove leading zero day values or day and hour values. Optional, defaults to .false. .

RETURNS

dmhs the returned string of form [d:h:]m:s

EXAMPLE

Sample Program:

```
program demo_sec2days
use M_time, only : sec2days
implicit none
   write(*,*)sec2days(129860)
   write(*,*)sec2days(80000.0d0)
   write(*,*)sec2days(80000.0,crop=.true.)
   write(*,*)sec2days('1 day 2.0hr 100 min 300.0seconds')
end program demo_sec2days
```

results:

```
1-12:04:20
0-22:13:20
22:13:20
```

```
     1-03:45:00
```

AUTHOR

John S. Urban, 2015

LICENSE

Public Domain

NAME

u2d(3f) - [M_time] given Unix Epoch Time returns DAT date-time array **(LICENSE:PD)**

SYNOPSIS

```
function u2d(unixtime) result (dat)

    class(*),intent(in),optional      :: unixtime
    ! integer
    ! real
    ! real(kind=realtime)

    integer                           :: dat(8)
```

DESCRIPTION

OPTIONS

unixtime

The "Unix Epoch" time, or the number of seconds since 00:00:00 on January 1st, 1970, UTC. If not present, use current time.

RETURNS

dat Integer array holding a "DAT" array, similar in structure to the array returned by the intrinsic **DATE_AND_TIME**(3f).

```
dat=[year,month,day,timezone,hour,minutes,seconds,milliseconds]
```

EXAMPLE

Sample program:

```
program demo_u2d
use M_time, only : u2d, d2u, fmtdate, realtime
implicit none
real(kind=realtime) :: today
integer :: dat(8)
   call date_and_time(values=dat) ! get the date using intrinsic
   today=d2u(dat)                 ! convert today to Julian Date
   write(*,*)'Today=',fmtdate(u2d(today))
   write(*,*)'Yesterday=',fmtdate(u2d(today-86400.0d0)) ! subtract day
   write(*,*)'Tomorrow=',fmtdate(u2d(today+86400.0d0))  ! add day
end program demo_u2d
```

results:

```
Today=Tuesday, July 19th, 2016 11:10:08 AM
Yesterday=Monday, July 18th, 2016 11:10:08 AM
Tomorrow=Wednesday, July 20th, 2016 11:10:08 AM
```

AUTHOR

John S. Urban, 2015

LICENSE

Public Domain

NAME

unix_to_date(3f) - [M_time] converts Unix Epoch Time to DAT date-time array **(LICENSE:PD)**

SYNOPSIS

```
subroutine unix_to_date(unixtime,dat,ierr)

    real(kind=realtime),intent(in) :: unixtime ! Unix time (seconds)
    integer,intent(out)            :: dat(8)   ! date and time array
    integer,intent(out)            :: ierr     ! 0 for successful execution
```

DESCRIPTION

Converts a Unix Epoch Time (UET) to a DAT date-time array.

OPTIONS

unixtime

The "Unix Epoch" time, or the number of seconds since 00:00:00 on January 1st, 1970, UTC; of type **real**(kind=realtime).

RETURNS

dat Integer array holding a "DAT" array, similar in structure to the array returned by the intrinsic **DATE_AND_TIME**(3f).

```
dat=[year,month,day,timezone,hour,minutes,seconds,milliseconds]
```

ierr Error code. If 0 no error occurred.

EXAMPLE

Sample program:

```
program demo_unix_to_date
use M_time, only : unix_to_date, u2d, fmtdate, realtime
implicit none
real(kind=realtime)           :: unixtime
real(kind=realtime),parameter :: DAY=86400.0d0 ! seconds in a day
integer                       :: dat(8)
integer                       :: ierr
   unixtime=1468939038.4639933d0            ! sample Unix Epoch time
   call unix_to_date(unixtime,dat,ierr)     ! create DAT array for today
   write(*,*)'Sample Date=',fmtdate(dat)
   call unix_to_date(unixtime-DAY,dat,ierr) ! go back one day
   write(*,*)'Day Before =',fmtdate(dat)    ! subtract day and print
   call unix_to_date(unixtime+DAY,dat,ierr) ! go forward one day
   write(*,*)'Day After  =',fmtdate(dat)    ! add day print
end program demo_unix_to_date
```

results:

```
Sample Date=Tuesday, July 19th, 2016 10:37:18 AM
Day Before =Monday, July 18th, 2016 10:37:18 AM
Day After  =Wednesday, July 20th, 2016 10:37:18 AM
```

AUTHOR

John S. Urban, 2015

LICENSE

Public Domain

NAME

v2mo(3f) - [M_time] returns the month name of a Common month number **(LICENSE:PD)**

SYNOPSIS

```
function v2mo(imonth) result(month_name)

    integer,intent(in)            :: imonth      ! month number (1-12)
    character(len=:),allocatable :: month_name  ! month name
```

DESCRIPTION

Given a Common Calendar month number, return the name of the month as a string.

OPTIONS

imonth Common month number (1-12). If out of the allowable range the month name returned will be 'UNKNOWN'.

RETURNS

month_name
A string representing a month name or the word 'UNKNOWN'

EXAMPLE

Sample program:

```
program demo_v2mo
use M_time, only : v2mo
implicit none
integer :: i
   do i=1,13
      write(*,*)v2mo(i)
   enddo
end program demo_v2mo
```

results:

```
January
February
March
April
May
June
July
August
September
October
November
December
UNKNOWN.
```

AUTHOR

John S. Urban, 2015

LICENSE

Public Domain

NAME

w2d(3f) - [M_time] calculate DAT date-time array from iso-8601 Week-numbering year date yyyy-Www-d **(LICENSE:PD)**

SYNOPSIS

```
subroutine w2d(iso_year,iso_week,iso_weekday,dat)

    integer,intent(in)      :: iso_year, iso_week, iso_weekday
    integer,intent(out)     :: dat(8)     ! output date array
```

DESCRIPTION

Given an ISO-8601 week return a "DAT" array defining a date and time, The ISO-8601 is supplied as three integer values defining the ISO year, week of year and weekday.

OPTIONS

iso_year

ISO-8601 year number for the given date

iso_week

ISO-8601 week number for the given date

iso_weekday

ISO-8601 weekday number for the given date

iso_name

ISO-8601 Week string for the data in the form "yyyy-Www-d".

RETURNS

dat "DAT" array (an integer array of the same format as the array returned by the intrinsic **DATE_AND_TIME**(3f)) describing the date to be used, which is the basic time description used by the other **M_time**(3fm) module procedures.

EXAMPLE

Sample program:

```
program demo_w2d
use M_time, only : w2d, fmtdate
implicit none
   write(*,'(a)')'Given Monday 29 December 2008 is written "2009-W01-1"'
   call printit(2009,1,1)
   write(*,'(a)')'Given Sunday 3 January 2010 is written "2009-W53-7"'
   call printit(2009,53,7)
   write(*,'(a)')'Given the Gregorian date Sun 31 December 2006 is written 2006-
   call printit(2006,52,7)
   write(*,'(a)')'Given 27 September 2008 is 2008-W39-6'
   call printit(2008,39,6)
contains
subroutine printit(iso_year,iso_week,iso_weekday)
integer  :: iso_year, iso_week, iso_weekday ! ISO-8601 Week:   2016-W29-1
integer  :: dat(8)                          ! input date array
   call w2d(iso_year,iso_week,iso_weekday,dat)
   write(*,'(a,i0)')'GIVEN:           '
   write(*,'(a,i0)')'ISO-8601 year    ',iso_year
   write(*,'(a,i0)')'ISO-8601 week    ',iso_week
   write(*,'(a,i0)')'ISO-8601 weekday ',iso_weekday
   write(*,'(a,i0)')'RESULT:          '
   write(*,'(a,*(i0:,","))')'   DAT array        ',dat
   write(*,'(a,/,77("="))')'    '//fmtdate(dat,'long')
end subroutine printit
```

```
    end program demo_w2d
```

Results:

```
    Given Monday 29 December 2008 is written "2009-W01-1"
    GIVEN:
    ISO-8601 year    2009
    ISO-8601 week    1
    ISO-8601 weekday 1
    RESULT:
       DAT array        2008,12,29,-240,0,0,0,0
        Monday, December 29th, 2008 12:00:00 AM UTC-04:00
    ==============================================================================
    Given Sunday 3 January 2010 is written "2009-W53-7"
    GIVEN:
    ISO-8601 year    2009
    ISO-8601 week    53
    ISO-8601 weekday 7
    RESULT:
       DAT array        2010,1,3,-240,0,0,0,0
        Sunday, January 3rd, 2010 12:00:00 AM UTC-04:00
    ==============================================================================
    Given the Gregorian date Sun 31 December 2006 is written 2006-W52-7
    GIVEN:
    ISO-8601 year    2006
    ISO-8601 week    52
    ISO-8601 weekday 7
    RESULT:
       DAT array        2006,12,31,-240,0,0,0,0
        Sunday, December 31st, 2006 12:00:00 AM UTC-04:00
    ==============================================================================
    Given 27 September 2008 is 2008-W39-6
    GIVEN:
    ISO-8601 year    2008
    ISO-8601 week    39
    ISO-8601 weekday 6
    RESULT:
       DAT array        2008,9,27,-240,0,0,0,0
        Saturday, September 27th, 2008 12:00:00 AM UTC-04:00
    ==============================================================================
```

DEFINITION

The ISO-8601 date and time standard was issued by the International Organization for Standardization (ISO). It is used (mainly) in government and business for fiscal years, as well as in timekeeping. The system specifies a week year atop the Gregorian calendar by defining a notation for ordinal weeks of the year.

An ISO week-numbering year (also called ISO year informally) has 52 or 53 full weeks. That is 364 or 371 days instead of the usual 365 or 366 days. The extra week is referred to here as a leap week, although ISO-8601 does not use this term. Weeks start with Monday. The first week of a year is the week that contains the first Thursday of the year (and, hence, always contains 4 January). ISO week year numbering therefore slightly deviates from the Gregorian for some days close to January 1st.

METHOD

Calculating a date given the year, week number and weekday

This method requires that one know the weekday of 4 January of the year in question. Add 3 to the number of this weekday, giving a correction to be used for dates within this year.

Method: Multiply the week number by 7, then add the weekday. From this sum subtract the correction for the year. The result is the ordinal date, which can be converted into a calendar date. If the ordinal date thus obtained is zero or negative, the date belongs to the previous calendar year; if greater than the number of days in the year, to the following year.

Example: year 2008, week 39, Saturday (day 6) Correction for 2008: 5 + 3 = 8 (39 x 7) + 6 = 279 279 - 8 = 271 Ordinal day 271 of a leap year is day 271 - 244 = 27 September Result: 27 September 2008

ISO_NAME

Week date representations are in the format YYYYWww-D.

- [YYYY] indicates the ISO week-numbering year which is slightly different from the traditional Gregorian calendar year.
- [Www] is the week number prefixed by the letter W, from W01 through W53.
- [D] is the weekday number, from 1 through 7, beginning with Monday and ending with Sunday.

For example, the Gregorian date 31 December 2006 corresponds to the Sunday of the 52nd week of 2006, and is written

```
2006-W52-7 (extended form)
or
2006W527 (compact form).
```

REFERENCE

From Wikipedia, the free encyclopedia 2016-08-08

AUTHOR

John S. Urban, 2015

LICENSE

Public Domain