NAME

Date::Manip::DM6 − Date manipulation routines

SYNOPSIS

```
use Date::Manip;

$version = DateManipVersion($flag);

Date_Init("VAR=VAL","VAR=VAL",...);

$date = ParseDate(\@args [,@opts]);
$date = ParseDate($string [,@opts]);
$date = ParseDate(\$string [,@opts]);

$date = ParseDateString($string [,@opts]);

$date = ParseDateFormat($format,$string);

@date = UnixDate($date,@format);
$date = UnixDate($date,@format);

$delta = ParseDateDelta(\@args   [,$mode]);
$delta = ParseDateDelta($string  [,$mode]);
$delta = ParseDateDelta(\$string [,$mode]);

@str = Delta_Format($delta, [$mode,] $dec,@format);
$str = Delta_Format($delta, [$mode,] $dec,@format);

$recur = ParseRecur($string,$base,$date0,$date1,$flags);
@dates = ParseRecur($string,$base,$date0,$date1,$flags);

$flag = Date_Cmp($date1,$date2);

$d = DateCalc($d1,$d2 [,$errref] [,$mode]);

$date = Date_SetTime($date,$hr,$min,$sec);
$date = Date_SetTime($date,$time);

$date = Date_SetDateField($date,$field,$val [,$nocheck]);

$date = Date_GetPrev($date,$dow,$today,$hr,$min,$sec);
$date = Date_GetPrev($date,$dow,$today,$time);

$date = Date_GetNext($date,$dow,$today,$hr,$min,$sec);
$date = Date_GetNext($date,$dow,$today,$time);

$name = Date_IsHoliday($date);
@name = Date_IsHoliday($date);

$listref = Events_List($date);
$listref = Events_List($date0,$date1);

$date = Date_ConvTZ($date,$from,$to);

$flag = Date_IsWorkDay($date [,$flag]);
```

```
$date = Date_NextWorkDay($date,$off [,$time]);

$date = Date_PrevWorkDay($date,$off [,$time]);

$date = Date_NearestWorkDay($date [,$tomorrowfirst]);
```

In the following routines, `$y` may be entered as either a 2 or 4 digit year (it will be converted to a 4 digit year based on the variable YYtoYYYY described below). Month and day should be numeric in all cases.

```
$day = Date_DayOfWeek($m,$d,$y);
$secs = Date_SecsSince1970($m,$d,$y,$h,$mn,$s);
$secs = Date_SecsSince1970GMT($m,$d,$y,$h,$mn,$s);
$days = Date_DaysSince1BC($m,$d,$y);
$day = Date_DayOfYear($m,$d,$y);
($y,$m,$d,$h,$mn,$s) = Date_NthDayOfYear($y,$n);
$days = Date_DaysInYear($y);
$days = Date_DaysInMonth($m,$y);
$wkno = Date_WeekOfYear($m,$d,$y,$first);
$flag = Date_LeapYear($y);
$day = Date_DaySuffix($d);
$tz = Date_TimeZone();
```

ROUTINES

DateManipVersion

```
$version = DateManipVersion($flag);
```

Returns the version of Date::Manip. If `$flag` is non-zero, timezone information is also returned.

Date_Init

```
Date_Init("VAR=VAL","VAR=VAL",...);
```

The Date_Init function is used to set any of the Date::Manip configuration variables described in the Date::Manip::Config document.

The strings to pass in are of the form ‘‘VAR=VAL’’. Any number may be included and they can come in any order. VAR may be any configuration variable. VAL is any allowed value for that variable. For example, to switch from English to French and use non-US format (so that 12/10 is Oct 12), do the following:

```
Date_Init("Language=French","DateFormat=non-US");
```

Note that variables are parsed in the order they are given, so ‘‘DateFormat=non−US’’, ‘‘ConfigFile=./manip.cnf’’ may not give the expected result. To be safe, ConfigFile should always appear first in the list.

ParseDate

```
$date = ParseDate(\@args [,@opts]);
$date = ParseDate($string [,@opts]);
$date = ParseDate(\$string [,@opts]);
```

This takes an array or a string containing a date and parses it. When the date is included as an array (for example, the arguments to a program) the array should contain a valid date in the first one or more elements (elements after a valid date are ignored). Elements containing a valid date are shifted from the array. The largest possible number of elements which can be correctly interpreted as a valid date are always used. If a string is entered rather than an array, that string is tested for a valid date. The string is unmodified, even if passed in by reference.

The ParseDate routine is primarily used to handle command line arguments. If you have a command where you want to enter a date as a command line argument, you can use Date::Manip to make something like the following work:

```
mycommand -date Dec 10 1997 -arg -arg2
```

No more reading man pages to find out what date format is required in a man page.

The `@opts` argument may contain values that can be passed to the `Date::Manip::Date::parse` method.

Historical note: this is originally why the Date::Manip routines were written (though long before they were released as the Date::Manip module). I was using a bunch of programs (primarily batch queue managers) where dates and times were entered as command line options and I was getting highly annoyed at the many different (but not compatible) ways that they had to be entered. Date::Manip originally consisted of basically 1 routine which I could pass ''@ARGV'' to and have it remove a date from the beginning.

ParseDateString

```
$date = ParseDateString($string [,@opts]);
```

This parses a string containing a date and returns it. Refer to the Date::Manip::Date documentation for valid date formats. The date returned is in the local time zone.

The `@opts` argument may contain values that can be passed to the `Date::Manip::Date::parse` method.

ParseDateFormat

```
$date = ParseDateFormat($format,$string);
```

This parses a string containing a date based on a format string and returns the date. Refer to the Date::Manip::Date documentation for the parse_format method for more information. The date returned is in the local time zone.

UnixDate

```
$out = UnixDate($date,$in);
@out = UnixDate($date,@in);
```

This takes a date and a list of strings containing formats roughly identical to the format strings used by the UNIX **date**(1) command. Each format is parsed and an array of strings corresponding to each format is returned.

The formats are described in the Date::Manip::Date document.

ParseDateDelta

```
$delta = ParseDateDelta(\@args   [,$mode]);
$delta = ParseDateDelta($string  [,$mode]);
$delta = ParseDateDelta(\$string [,$mode]);
```

In the first form, it takes an array and shifts a valid delta from it. In the other two forms, it parses a string to see if it contains a valid delta.

A valid delta is returned if found. Otherwise, an empty string is returned.

The delta can be converted to 'exact', 'semi', or 'approx' using the Date::Manip::Delta::convert method if `$mode` is passed in.

Delta_Format

```
$out = Delta_Format($delta [,$mode], $dec,$in);
@out = Delta_Format($delta [,$mode], $dec,@in);
```

This is similar to the UnixDate routine except that it extracts information from a delta.

When formatting fields in a delta, the Date::Manip 6.00 formats have changed and are much more powerful. The old 5.xx formats are still available for the Delta_Format command for backward compatibility. These formats include:

```
%Xv  : print the value of the field X
```

```
%Xd  : print the value of the field X and all
       smaller units in terms of X

%Xh  : print the value of field X and all
       larger units in terms of X

%Xt  : print the value of all fields in
       terms of X
```

These make use of the `$mode` and `$dec` arguments to determine how to format the information.

`$dec` is an integer, and is required, It tells the number of decimal places to use.

`$mode` is either ‘‘exact’’, ‘‘semi’’, or ‘‘approx’’ and defaults to ‘‘exact’’ if it is not included.

In ‘‘exact’’ mode, only exact relationships are used. This means that there can be no mixing of the Y/M, W/D, and H/MN/S segments (for non-business deltas, or Y/M, W, and D/H/MN/S segments for business deltas) because there is no exact relation between the fields of each set.

In ‘‘semi’’ mode, the semi-approximate relationships are used so there is no mixing between Y/M and W/D/H/MN/S.

In ‘‘approx’’ mode, approximate relationships are used so all fields can mix.

The semi-approximate and approximate relationships are described in the Date::Manip::Delta manual.

So, in ‘‘exact’’ mode, with a non-business delta, and `$dec` = 2, the following are equivalent:

```
old style    new style
---------    ---------
%Xv          %Xv
%hd          %.2hhs
%hh          %.2hdh
%ht          %.2hds
%yd          %.2yyM
```

In ‘‘approximate’’ mode, the following are equivalent:

```
old style    new style
---------    ---------
%Xv          %Xv
%hd          %.2hhs
%hh          %.2hdh
%ht          %.2hys
%yd          %.2yys
```

If you want to use the new style formats in Delta_Format, use one of the calls:

```
Delta_Format($delta, @in);
Delta_Format($delta, undef, @in);
```

If the first element of `@in` is an integer, you have to use the 2nd form.

The old formats will remain available for the time being, though at some point they may be deprecated.

DateCalc

```
 $d = DateCalc($d1,$d2 [,\$err] [,$mode]);
```

This takes two dates, deltas, or one of each and performs the appropriate calculation with them. Dates must be a string that can be parsed by ParseDateString. Deltas must be a string that can be parsed by ParseDateDelta. Two deltas add together to form a third delta. A date and a delta returns a 2nd date.

Two dates return a delta (the difference between the two dates).

Since the two items can be interpreted as either dates or deltas, and since many strings can be interpreted as both a date or a delta, it is a good idea to pass the input through ParseDateDelta, if appropriate if there is any ambiguity. For example, the string ‘‘09:00:00’’ can be interpreted either as a date (today at 9:00:00) or a delta (9 hours). To avoid unexpected results, avoid calling DateCalc as:

```
$d = DateCalc("09:00:00",$someothervalue);
```

Instead, call it as:

```
$d = DateCalc(ParseDate("09:00:00"),$someothervalue);
```

to force it to be a date, or:

```
$d = DateCalc(ParseDateDelta("09:00:00"),$someothervalue);
```

to force it to be a delta. This will avoid unexpected results. Passing something through ParseDate is optional since they will be treated as dates by default (and for performance reasons, you’re better off not calling ParseDate).

If there is no ambiguity, you are better off NOT doing this for performance reasons. If the delta is a business delta, you definitely should NOT do this.

One other thing to note is that when parsing dates, a delta can be interpreted as a date relative to now. DateCalc will ALWAYS treat a delta as a delta, NOT a date.

For details on how calculations are done, refer to the Date::Manip::Calc documentation.

By default, math is done using an exact mode.

If two deltas, or a date and a delta are passed in, `$mode` may be used to force the delta to be either business or non-business mode deltas. If `$mode` is 0 or 1, the delta(s) will be non-business. Otherwise, they will be business deltas. If `$mode` is passed in, it will be used only if the business or non-business state was not explicitly set in the delta. `$mode` can also be an y of the modes discussed in the Date::Manip::Calc documentation.

If two dates are passed in, `$mode` is used to determine the type of calculation. By default, an exact delta is produced. If `$mode` is 1, an approximate delta is produced. If `$mode` is 2, a business approximate (bapprox) mode calculation is done. If `$mode` is 3, a exact business mode delta is produced.

If \$err is passed in, it is set to:

```
1 is returned if $d1 is not a delta or date
2 is returned if $d2 is not a delta or date
3 if any other error occurs.
```

This argument is optional, but if included, it must come before `$mode`.

Nothing is returned if an error occurs.

ParseRecur

```
$recur = ParseRecur($string [,$base,$date0,$date1,$flags]);
@dates = ParseRecur($string [,$base,$date0,$date1,$flags]);
```

This parses a string containing a recurrence and returns a fully specified recurrence, or a list of dates referred to.

`$string` can be any of the forms:

```
FREQ
FREQ*FLAGS
FREQ*FLAGS*BASE
FREQ*FLAGS*BASE*DATE0
FREQ*FLAGS*BASE*DATE0*DATE1
```

where FREQ is a frequence (see the Date::Manip::Delta documentation), FLAGS is a comma separated list of flags, and BASE, DATE0, and DATE1 are date strings. The dates and flags can also be passed in as `$base`, `$date0`, `$date1`, and `$flags`, and these will override any values in `$string`.

In scalar context, the fully specified recurrence (or as much information as is available) will be returned. In list context, a list of dates will be returned.

Date_Cmp

```
$flag = Date_Cmp($date1,$date2);
```

This takes two dates and compares them. Any dates that can be parsed will be compared.

Date_GetPrev

```
$date = Date_GetPrev($date,$dow, $curr [,$hr,$min,$sec]);
$date = Date_GetPrev($date,$dow, $curr [,$time]);
$date = Date_GetPrev($date,undef,$curr,$hr,$min,$sec);
$date = Date_GetPrev($date,undef,$curr,$time);
```

This takes a date (any string that may be parsed by ParseDateString) and finds the previous occurrence of either a day of the week, or a certain time of day.

This is documented in the ''prev'' method in Date::Manip::Date, except that here, `$time` is a string (HH, HH:MN:, or HH:MN:SS), and `$dow` may be a string of the form ''Fri'' or ''Friday''.

Date_GetNext

```
$date = Date_GetNext($date,$dow, $curr [,$hr,$min,$sec]);
$date = Date_GetNext($date,$dow, $curr [,$time]);
$date = Date_GetNext($date,undef,$curr,$hr,$min,$sec);
$date = Date_GetNext($date,undef,$curr,$time);
```

Similar to Date_GetPrev.

Date_SetTime

```
$date = Date_SetTime($date,$hr,$min,$sec);
$date = Date_SetTime($date,$time);
```

This takes a date (any string that may be parsed by ParseDateString) and sets the time in that date. For example, one way to get the time for 7:30 tomorrow would be to use the lines:

```
$date = ParseDate("tomorrow");
$date = Date_SetTime($date,"7:30");
```

`$time` is a string (HH, HH:MN, or HH:MN:SS).

Date_SetDateField

```
$date = Date_SetDateField($date,$field,$val);
```

This takes a date and sets one of its fields to a new value. `$field` is any of the strings ''y'', ''m'', ''d'', ''h'', ''mn'', ''s'' (case insensitive) and `$val` is the new value.

Date_IsHoliday

```
$name = Date_IsHoliday($date);
@name = Date_IsHoliday($date);
```

This returns undef if `$date` is not a holiday, or a string containing the name of the holiday otherwise (or a list of names in list context). An empty string is returned for an unnamed holiday.

Date_IsWorkDay

```
$flag = Date_IsWorkDay($date [,$flag]);
```

This returns 1 if `$date` is a work day. If`$flag` is non-zero, the time is check ed to see if it falls within work hours. It returns an empty string if `$date` is not valid.

Events_List

```
$ref = Events_List($date);
$ref = Events_List($date,0      [,$flag]);
$ref = Events_List($date,$date1 [,$flag]);
```

This returns a list of events. If `$flag` is not given, or is equal to 0, the list (returned as a reference) is similar to the the list returned by the Date::Manip::Date::list_events method with `$format` = ‘‘dates’’. The only difference is that it is formatted slightly different to be backward compatible with Date::Manip 5.xx.

The data from the list_events method is:

```
( [DATE1, NAME1a, NAME1b, ...],
  [DATE2, NAME2a, NAME2b, ...],
  ...
)
```

The reference returned from Events_List (if `$flag` = 0) is:

```
[ DATE1, [NAME1a, NAME1b, ...],
  DATE2, [DATE2a, DATE2b, ...],
  ...
]
```

For example, if the following events are defined:

```
2000-01-01 ; 2000-03-21  = Winter
2000-03-22 ; 2000-06-21  = Spring
2000-02-01               = Event1
2000-05-01               = Event2
2000-04-01-12:00:00      = Event3
```

the following examples illustrate the function:

```
Events_List("2000-04-01")
 => [ 2000040100:00:00, [ Spring ] ]

Events_List("2000-04-01 12:30");
 => [ 2000040112:30:00, [ Spring, Event3 ] ]

Events_List("2000-04-01",0);
 => [ 2000040100:00:00, [ Spring ],
      2000040112:00:00, [ Spring, Event3 ],
      2000040113:00:00, [ Spring ] ]

Events_List("2000-03-15","2000-04-10");
 => [ 2000031500:00:00, [ Winter ],
      2000032200:00:00, [ Spring ]
      2000040112:00:00, [ Spring, Event3 ]
      2000040113:00:00, [ Spring ] ]
```

If `$flag` is 1, then a tally of the amount of time given to each event is returned. Time for which two or more events apply is counted for both.

```
Events_List("2000-03-15","2000-04-10",1);
 => { Event3 => +0:0:+0:0:1:0:0,
      Spring => +0:0:+2:4:23:0:0,
      Winter => +0:0:+1:0:0:0:0
    }
```

When `$flag` is 2, a more complex tally with no event counted twice is returned.

```
Events_List("2000-03-15","2000-04-10",2);
 => { Event3+Spring => +0:0:+0:0:1:0:0,
      Spring        => +0:0:+2:4:22:0:0,
      Winter        => +0:0:+1:0:0:0:0
    }
```

The hash contains one element for each combination of events.

In both of these cases, there may be a hash element with an empty string as the key which contains the amount of time with no events active.

Date_DayOfWeek

```
$day = Date_DayOfWeek($m,$d,$y);
```

Returns the day of the week (1 for Monday, 7 for Sunday).

Date_SecsSince1970

```
$secs = Date_SecsSince1970($m,$d,$y,$h,$mn,$s);
```

Returns the number of seconds since Jan 1, 1970 00:00 (negative if date is earlier) in the current timezone.

Date_SecsSince1970GMT

```
$secs = Date_SecsSince1970GMT($m,$d,$y,$h,$mn,$s);
```

Returns the number of seconds since Jan 1, 1970 00:00 GMT (negative if date is earlier). Note that the date is still given in the current timezone, NOT GMT.

Date_DaysSince1BC

```
$days = Date_DaysSince1BC($m,$d,$y);
```

Returns the number of days since Dec 31, 1BC. This includes the year 0001.

Date_DayOfYear

```
$day = Date_DayOfYear($m,$d,$y);
```

Returns the day of the year (1 to 366)

Date_NthDayOfYear

```
($y,$m,$d,$h,$mn,$s) = Date_NthDayOfYear($y,$n);
```

Returns the year, month, day, hour, minutes, and decimal seconds given a floating point day of the year.

All arguments must be numeric. `$n` must be greater than or equal to 1 and less than 366 on non-leap years and 367 on leap years.

NOTE: When `$n` is a decimal number, the results are non-intuitive perhaps. Day 1 is Jan 01 00:00. Day 2 is Jan 02 00:00. Intuitively, you might think of day 1.5 as being 1.5 days after Jan 01 00:00, but this would mean that Day 1.5 was Jan 02 12:00 (which is later than Day 2). The best way to think of this function is a time line starting at 1 and ending at 366 (in a non-leap year). In terms of a delta, think of `$n` as the number of days after Dec 31 00:00 of the previous year.

Date_DaysInYear

```
$days = Date_DaysInYear($y);
```

Returns the number of days in the year (365 or 366)

Date_DaysInMonth

```
$days = Date_DaysInMonth($m,$y);
```

Returns the number of days in the month.

Date_WeekOfYear

```
$wkno = Date_WeekOfYear($m,$d,$y,$first);
```

Figure out week number. `$first` is the first day of the week which is usually 1 (Monday) or 7 (Sunday), but could be any number between 1 and 7 in practice.

NOTE: This routine should only be called in rare cases. Use UnixDate with the `%W`, `%U`, `%J`, `%L` formats instead. This routine returns a week between 0 and 53 which must then be ''fixed'' to get into the ISO−8601 weeks from 1 to 53. A date which returns a week of 0 actually belongs to the last week of the previous year. A date which returns a week of 53 may belong to the first week of the next year.

Date_LeapYear

```
$flag = Date_LeapYear($y);
```

Returns 1 if the argument is a leap year Written by David Muir Sharnoff <muir@idiom.com>

Date_DaySuffix

```
$day = Date_DaySuffix($d);
```

Add ‘st’, ‘nd’, ‘rd’, ‘th’ to a date (i.e. 1st, 22nd, 29th). Works for international dates.

Date_TimeZone

```
$tz = Date_TimeZone;
```

This determines and returns the local time zone. If it is unable to determine the local time zone, the following error occurs:

```
ERROR: Date::Manip unable to determine Time Zone.
```

See the Date::Manip::TZ documentation (DETERMINING THE LOCAL TIME ZONE) for more information.

Date_ConvTZ

```
$date = Date_ConvTZ($date,$from,$to);
```

This converts a date (which MUST be in the format returned by ParseDate) from one time zone to another.

`$from` and `$to` each default to the local time zone. If they are given, they must be any time zone or alias understood by Date::Manip.

If an error occurs, an empty string is returned.

Date_NextWorkDay

```
$date = Date_NextWorkDay($date,$off [,$time]);
```

Finds the day `$off` work days from now. If`$time` is passed in, we must also tak e into account the time of day.

If `$time` is not passed in, day 0 is today (if today is a workday) or the next work day if it isn’t. In any case, the time of day is unaffected.

If `$time` is passed in, day 0 is now (if now is part of a workday) or the start of the very next work day.

Date_PrevWorkDay

```
$date = Date_PrevWorkDay($date,$off [,$time]);
```

Similar to Date_NextWorkDay.

Date_NearestWorkDay

```
$date = Date_NearestWorkDay($date [,$tomorrowfirst]);
```

This looks for the work day nearest to `$date`. If`$date` is a w ork day, it is returned. Otherwise, it will look forward or backwards in time 1 day at a time until a work day is found. If `$tomorrowfirst` is non-zero (or if it is omitted and the config variable TomorrowFirst is non-zero), we look to the future first. Otherwise, we look in the past first. In other words, in a normal week, if `$date` is Wednesday, `$date` is returned. If `$date` is Saturday, Friday is returned. If `$date` is Sunday, Monday is returned. If Wednesday is a holiday, Thursday is returned if `$tomorrowfirst` is non-nil or Tuesday otherwise.

For all of the functions which return a date, the format of the returned date is governed by the Printable config variable. If a date is returned, it is in the local time zone, NOT the time zone the date was parsed in.

SEE ALSO

Date::Manip − main module documentation

LICENSE

AUTHOR

Sullivan Beck (sbeck@cpan.org)