NAME

Date::Manip::DM5 − Date manipulation routines

SYNOPSIS

```
use Date::Manip;

$version = DateManipVersion;

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

$date = ParseDate(\@args);
$date = ParseDate($string);
$date = ParseDate(\$string);

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

$delta = ParseDateDelta(\@args);
$delta = ParseDateDelta($string);
$delta = ParseDateDelta(\$string);

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

$date = Date_SetTime($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);

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

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

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

$date = Date_NextWorkDay($date,$off [,$flag]);

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

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

The above routines all check to make sure that Date_Init is called. If it hasn't been, they will call it automatically. As a result, there is usually no need to call Date_Init explicitly unless you want to change some of the config variables (described below). They also do error checking on the input.

The routines listed below are intended primarily for internal use by other Date::Manip routines. They do little or no error checking, and do not explicitly call Date_Init. Those functions are all done in the main Date::Manip routines above.

Because they are significantly faster than the full Date::Manip routines, they are available for use with a few caveats. Since little or no error checking is done, it is the responsibility of the programmer to ensure that valid data (AND valid dates) are passed to them. Passing invalid data (such as a non-numeric month) or invalid dates (Feb 31) will fail in unpredictable ways (possibly returning erroneous results). Also, since Date_Init is not called by these, it must be called explicitly by the programmer before using these routines.

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. Most (if not all) of the information below can be gotten from UnixDate which is really the way I intended it to be gotten, but there are reasons to use these (these are significantly faster).

```
$day = Date_DayOfWeek($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

Date_Init

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

Normally, it is not necessary to explicitly call Date_Init. The first time any of the other routines are called, Date_Init will be called to set everything up. If for some reason you want to change the configuration of Date::Manip, you can pass the appropriate string or strings into Date_Init to reinitialize things.

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. A list of all configuration variables is given in the section CUSTOMIZING DATE::MANIP below. 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");
```

If Date_Init is called in list context, it will return a list of all config variables and their values suitable for passing in to Date_Init to return Date::Manip to the current state. The only possible problem is that by default, holidays will not be erased, so you may need to prepend the ''EraseHolidays=1'' element to the list.

ParseDate

```
$date = ParseDate(\@args);
$date = ParseDate($string);
$date = ParseDate(\$string);
```

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 real work is done in the ParseDateString routine.

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.

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);
```

This routine is called by ParseDate, but it may also be called directly to save some time (a negligible amount).

NOTE: One of the most frequently asked questions that I have gotten is how to parse seconds since the epoch. ParseDateString cannot simply parse a number as the seconds since the epoch (it conflicts with some ISO-8601 date formats). There are two ways to get this information. First, you can do the following:

```
$secs = ...         # seconds since Jan 1, 1970  00:00:00 GMT
$date = DateCalc("Jan 1, 1970  00:00:00 GMT","+ $secs");
```

Second, you can call it directly as:

```
$date = ParseDateString("epoch $secs");
```

To go backwards, just use the ''%s'' format of UnixDate:

```
$secs = UnixDate($date,"%s");
```

A full date actually includes 2 parts: date and time. A time must include hours and minutes and can optionally include seconds, fractional seconds, an am/pm type string, and a time zone. For example:

```
[at] HH:MN              [Zone]
[at] HH:MN         [am] [Zone]
[at] HH:MN:SS      [am] [Zone]
[at] HH:MN:SS.SSSS [am] [Zone]
[at] HH            am   [Zone]
```

Hours can be written using 1 or 2 digits, but the single digit form may only be used when no ambiguity is introduced (i.e. when it is not immediately preceded by a digit).

A time is usually entered in 24 hour mode, but 12 hour mode can be used as well if AM/PM are entered (AM can be entered as AM or A.M. or other variations depending on the language).

Fractional seconds are also supported in parsing but the fractional part is discarded (with NO rounding occurring).

Time zones always appear immediately after the time. A number of different forms are supported (see the section TIME ZONES below).

Incidentally, the time is removed from the date before the date is parsed, so the time may appear before or after the date, or between any two parts of the date.

Valid date formats include the ISO 8601 formats:

```
YYYYMMDDHHMNSSF...
YYYYMMDDHHMNSS
YYYYMMDDHHMN
YYYYMMDDHH
YY-MMDDHHMNSSF...
YY-MMDDHHMNSS
YY-MMDDHHMN
YY-MMDDHH
YYYYMMDD
YYYYMM
YYYY
YY-MMDD
YY-MM
YY
YYYYwWWD      ex.  1965-W02-2
YYwWWD
YYYYDOY       ex.  1965-045
YYDOY
```

In the above list, YYYY and YY signify 4 or 2 digit years, MM, DD, HH, MN, SS refer to two digit month, day, hour, minute, and second respectively. F... refers to fractional seconds (any number of digits) which will be ignored. In all cases, the date and time parts may be separated by the letter ‘‘T’’ (but this is optional), so
2002-12-10-12:00:00
2002-12-10T12:00:00 are identical.

The last 4 formats can be explained by example: 1965-w02-2 refers to Tuesday (day 2) of the 2nd week of 1965. 1965-045 refers to the 45th day of 1965.

In all cases, parts of the date may be separated by dashes ‘‘-’’. If this is done, 1 or 2 digit forms of MM, DD, etc. may be used. All dashes are optional except for those given in the table above (which MUST be included for that format to be correctly parsed). So 19980820, 1998-0820, 1998-08-20, 1998-8-20, and 199808-20 are all equivalent, but that date may NOT be written as 980820 (it must be written as 98-0820).

NOTE: Even though not allowed in the standard, the time zone for an ISO-8601 date is flexible and may be any of the time zones understood by Date::Manip.

Additional date formats are available which may or may not be common including:

```
MM/DD  **
MM/DD/YY  **
MM/DD/YYYY  **

mmmDD         DDmmm                    mmmYYYY/DD     mmmYYYY
mmmDD/YY      DDmmmYY       DD/YYmmm   YYYYmmmDD      YYYYmmm
mmmDDYYYY     DDmmmYYYY     DDYYYYmmm  YYYY/DDmmm
```

Where mmm refers to the name of a month. All parts of the date can be separated by valid separators (space, ‘‘/’’, or ‘‘.’’). The separator ‘‘−’’ may be used as long as it doesn’t conflict with an ISO 8601 format, but this is discouraged since it is easy to overlook conflicts. For example, the format MM/DD/YY is just fine, but MM-DD-YY does not work since it conflicts with YY-MM-DD. To be safe, if ‘‘−’’ is used as a separator in a non-ISO format, they should be turned into ‘‘/’’ before calling the Date::Manip routines. As with ISO 8601 formats, all separators are optional except for those given as a ‘‘/’’ in the list above.

** Note that with these formats, Americans tend to write month first, but many other countries tend to write day first. The latter behavior can be obtained by setting the config variable DateFormat to something other than ‘‘US’’ (see CUSTOMIZING DATE::MANIP below).

Date separators are treated very flexibly (they are converted to spaces), so the following dates are all equivalent:

```
12/10/1965
12−10 / 1965
12 // 10 −. 1965
```

In some cases, this may actually be TOO flexible, but no attempt is made to trap this.

Years can be entered as 2 or 4 digits, days and months as 1 or 2 digits. Both days and months must include 2 digits whenever they are immediately adjacent to another numeric part of the date or time. Date separators are required if single digit forms of DD or MM are used. If separators are not used, the date will either be unparsable or will get parsed incorrectly.

Miscellaneous other allowed formats are:

which dofw in mmm in YY	‘‘first Sunday in June 1996 at 14:00’’ **
dofw week num YY	‘‘Sunday week 22 1995’’ **
which dofw YY	‘‘22nd Sunday at noon’’ **
dofw which week YY	‘‘Sunday 22nd week in 1996’’ **
next/last dofw	‘‘next Friday at noon’’
next/last week/month	‘‘next month’’
in num days/weeks/months	‘‘in 3 weeks at 12:00’’
num days/weeks/months later	‘‘3 weeks later’’
num days/weeks/months ago	‘‘3 weeks ago’’
dofw in num week	‘‘Friday in 2 weeks’’
in num weeks dofw	‘‘in 2 weeks on Friday’’
dofw num week ago	‘‘Friday 2 weeks ago’’
num week ago dofw	‘‘2 weeks ago Friday’’
last day in mmm in YY	‘‘last day of October’’
dofw	‘‘Friday’’ (Friday of current week)
Nth	‘‘12th’’, ‘‘1st’’ (day of current month)
epoch SECS	seconds since the epoch

(negative values are
supported)

** Note that the formats ‘‘Sunday week 22’’ and ‘‘22nd Sunday’’ give very different behaviors. ‘‘Sunday week 22’’ returns the Sunday of the 22nd week of the year based on how week 1 is defined. ISO 8601 defines week one to contain Jan 4, so ‘‘Sunday week 1’’ might be the first or second Sunday of the current year, or the last Sunday of the previous year. ‘‘22nd Sunday’’ gives the actual 22nd time Sunday occurs in a given year, regardless of the definition of a week.

Note that certain words such as ‘‘in’’, ‘‘at’’, ‘‘of’’, etc. which commonly appear in a date or time are ignored. Also, the year is always optional.

In addition, the following strings are recognized:
today (exactly now OR today at a given time if a time is specified)
now (synonym for today)
yesterday (exactly 24 hours ago unless a time is specified)
tomorrow (exactly 24 hours from now unless a time is specified)
noon (12:00:00)
midnight (00:00:00) Other languages have similar (and in some cases additional) strings.

Some things to note:

All strings are case insensitive. ‘‘December’’ and ‘‘DEceMBer’’ both work.

When a part of the date is not given, defaults are used: year defaults to current year; hours, minutes, seconds to 00.

The year may be entered as 2 or 4 digits. If entered as 2 digits, it will be converted to a 4 digit year. There are several ways to do this based on the value of the YYtoYYYY variable (described below). The default behavior it to force the 2 digit year to be in the 100 year period CurrYear−89 to CurrYear+10. So in 1996, the range is [1907 to 2006], and the 2 digit year 05 would refer to 2005 but 07 would refer to 1907. See CUSTOMIZING DATE::MANIP below for information on YYtoYYYY for other methods.

Dates are always checked to make sure they are valid.

In all of the formats, the day of week (‘‘Friday’’) can be entered anywhere in the date and it will be checked for accuracy. In other words,
‘‘Tue Jul 16 1996 13:17:00’’ will work but
‘‘Jul 16 1996 Wednesday 13:17:00’’ will not (because Jul 16, 1996 is Tuesday, not Wednesday). Note that depending on where the weekday comes, it may give unexpected results when used in array context (with ParseDate). For example, the date (‘‘Jun’’,‘‘25’’,‘‘Sun’’,‘‘1990’’) would return June 25 of the current year since Jun 25, 1990 is not Sunday.

The times ‘‘12:00 am’’, ‘‘12:00 pm’’, and ‘‘midnight’’ are not well defined. For good or bad, I use the following convention in Date::Manip:
midnight = 12:00am = 00:00:00
noon = 12:00pm = 12:00:00 and the day goes from 00:00:00 to 23:59:59. In other words, midnight is the beginning of a day rather than the end of one. The time 24:00:00 is also allowed (though it is automatically transformed to 00:00:00 of the following day).

The format of the date returned is YYYYMMDDHH:MM:SS. The advantage of this time format is that two times can be compared using simple string comparisons to find out which is later. Also, it is readily understood by a human. Alternate forms can be used if that is more convenient. See Date_Init below and the config variable Internal.

NOTE: The format for the date is going to change at some point in the future to YYYYMMDDHH:MN:SS+HHMN*FLAGS. In order to maintain compatibility, you should use UnixDate to extract information from a date, and Date_Cmp to compare two dates. The simple string comparison will only work for dates in the same time zone.

UnixDate

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

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.

`$date` may be any string that can be parsed by ParseDateString.

The format options are:

```
Year
    %y     year                     - 00 to 99
    %Y     year                     - 0001 to 9999
Month, Week
    %m     month of year            - 01 to 12
    %f     month of year            - " 1" to "12"
    %b,%h  month abbreviation       - Jan to Dec
    %B     month name               - January to December
Day
    %j     day of the year          - 001 to 366
    %d     day of month             - 01 to 31

    %e     day of month             - " 1" to "31"
    %v     weekday abbreviation     - " S"," M"," T"," W","Th"," F","Sa"
    %a     weekday abbreviation     - Sun to Sat
    %A     weekday name             - Sunday to Saturday
    %w     day of week              - 1 (Monday) to 7 (Sunday)
    %E     day of month with suffix - 1st, 2nd, 3rd...
Hour
    %H     hour                     - 00 to 23
    %k     hour                     - " 0" to "23"
    %i     hour                     - " 1" to "12"
    %I     hour                     - 01 to 12
    %p     AM or PM
Minute, Second, Time zone
    %M     minute                   - 00 to 59
    %S     second                   - 00 to 59
    %Z     time zone                - "EDT"
    %z     time zone as GMT offset  - "+0100"
Epoch (see NOTE 3 below)
    %s     seconds from 1/1/1970 GMT- negative if before 1/1/1970
    %o     seconds from Jan 1, 1970
           in the current time zone
Date, Time
    %c     %a %b %e %H:%M:%S %Y     - Fri Apr 28 17:23:15 1995
    %C,%u  %a %b %e %H:%M:%S %z %Y  - Fri Apr 28 17:25:57 EDT 1995
    %g     %a, %d %b %Y %H:%M:%S %z - Fri, 28 Apr 1995 17:23:15 EDT
    %D     %m/%d/%y                 - 04/28/95
    %x     %m/%d/%y or %d/%m/%y     - 04/28/95 or 28/04/28
                                      (Depends on DateFormat variable)
    %l     date in ls(1) format (see NOTE 1 below)
             %b %e $H:$M            - Apr 28 17:23  (if within 6 months)
             %b %e  %Y              - Apr 28  1993  (otherwise)
    %r     %I:%M:%S %p              - 05:39:55 PM
```

```
    %R      %H:%M                       - 17:40
    %T,%X   %H:%M:%S                    - 17:40:58
    %V      %m%d%H%M%y                  - 0428174095
    %Q      %Y%m%d                      - 19961025
    %q      %Y%m%d%H%M%S                - 19961025174058
    %P      %Y%m%d%H%M%S                - 1996102517:40:58
    %O      %Y-%m-%dT%H:%M:%S           - 1996-10-25T17:40:58
    %F      %A, %B %e, %Y               - Sunday, January  1, 1996
    %K      %Y-%j                       - 1997-045
Special Year/Week formats (see NOTE 2 below)
    %G      year, Monday as first
            day of week                 - 0001 to 9999
    %W      week of year, Monday
            as first day of week        - 01 to 53
    %L      year, Sunday as first
            day of week                 - 0001 to 9999
    %U      week of year, Sunday
            as first day of week        - 01 to 53
    %J      %G-W%W-%w                   - 1997-W02-2
Other formats
    %n      insert a newline character
    %t      insert a tab character
    %%      insert a `%' character
    %+      insert a `+' character
The following formats are currently unused but may be used in the future:
    N 1234567890 !@#$^&*()_|-=\`[];',./~{}:<>?
They currently insert the character following the %, but may (and probably
will) change in the future as new formats are added.
```

If a lone percent is the final character in a format, it is ignored.

The formats used in this routine were originally based on date.pl (version 3.2) by Terry McGonigal, as well as a couple taken from different versions of the Solaris **date**(1) command. Also, several have been added which are unique to Date::Manip.

NOTE 1:

The ls format (%l) applies to date within the past OR future 6 months!

NOTE 2:

The `%U`, `%W`, `%L`, `%G`, and `%J` formats are used to support the ISO-8601 format: YYYY-wWW-D. In this format, a date is written as a year, the week of the year, and the day of the week. Technically, the week may be considered to start on any day of the week, but Sunday and Monday are the both common choices, so both are supported.

The `%W` and `%G` formats return the week-of-year and the year treating weeks as starting on Monday.

The `%U` and `%L` formats return the week-of-year and the year treating weeks as starting on Sunday.

Most of the time, the `%L` and `%G` formats returns the same value as the `%Y` format, but there is a problem with days occurring in the first or last week of the year.

The ISO-8601 representation of Jan 1, 1993 written in the YYYY-wWW-D format is actually 1992-W53-5. In other words, Jan 1 is treated as being in the last week of the preceding year. Depending on the year, days in the first week of a year may belong to the previous year, and days in the final week of a year may belong to the next year. The week is assigned to the year which has most of the days. For example, if the week starts on Sunday, then the last week of 2003 is 2003-12-28 to 2004-01-03. This week is assigned to 2003 since 4 of the days in it are in 2003 and only 3 of them

are in 2004. The first week of 2004 starts on 2004−01−04.

The %U and %W formats return a week-of-year number from 01 to 53. %L and %G return the corresponding year, and to get this type of information, you should always use the (%W,%G) combination or (%U,%L) combination. %Y should not be used as it will yield incorrect results.

%J returns the full ISO−8601 format (%G−W%W−%w).

NOTE 3:

The %s and %o formats return negative values if the date is before the start of the epoch. Other Unix utilities would return an error, or a zero, so if you are going to use Date::Manip in conjunction with these, be sure to check for a negative value.

ParseDateDelta

```
$delta = ParseDateDelta(\@args);
$delta = ParseDateDelta($string);
$delta = ParseDateDelta(\$string);
```

This takes an array and shifts a valid delta date (an amount of time) from the array. Recognized deltas are of the form:

```
+Yy +Mm +Ww +Dd +Hh +MNmn +Ss
   examples:
     +4 hours +3mn −2second
     + 4 hr 3 minutes −2
     4 hour + 3 min −2 s
+Y:+M:+W:+D:+H:+MN:+S
   examples:
     0:0:0:0:4:3:−2
     +4:3:−2
mixed format
   examples:
     4 hour 3:−2
```

A field in the format +Yy is a sign, a number, and a string specifying the type of field. The sign is ‘‘+’’, ‘‘−’’, or absent (defaults to the next larger element). The valid strings specifying the field type are:

```
y:  y, yr, year, years
m:  m, mon, month, months
w:  w, wk, ws, wks, week, weeks
d:  d, day, days
h:  h, hr, hour, hours
mn: mn, min, minute, minutes
s:  s, sec, second, seconds
```

Also, the ‘‘s’’ string may be omitted. The sign, number, and string may all be separated from each other by any number of whitespace.

In the date, all fields must be given in the order: Y M W D H MN S. Any number of them may be omitted provided the rest remain in the correct order. In the 2nd (colon) format, from 2 to 7 of the fields may be given. For example +D:+H:+MN:+S may be given to specify only four of the fields. In any case, both the MN and S field may be present. No spaces may be present in the colon format.

Deltas may also be given as a combination of the two formats. For example, the following is valid: +Yy +D:+H:+MN:+S. Again, all fields must be given in the correct order.

The word ‘‘in’’ may be given (prepended in English) to the delta (‘‘in 5 years’’) and the word ‘‘ago’’ may be given (appended in English) (‘‘6 months ago’’). The ‘‘in’’ is completely ignored. The ‘‘ago’’ has the affect of reversing all signs that appear in front of the components of the delta. I.e. ‘‘−12 yr 6 mon ago’’ is identical to ‘‘+12yr +6mon’’ (don’t forget that there is an implied minus sign in front of

the 6 because when no sign is explicitly given, it carries the previously entered sign).

One thing is worth noting. The year/month and day/hour/min/sec parts are returned in a ‘‘normalized’’ form. That is, the signs are adjusted so as to be all positive or all negative. For example, ‘‘+ 2 day − 2hour’’ does not return ‘‘0:0:0:2:−2:0:0’’. It returns ‘‘+0:0:0:1:22:0:0’’ (1 day 22 hours which is equivalent). I find (and I think most others agree) that this is a more useful form.

Since the year/month and day/hour/min/sec parts must be normalized separately there is the possibility that the sign of the two parts will be different. So, the delta ‘‘+ 2years −10 months − 2 days + 2 hours’’ produces the delta ‘‘+1:2:−0:1:22:0:0’’.

It is possible to include a sign for all elements that is output. See the configuration variable DeltaSigns below.

NOTE: The internal format of the delta changed in version 5.30 from Y:M:D:H:MN:S to Y:M:W:D:H:MN:S . Also, it is going to change again at some point in the future to Y:M:W:D:H:MN:S*FLAGS . Use the routine Delta_Format to extract information rather than parsing it yourself.

Delta_Format

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

This is similar to the UnixDate routine except that it extracts information from a delta. Unlike the UnixDate routine, most of the formats are 2 characters instead of 1.

Formats currently understood are:

```
%Xv     : the value of the field named X
%Xd     : the value of the field X, and all smaller fields, expressed in
          units of X
%Xh     : the value of field X, and all larger fields, expressed in units
          of X
%Xt     : the value of all fields expressed in units of X

X is one of y,M,w,d,h,m,s (case sensitive).

%%      : returns a "%"
```

So, the format ‘‘%hd’’ means the values of H, MN, and S expressed in hours. So for the delta ‘‘0:0:0:0:2:30:0’’, this format returns 2.5.

Delta_Format can operate in two modes: exact and approximate. The exact mode is done by default. Approximate mode can be done by passing in the string ‘‘approx’’ as the 2nd argument.

In exact mode, Delta_Format only understands ‘‘exact’’ relationships. This means that there can be no mixing of the Y/M and W/D/H/MN/S segments because the relationship because, depending on when the delta occurs, there is no exact relation between the number of years or months and the number of days.

The two sections are treated completely separate from each other. So, the delta ‘‘1:6:1:2:12:0:0’’ would return the following values:

```
%yt = 1.5 (1 year, 6 months)
%Mt = 18

%dt = 9.5 (1 week, 2 days, 12 hours)
```

In approximate mode, the relationship of 1 year = 365.25 days is applied (with 1 month equal to 1/12 of a year exactly). So the delta ‘‘1:6:1:2:12:0:0’’ would return the following values:

```
%dt = 557.375 (1.5 years of 365.25 days + 9.5 days)
```

If `$dec` is non-zero, the `%Xd` and `%Xt` values are formatted to contain `$dec` decimal places.

ParseRecur

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

A recurrence refers to a recurring event, and more specifically, an event which occurs on a regular basis. A fully specified recurring event may requires up to four pieces of information.

First, it requires a description of the frequency of the event. Examples include ‘‘the first of every month’’, ‘‘every other day’’, ‘‘the 4th Thursday of each month at 2:00 PM’’, and ‘‘every 2 hours and 30 minutes’’.

Second, it may require a base date to work from. This piece of information is not required for every type of recurrence. For example, if the frequency is ‘‘the first of every month’’, no base date is required. All the information about when the event occurs is included in the frequency description. If the frequency were ‘‘every other day’’ though, you need to know at least one day on which the event occurred.

Third, the recurring event may have a range (a starting and ending date).

Fourth, there may be some flags included which modify the behavior of the above information.

The fully specified recurrence is written as these 5 pieces of information (both a start and end date) as an asterisk separated list:

```
freq*flags*base*date0*date1
```

Here, base, date0, and date1 are any strings (which must not contain any asterisks) which can be parsed by ParseDate. flags is a comma separated list of flags (described below), and freq is a string describing the frequency of the recurring event.

The syntax of the frequency description is a colon separated list of the format Y:M:W:D:H:MN:S (which stand for year, month, week, etc.). One (and only one) of the colons may optionally be replaced by an asterisk, or an asterisk may be prepended to the string. For example, the following are all valid frequency descriptions:

```
1:2:3:4:5:6:7
1:2*3:4:5:6:7
*1:2:3:4:5:6:7
```

But the following are NOT valid because they contain 2 or more asterisks:

```
1:2*3:4:5*6:7
1*2*3:4:5*6:7
*1:2:3:4:5:6*7
```

If an asterisk is included, values to the left of it refer to the number of times that time interval occurs between recurring events. For example, if the first part of the recurrence is:

```
1:2*
```

this says that the recurring event occurs approximately every 1 year and 2 months. I say approximately, because elements to the right of the asterisk, as well as any flags included in the recurrence will affect when the actual events occur.

If no asterisks are included, then the entire recurrence is of this form. For example,

```
0:0:0:1:12:0:0
```

refers to an event that occurs every 1 day, 12 hours.

Values that occur after an asterisk refer to a specific value for that type of time element (i.e. exactly as

it would appear on a calendar or a clock). For example, if the recurrence ends with:

```
*12:0:0
```

then the recurring event occurs at 12:00:00 (noon).

For example:

```
0:0:2:1:0:0:0        every 2 weeks and 1 day
0:0:0:0:5:30:0       every 5 hours and 30 minutes
0:0:0:2*12:30:0      every 2 days at 12:30 (each day)
```

Values to the right of the asterisk can be listed a single values, ranges (2 numbers separated by a dash "−"), or a comma separated list of values or ranges. In most cases, negative values are appropriate for the week or day values. −1 stands for the last possible value, −2 for the second to the last, etc.

Some examples are:

```
0:0:0:1*2,4,6:0:0    every day at at 2:00, 4:00, and 6:00
0:0:0:2*12−13:0,30:0 every other day at 12:00, 12:30, 13:00,
                     and 13:30
0:1:0*−1:0:0:0       the last day of every month
*1990−1995:12:0:1:0:0:0
                     Dec 1 in 1990 through 1995
```

There is no way to express the following with a single recurrence:

```
every day at 12:30 and 1:00
```

You have to use two recurrences to do this.

When a non-zero day element occurs to the right of the asterisk, it can take on multiple meanings, depending on the value of the month and week elements. It can refer to the day of the week, day of the month, or day of the year. Similarly, if a non-zero week element occurs to the right of the asterisk, it actually refers to the nth time a certain day of the week occurs, either in the month or in the year.

If the week element is non-zero and the day element is non-zero (and to the right of the asterisk), the day element refers to the day of the week. It can be any value from 1 to 7 (negative values −1 to −7 are also allowed). If you use the ISO 8601 convention, the first day of the week is Monday (though Date::Manip can use any day as the start of the week by setting the FirstDay config variable). So, assuming that you are using the ISO 8601 convention, the following examples illustrate day-of-week recurrences:

```
0:1*4:2:0:0:0        4th Tuesday (day 2) of every month
0:1*−1:2:0:0:0       last Tuesday of every month
0:0:3*2:0:0:0        every 3rd Tuesday (every 3 weeks
                     on 2nd day of week)
1:0*12:2:0:0:0       the 12th Tuesday of each year
```

If the week element is non-zero, and the day element is zero, the day defaults to 1 (i.e. the first day of the week).

```
0:1*2:0:0:0:0        the 2nd occurrence of FirstDay
                     in the year (typically Monday)
0:1*2:1:0:0:0        the same
```

If the week element is zero and the month element is non-zero, the day value is the day of the month (it can be from 1 to 31 or −1 to −31 counting from the end of the month). If a value of 0 is given, it defaults to 1.

```
3*1:0:2:12:0:0       every 3 years on Jan 2 at noon
0:1*0:2:12,14:0:0    2nd of every month at 12:00 and 14:00
0:1:0*-2:0:0:0       2nd to last day of every month
```

If the day given refers to the 29th, 30th, or 31st, in a month that does not have that number of days, it is ignored. For example, if you ask for the 31st of every month, it will return dates in Jan, Mar, May, Jul, etc. Months with fewer than 31 days will be ignored.

If both the month and week elements are zero, and the year element is non-zero, the day value is the day of the year (1 to 365 or 366 — or the negative numbers to count backwards from the end of the year).

```
1:0:0*45:0:0:0       45th day of every year
```

Specifying a day that doesn't occur in that year silently ignores that year. The only result of this is that specifying +366 or −366 will ignore all years except leap years.

I realize that this looks a bit cryptic, but after a discussion on the CALENDAR mailing list, it appeared like there was no concise, flexible notation for handling recurring events. ISO 8601 notations were very bulky and lacked the flexibility I wanted. As a result, I developed this notation (based on crontab formats, but with much more flexibility) which fits in well with this module. Even better, it is able to express every type of recurring event I could think of that is used in common life in (what I believe to be) a very concise and elegant way.

If ParseRecur is called in scalar context, it returns a string containing a fully specified recurrence (or as much of it as can be determined with unspecified fields left blank). In list context, it returns a list of all dates referred to by a recurrence if enough information is given in the recurrence. All dates returned are in the range:

```
date0 <= date < date1
```

The argument `$string` can contain any of the parts of a full recurrence. For example:

```
freq
freq*flags
freq**base*date0*date1
```

The only part which is required is the frequency description. Any values contained in `$string` are overridden or modified by values passed in as parameters to ParseRecur.

NOTE: If a recurrence has a date0 and date1 in it AND a date0 and date1 are passed in to the function, both sets of criteria apply. If flags are passed in, they override any flags in the recurrence UNLESS the flags passed in start with a plus (+) character in which case they are appended to the flags in the recurrence.

NOTE: Base dates are only used with some types of recurrences. For example,

```
0:0:3*2:0:0:0        every 3rd Tuesday
```

requires a base date. If a base date is specified which doesn't match the criteria (for example, if a base date falling on Monday were passed in with this recurrence), the base date is moved forward to the first relevant date.

Other dates do not require a base date. For example:

```
0:0*3:2:0:0:0        third Tuesday of every month
```

A recurrence written in the above format does NOT provide default values for base, date0, or date1. They must be specified in order to get a list of dates.

A base date is not used entirely. It is only used to provide the parts necessary for the left part of a recurrence. For example, the recurrence:

```
  1:3*0:4:0:0:0        every 1 year, 3 months on the 4th day of the month
```

would only use the year and month of the base date.

There are a small handful of English strings which can be parsed in place of a numerical recur description. These include:

```
every 2nd day [in 1997]
every 2nd day in June [1997]
2nd day of every month [in 1997]
2nd Tuesday of every month [in 1997]
last Tuesday of every month [in 1997]
every Tuesday [in 1997]
every 2nd Tuesday [in 1997]
every 2nd Tuesday in June [1997]
```

Each of these set base, date0, and date1 to a default value (the current year with Jan 1 being the base date is the default if the year and month are missing).

The following flags (case insensitive) are understood:

```
PDn   : n is 1-7.  Means the previous day n not counting today
PTn   : n is 1-7.  Means the previous day n counting today
NDn   : n is 1-7.  Means the next day n not counting today
NTn   : n is 1-7.  Means the next day n counting today

FDn   : n is any number.  Means step forward n days.
BDn   : n is any number.  Means step backward n days.
FWn   : n is any number.  Means step forward n workdays.
BWn   : n is any number.  Means step backward n workdays.

CWD   : the closest work day (using the TomorrowFirst config variable).
CWN   : the closest work day (looking forward first).
CWP   : the closest work day (looking backward first).

NWD   : next work day counting today
PWD   : previous work day counting today
DWD   : next/previous work day (TomorrowFirst config) counting today

EASTER: select easter for this year (the M, W, D fields are ignored
        in the recur).
```

CWD, CWN, and CWP will usually return the same value, but if you are starting at the middle day of a 3-day weekend (for example), it will return either the first work day of the following week, or the last work day of the previous week depending on whether it looks forward or backward first.

All flags are applied AFTER the recurrence dates are calculated, and they may move a date outside of the date0 to date1 range. No check is made for this.

The workday flags do not act exactly the same as a business mode calculation. For example, a date that is Saturday with a FW1 steps forward to the first workday (i.e. Monday).

Date_Cmp

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

This takes two dates and compares them. Almost all dates can be compared using the Perl ''cmp'' command. The only time this will not work is when comparing dates in different time zones. This routine will take that into account.

NOTE: This routine currently does little more than use ''cmp'', but once the internal format for storing dates is in place (where time zone information is kept as part of the date), this routine will become

more important. You should use this routine in preparation for that version.

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 types of dates can be interpreted as deltas (and vice versa), it is a good idea to pass the input through ParseDate or ParseDateDelta as appropriate. 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.

Note that in many cases, it is somewhat ambiguous what the delta actually refers to. Although it is ALWAYS known how many months in a year, hours in a day, etc., it is NOT known (in the generals case) how many days are in a month. As a result, the part of the delta containing month/year and the part with sec/min/hr/day must be treated separately. For example, ''Mar 31, 12:00:00'' plus a delta of 1month 2days would yield ''May 2 12:00:00''. The year/month is first handled while keeping the same date. Mar 31 plus one month is Apr 31 (but since Apr only has 30 days, it becomes Apr 30). Apr 30 + 2 days is May 2. As a result, in the case where two dates are entered, the resulting delta can take on two different forms. By default ($mode=0), an absolutely correct delta (ignoring daylight saving time) is returned in weeks, days, hours, minutes, and seconds.

If `$mode` is 1, the math is done using an approximate mode where a delta is returned using years and months as well. The year and month part is calculated first followed by the rest. For example, the two dates ''Mar 12 1995'' and ''Apr 13 1995'' would have an exact delta of ''31 days'' but in the approximate mode, it would be returned as ''1 month 1 day''. Also, ''Mar 31'' and ''Apr 30'' would have deltas of ''30 days'' or ''1 month'' (since Apr 31 doesn't exist, it drops down to Apr 30). Approximate mode is a more human way of looking at things (you'd say 1 month and 2 days more often then 33 days), but it is less meaningful in terms of absolute time. In approximate mode `$d1` and `$d2` must be dates. If either or both is a delta, the calculation is done in exact mode.

If `$mode` is 2, a business mode is used. That is, the calculation is done using business days, ignoring holidays, weekends, etc. In order to correctly use this mode, a config file must exist which contains the section defining holidays (see documentation on the config file below). The config file can also define the work week and the hours of the work day, so it is possible to have different config files for different businesses.

For example, if a config file defines the workday as 08:00 to 18:00, a work week consisting of Mon-Sat, and the standard (American) holidays, then from Tuesday at 12:00 to the following Monday at 14:00 is 5 days and 2 hours. If the ''end'' of the day is reached in a calculation, it automatically switches to the next day. So, Tuesday at 12:00 plus 6 hours is Wednesday at 08:00 (provided Wed is not a holiday). Also, a date that is not during a workday automatically becomes the start of the next workday. So, Sunday 12:00 and Monday at 03:00 both automatically becomes Monday at 08:00 (provided Monday is not a holiday). In business mode, any combination of date and delta may be entered, but a delta should not contain a year or month field (weeks are fine though).

See Date::Manip::Calc for some additional comments about business mode calculations.

Note that a business week is treated the same as an exact week (i.e. from Tuesday to Tuesday, regardless of holidays). Because this means that the relationship between days and weeks is NOT unambiguous, when a delta is produced from two dates, it will be in terms of d/h/mn/s (i.e. no week field).

If `$mode` is 3 (which only applies when two dates are passed in), an exact business mode is used. In this case, it returns a delta as an exact number of business days/hours/etc. between the two. Weeks, months, and years are ignored.

Any other non-nil value of `$mode` is treated as `$mode`=1 (approximate mode).

The mode can be automatically set in the dates/deltas passed by including a key word somewhere in it. For example, in English, if the word ‘‘approximately’’ is found in either of the date/delta arguments, approximate mode is forced. Likewise, if the word ‘‘business’’ or ‘‘exactly’’ appears, business/exact mode is forced (and `$mode` is ignored). So, the two following are equivalent:

```
$date = DateCalc("today","+ 2 business days",\$err);
$date = DateCalc("today","+ 2 days",\$err,2);
```

Note that if the keyword method is used instead of passing in `$mode`, it is important that the keyword actually appear in the argument passed in to DateCalc. The following will NOT work:

```
$delta = ParseDateDelta("+ 2 business days");
$today = ParseDate("today");
$date = DateCalc($today,$delta,\$err);
```

because the mode keyword is removed from a date/delta by the parse routines, and the mode is reset each time a parse routine is called. Since DateCalc parses both of its arguments, whatever mode was previously set is ignored.

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 is returned if the date is outside the years 1000 to 9999 This argument is optional, but if included, it must come before `$mode`.

Nothing is returned if an error occurs.

When a delta is returned, the signs such that it is strictly positive or strictly negative (‘‘1 day − 2 hours’’ would never be returned for example). The only time when this cannot be enforced is when two deltas with a year/month component are entered. In this case, only the signs on the day/hour/min/sec part are standardized.

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");
```

Note that in this routine (as well as the other routines below which use a time argument), no real parsing is done on the times. As a result,

```
$date = Date_SetTime($date,"13:30");
```

works, but

```
$date = Date_SetTime($date,"1:30 PM");
```

doesn’t.

Date_SetDateField

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

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.

If `$nocheck` is non-zero, no check is made as to the validity of the date.

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.

If `$dow` is defined, the previous occurrence of the day of week is returned. `$dow` may either be a string (such as ‘‘Fri’’ or ‘‘Friday’’) or a number (between 1 and 7). The date of the previous `$dow` is returned.

If `$date` falls on the day of week given by `$dow`, the date returned depends on `$curr`. If`$curr` is 0, the date returned is a week before `$date`. If`$curr` is 1, the date returned is the same as `$date`. If `$curr` is 2, the date returned (including the time information) is required to be before `$date`.

If a time is passed in (either as separate hours, minutes, seconds or as a time in HH:MM:SS or HH:MM format), the time on this date is set to it. The following examples should illustrate the use of Date_GetPrev:

```
date                    dow    curr  time            returns
Fri Nov 22 18:15:00     Thu    any   12:30           Thu Nov 21 12:30:00
Fri Nov 22 18:15:00     Fri    0     12:30           Fri Nov 15 12:30:00
Fri Nov 22 18:15:00     Fri    1/2   12:30           Fri Nov 22 12:30:00

Fri Nov 22 18:15:00     Fri    1     18:30           Fri Nov 22 18:30:00
Fri Nov 22 18:15:00     Fri    2     18:30           Fri Nov 15 18:30:00
```

If `$dow` is undefined, then a time must be entered, and the date returned is the previous occurrence of this time. If `$curr` is non-zero, the current time is returned if it matches the criteria passed in. In other words, the time returned is the last time that a digital clock (in 24 hour mode) would have displayed the time you passed in. If you define hours, minutes and seconds default to 0 and you might jump back as much as an entire day. If hours are undefined, you are looking for the last time the minutes/seconds appeared on the digital clock, so at most, the time will jump back one hour.

```
date                curr  hr     min    sec      returns
Nov 22 18:15:00     0/1   18     undef  undef    Nov 22 18:00:00
Nov 22 18:15:00     0/1   18     30     0        Nov 21 18:30:00
Nov 22 18:15:00     0     18     15     undef    Nov 21 18:15:00
Nov 22 18:15:00     1     18     15     undef    Nov 22 18:15:00
Nov 22 18:15:00     0     undef  15     undef    Nov 22 17:15:00
Nov 22 18:15:00     1     undef  15     undef    Nov 22 18:15:00
```

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_IsHoliday

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

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

Events_List

```
$ref = Events_List($date);
$ref = Events_List($date ,0      [,$flag]);
$ref = Events_List($date0,$date1 [,$flag]);
```

This returns a list of events. Events are defined in the Events section of the config file (discussed below).

In the first form (a single argument), `$date` is any string containing a date. A list of events active at that precise time will be returned. The format is similar to when `$flag`=0, except only a single time will be returned.

In all other cases, a range of times will be used. If the 2nd argument evaluates to 0, the range of times will be the 24 hour period from midnight to midnight containing `$date`. Otherwise, the range is given by the two dates.

The value of `$flag` determines the format of the information that is returned.

With `$flag`=0, the events are returned as a reference to a list of the form:

```
[ date, [ list_of_events ], date, [ list_of_events ], ... ]
```

For example, if the following events are defined (using the syntax discussed below in the description of the Event section of the config file):

```
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
```

might result in the following output:

```
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 ] ]
```

Much more complicated events can be defined using recurrences.

When `$flag` is non-zero, the format of the output is changed. If `$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);
 => { Winter => +0:0:1:0:0:0:0,
      Spring => +0:0:2:5:0:0:0,
      Event3 => +0:0:0:0:1:0:0 }
```

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

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

The hash contains one element for each combination of events.

Date_DayOfWeek

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

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

All arguments must be numeric.

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).

All arguments must be numeric.

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). If CurrTZ is ''IGNORE'', the number will be identical to Date_SecsSince1970 (i.e. the date given will be treated as being in GMT).

All arguments must be numeric.

Date_DaysSince1BC

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

Returns the number of days since Dec 31, 1BC. This includes the year 0000.

All arguments must be numeric.

Date_DayOfYear

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

Returns the day of the year (001 to 366)

All arguments must be numeric.

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.

All arguments must be numeric.

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 TIME ZONES section below for more information.

Date_ConvTZ

```
$date = Date_ConvTZ($date);
$date = Date_ConvTZ($date,$from);
$date = Date_ConvTZ($date,"",$to [,$errlev]);
$date = Date_ConvTZ($date,$from,$to [,$errlev]);
```

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

If it is called with no arguments, the date is converted from the local time zone to the time zone specified by the config variable ConvTZ (see documentation on ConvTZ below). If ConvTZ is set to ''IGNORE'', no conversion is done.

If called with `$from` but no `$to`, the time zone is converted from the time zone in `$from` to ConvTZ (of TZ if ConvTZ is not set). Again, no conversion is done if ConvTZ is set to ''IGNORE''.

If called with `$to` but no `$from`, `$from` defaults to ConvTZ (if set) or the local time zone otherwise. Although this does not seem immediately obvious, it actually makes sense. By default, all dates that are parsed are converted to ConvTZ, so most of the dates being worked with will be stored in that time zone.

If Date_ConvTZ is called with both `$from` and `$to`, the date is converted from the time zone `$from` to `$to`.

NOTE: As in all other cases, the $date returned from Date_ConvTZ has no time zone information included as part of it, so calling UnixDate with the ‘‘%z’’ format will return the time zone that Date::Manip is working in (usually the local time zone).

Example: To convert 2/2/96 noon PST to CST (regardless of what time zone you are in, do the following:

```
$date = ParseDate("2/2/96 noon");
$date = Date_ConvTZ($date,"PST","CST");
```

Both time zones MUST be in one of the formats listed below in the section TIME ZONES.

If an error occurs, $errlev determines what happens:

```
0   : the program dies
1   : a warning is produced and nothing is returned
2   : the function silently returns nothing
```

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.

Date_NextWorkDay

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

Finds the day $off work days from now. If$flag is non-zero, we must also tak e into account the time of day.

If $flag is zero, 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 $flag is non-zero, 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,$off [,$flag]);
```

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.

DateManipVersion

```
$version = DateManipVersion;
```

Returns the version of Date::Manip.

TIME ZONES

With the release of Date::Manip 6.00, time zones and daylight saving time are now fully supported in Date::Manip. 6.00 uses information from several standards (most importantly the Olson zoneinfo database) to get a list of all known time zones.

Unfortunately, 6.00 requires a newer version of perl, so I will continue to support the 5.xx release for a while. However, the way I will support time zones in 5.xx has changed. Previously, new time zones would be added on request. That is no longer the case. Time zones for 5.xx are now generated automatically from those available in 6.00.

The following time zone names are currently understood (and can be used in parsing dates). These are zones defined in RFC 822.

```
Universal:  GMT, UT
US zones :  EST, EDT, CST, CDT, MST, MDT, PST, PDT
Military :  A to Z (except J)
Other    :  +HHMM or -HHMM
ISO 8601 :  +HH:MM, +HH, -HH:MM, -HH
```

In addition, the following time zone abbreviations are also accepted. These do not come from a standard, but were included in previous releases of Date::Manip 5.xx and are preserved here for backward compatibility:

```
IDLW    -1200    International Date Line West
NT      -1100    Nome
SAT     -0400    Chile
CLDT    -0300    Chile Daylight
AT      -0200    Azores
MEWT    +0100    Middle European Winter
MEZ     +0100    Middle European
FWT     +0100    French Winter
GB      +0100    GMT with daylight savings
SWT     +0100    Swedish Winter
MESZ    +0200    Middle European Summer
FST     +0200    French Summer
METDST  +0200    An alias for MEST used by HP-UX
EETDST  +0300    An alias for eest used by HP-UX
EETEDT  +0300    Eastern Europe, USSR Zone 1
BT      +0300    Baghdad, USSR Zone 2
IT      +0330    Iran
ZP4     +0400    USSR Zone 3
ZP5     +0500    USSR Zone 4
IST     +0530    Indian Standard
ZP6     +0600    USSR Zone 5
AWST    +0800    Australian Western Standard
ROK     +0900    Republic of Korea
AEST    +1000    Australian Eastern Standard
ACDT    +1030    Australian Central Daylight
CADT    +1030    Central Australian Daylight
AEDT    +1100    Australian Eastern Daylight
EADT    +1100    Eastern Australian Daylight
NZT     +1200    New Zealand
IDLE    +1200    International Date Line East
```

All other time zone abbreviations come from the standards. In many cases, an abbreviation may be used for multiple time zones. For example, NST stands for Newfoundland Standard -0330 and North Sumatra +0630. In these cases, only 1 of the two is available. I have tried to use the most recent definition, and of those (if multiple time zones use the abbreviation), the most commonly used. I don't claim that I'm correct in all cases, but I've done the best I could.

The list of abbreviations available is documented in the Date::Manip::DM5abbrevs document.

Date::Manip must be able to determine the time zone the user is in. It does this by looking in the following places:

```
$Date::Manip::TZ (set with Date_Init or in Manip.pm)
$ENV{TZ}
the Unix `date` command (if available)
$main::TZ
/etc/TIMEZONE
/etc/time zone
```

At least one of these should contain a time zone in one of the supported forms. If none do by default, the TZ variable must be set with Date_Init.

The time zone may be in the STD#DST format (in which case both abbreviations must be in the table above) or any of the formats described above. The STD#DST format is NOT available when parsing a date however. The following forms are also available and are treated similar to the STD#DST forms:

```
US/Pacific
US/Mountain
US/Central
US/Eastern
Canada/Pacific
Canada/Mountain
Canada/Central
Canada/Eastern
```

CUSTOMIZING DATE::MANIP

There are a number of variables which can be used to customize the way Date::Manip behaves. There are also several ways to set these variables.

At the top of the Manip.pm file, there is a section which contains all customization variables. These provide the default values.

These can be overridden in a global config file if one is present (this file is optional). If the GlobalCnf variable is set in the Manip.pm file, it contains the full path to a config file. If the file exists, its values will override those set in the Manip.pm file. A sample config file is included with the Date::Manip distribution. Modify it as appropriate and copy it to some appropriate directory and set the GlobalCnf variable in the Manip.pm file.

Each user can have a personal config file which is of the same form as the global config file. The variables PersonalCnf and PersonalCnfPath set the name and search path for the personal config file. This file is also optional. If present, it overrides any values set in the global file.

NOTE: if you use business mode calculations, you must have a config file (either global or personal) since this is the only place where you can define holidays.

Finally, any variables passed in through Date_Init override all other values.

A config file can be composed of several sections. The first section sets configuration variables. Lines in this section are of the form:

```
VARIABLE = VALUE
```

For example, to make the default language French, include the line:

```
Language = French
```

Only variables described below may be used. Blank lines and lines beginning with a pound sign (#) are ignored. All spaces are optional and strings are case insensitive.

A line which starts with an asterisk (*) designates a new section. For example, the HOLIDAY section starts with a line:

```
*Holiday
```

The various sections are defined below.

DATE::MANIP VARIABLES

All Date::Manip variables which can be used are described in the following section.

IgnoreGlobalCnf

If this variable is used (any value is ignored), the global config file is not read. It must be present in the initial call to Date_Init or the global config file will be read.

EraseHolidays

If this variable is used (any value is ignored), the current list of defined holidays is erased. A new set will be set the next time a config file is read in. This can be set in either the global config file or as a Date_Init argument (in which case holidays can be read in from both the global and personal config files) or in the personal config file (in which case, only holidays in the personal config file are counted).

PathSep

This is a regular expression used to separate multiple paths. For example, on Unix, it defaults to a colon (:) so that multiple paths can be written PATH1:PATH2 . For Win32 platforms, it defaults to a semicolon (;) so that paths such as ''c:\;d:\'' will work.

GlobalCnf

This variable can be passed into Date_Init to point to a global configuration file. The value must be the complete path to a config file.

By default, no global config file is read. Any time a global config file is read, the holidays are erased.

Paths may have a tilde (~) expansion on platforms where this is supported (currently Unix and VMS).

PersonalCnf

This variable can be passed into Date_Init or set in a global config file to set the name of the personal configuration file.

The default name for the config file is .DateManip.cnf on all Unix platforms and Manip.cnf on all non-Unix platforms (because some of them insist on 8.3 character filenames :−).

PersonalCnfPath

This is a list of paths separated by the separator specified by the PathSep variable. These paths are each checked for the PersonalCnf config file.

Paths may have a tilde (~) expansion on platforms where this is supported (currently Unix and VMS).

Language

Date::Manip can be used to parse dates in many different languages. Currently, it is configured to read the following languages (the version in which they added is included for historical interest):

```
English     (default)
French      (5.02)
Swedish     (5.05)
German      (5.31)
Dutch       (5.32)     aka Nederlands
Polish      (5.32)
Spanish     (5.33)
Portuguese  (5.34)
Romanian    (5.35)
Italian     (5.35)
Russian     (5.41)
Turkish     (5.41)
Danish      (5.41)
```

Others can be added easily. Language is set to the language used to parse dates. If you are interested in providing a translation for a new language, email me (see the AUTHOR section below) and I'll send you a list of things that I need.

DateFormat
Different countries look at the date 12/10 as Dec 10 or Oct 12. In the United States, the first is most common, but this certainly doesn't hold true for other countries. Setting DateFormat to ''US'' forces the first behavior (Dec 10). Setting DateFormat to anything else forces the second behavior (Oct 12).

TZ If set, this defines the local time zone. See the TIME ZONES section above for information on its format.

ConvTZ
All date comparisons and calculations must be done in a single time zone in order for them to work correctly. So, when a date is parsed, it should be converted to a specific time zone. This allows dates to easily be compared and manipulated as if they are all in a single time zone.

The ConvTZ variable determines which time zone should be used to store dates in. If it is left blank, all dates are converted to the local time zone (see the TZ variable above). If it is set to one of the time zones listed above, all dates are converted to this time zone. Finally, if it is set to the string ''IGNORE'', all time zone information is ignored as the dates are read in (in this case, the two dates ''1/1/96 12:00 GMT'' and ''1/1/96 12:00 EST'' would be treated as identical).

Internal
When a date is parsed using ParseDate, that date is stored in an internal format which is understood by the Date::Manip routines UnixDate and DateCalc. Originally, the format used to store the date internally was:

```
YYYYMMDDHH:MN:SS
```

It has been suggested that I remove the colons (:) to shorten this to:

```
YYYYMMDDHHMNSS
```

The main advantage of this is that some databases are colon delimited which makes storing a date from Date::Manip tedious.

In order to maintain backwards compatibility, the Internal variable was introduced. Set it to 0 (to use the old format) or 1 (to use the new format).

FirstDay
It is sometimes necessary to know what day of week is regarded as first. By default, this is set to Monday, but many countries and people will prefer Sunday (and in a few cases, a different day may be desired). Set the FirstDay variable to be the first day of the week (1=Monday, 7=Sunday) Monday should be chosen to to comply with ISO 8601.

WorkWeekBeg, WorkWeekEnd
The first and last days of the work week. By default, Monday and Friday. WorkWeekBeg must come before WorkWeekEnd numerically. The days are numbered from 1 (Monday) to 7 (Sunday).

There is no way to handle an odd work week of Thu to Mon for example or 10 days on, 4 days off.

WorkDay24Hr
If this is non-nil, a work day is treated as being 24 hours long. The WorkDayBeg and WorkDayEnd variables are ignored in this case.

WorkDayBeg, WorkDayEnd
The times when the work day starts and ends. WorkDayBeg must come before WorkDayEnd (i.e. there is no way to handle the night shift where the work day starts one day and ends another). Also, the workday MUST be more than one hour long (of course, if this isn't the case, let me know... I want a job there!).

The time in both can be in any valid time format (including international formats), but seconds will be ignored.

TomorrowFirst
Periodically, if a day is not a business day, we need to find the nearest business day to it. By default, we'll look to ''tomorrow'' first, but if this variable is set to 0, we'll look to ''yesterday'' first. This is only used in the Date_NearestWorkDay and is easily overridden (see documentation for that function).

DeltaSigns
Prior to Date::Manip version 5.07, a negative delta would put negative signs in front of every component (i.e. ''0:0:−1:−3:0:−4''). By default, 5.07 changes this behavior to print only 1 or two signs in front of the year and day elements (even if these elements might be zero) and the sign for year/month and day/hour/minute/second are the same. Setting this variable to non-zero forces deltas to be stored with a sign in front of every element (including elements equal to 0).

Jan1Week1
ISO 8601 states that the first week of the year is the one which contains Jan 4 (i.e. it is the first week in which most of the days in that week fall in that year). This means that the first 3 days of the year may be treated as belonging to the last week of the previous year. If this is set to non-nil, the ISO 8601 standard will be ignored and the first week of the year contains Jan 1.

YYtoYYYY
By default, a 2 digit year is treated as falling in the 100 year period of CURR−89 to CURR+10. YYtoYYYY may be set to any integer N to force a 2 digit year into the period CURR-N to CURR+(99−N). A value of 0 forces the year to be the current year or later. A value of 99 forces the year to be the current year or earlier. Since I do no checking on the value of YYtoYYYY, you can actually have it any positive or negative value to force it into any century you want.

YYtoYYYY can also be set to ''C'' to force it into the current century, or to ''C##'' to force it into a specific century. So, in 1998, ''C'' forces 2 digit years to be 1900−1999 and ''C18'' would force it to be 1800−1899.

It can also be set to the form ''C####'' to force it into a specific 100 year period. C1950 refers to 1950−2049.

UpdateCurrTZ
If a script is running over a long period of time, the time zone may change during the course of running it (i.e. when daylight saving time starts or ends). As a result, parsing dates may start putting them in the wrong time zone. Since a lot of overhead can be saved if we don't have to check the current time zone every time a date is parsed, by default checking is turned off. Setting this to non-nil will force time zone checking to be done every time a date is parsed... but this will result in a considerable performance penalty.

A better solution would be to restart the process on the two days per year where the time zone switch occurs.

IntCharSet
If set to 0, use the US character set (7−bit ASCII) to return strings such as the month name. If set to 1, use the appropriate international character set. For example, If you want your French representation of December to have the accent over the first ''e'', you'll want to set this to 1.

ForceDate
This variable can be set to a date in the format: YYYY−MM−DD−HH:MN:SS to force the current date to be interpreted as this date. Since the current date is used in parsing, this string will not be parsed and MUST be in the format given above.

TodayIsMidnight
If set to a true value (e.g. 1), then ''today'' will mean the same as ''midnight today''; otherwise it will mean the same as ''now''.

HOLIDAY SECTION

The holiday section of the config file is used to define holidays. Each line is of the form:

```
DATE = HOLIDAY
```

HOLIDAY is the name of the holiday (or it can be blank in which case the day will still be treated as a holiday... for example the day after Thanksgiving or Christmas is often a work holiday though neither are named).

DATE is a string which can be parsed to give a valid date in any year. It can be of the form

```
Date
Date + Delta
Date − Delta
Recur
```

A valid holiday section would be:

```
*Holiday

1/1                             = New Year's Day
third Monday in Feb             = Presidents' Day
fourth Thu in Nov               = Thanksgiving

# The Friday after Thanksgiving is an unnamed holiday most places
fourth Thu in Nov + 1 day       =

1*0:0:0:0:0:0*EASTER            = Easter
1*11:0:11:0:0:0*DWD             = Veteran's Day (observed)
1*0:0:0:0:0:0*EASTER,PD5        = Good Friday
```

In a Date + Delta or Date − Delta string, you can use business mode by including the appropriate string (see documentation on DateCalc) in the Date or Delta. So (in English), the first workday before Christmas could be defined as:

```
12/25 − 1 business day          =
```

The dates may optionally contain the year. For example, the dates

```
1/1
1/1/1999
```

refers to Jan 1 in any year or in only 1999 respectively. For dates that refer to any year, the date must be written such that by simply appending the year (separated by spaces) it can be correctly interpreted. This will work for everything except ISO 8601 dates, so ISO 8601 dates may not be used in this case.

Note that the dates are specified in whatever format is set using the Date_Init options, so if the standard parsing is D/M/YYYY, you would need to specify it as:

```
25/12/2002            = Christmas
```

In cases where you are interested in business type calculations, you'll want to define most holidays using recurrences, since they can define when a holiday is celebrated in the financial world. For example, Christmas should be defined as:

```
1*12:0:24:0:0:0*FW1  = Christmas
```

NOTE: It was pointed out to me that using a similar type recurrence to define New Years does not work. The recurrence:

```
1*12:0:31:0:0:0*FW1
```

fails (worse, it goes into an infinite loop). The problem is that each holiday definition is applied to a specific year and it expects to find the holiday for that year. When this recurrence is applied to the year 1995, it returns the holiday for 1996 and fails.

Use the recurrence:

```
1*1:0:1:0:0:0*NWD
```

instead.

If you wanted to define both Christmas and Boxing days (Boxing is the day after Christmas, and is celebrated in some parts of the world), you could do it in one of the following ways:

```
1*12:0:24:0:0:0*FW1  = Christmas
1*12:0:25:0:0:0*FW1  = Boxing

 1*12:0:24:0:0:0*FW1 = Christmas
01*12:0:24:0:0:0*FW1 = Boxing

1*12:0:24:0:0:0*FW1   = Christmas
1*12:0:25:0:0:0*FW1,a = Boxing
```

The following examples will NOT work:

```
1*12:0:24:0:0:0*FW1  = Christmas
1*12:0:24:0:0:0*FW2  = Boxing

1*12:0:24:0:0:0*FW1  = Christmas
1*12:0:24:0:0:0*FW1  = Boxing
```

The reasoning behind all this is as follows:

Holidays go into affect the minute they are parsed. So, in the case of:

```
1*12:0:24:0:0:0*FW1  = Christmas
1*12:0:24:0:0:0*FW2  = Boxing
```

the minute the first line is parsed, Christmas is defined as a holiday. The second line then steps forward 2 work days (skipping Christmas since that's no longer a work day) and define the work day two days after Christmas, NOT the day after Christmas.

An good alternative would appear to be:

```
1*12:0:24:0:0:0*FW1  = Christmas
1*12:0:24:0:0:0*FW1  = Boxing
```

This unfortunately fails because the recurrences are currently stored in a hash. Since these two recurrences are identical, they fail (the first one is overwritten by the second and in essence, Christmas is never defined).

To fix this, make them unique with either a fake flag (which is ignored):

```
1*12:0:24:0:0:0*FW1,a  = Boxing
```

or adding an innocuous 0 somewhere:

```
01*12:0:24:0:0:0*FW1   = Boxing
```

The other good alternative would be to make two completely different recurrences such as:

```
1*12:0:24:0:0:0*FW1  = Christmas
1*12:0:25:0:0:0*FW1  = Boxing
```

At times, you may want to switch back and forth between two holiday files. This can be done by calling the following:

```
Date_Init("EraseHolidays=1","PersonalCnf=FILE1");
...
Date_Init("EraseHolidays=1","PersonalCnf=FILE2");
...
```

EVENTS SECTION

The Events section of the config file is similar to the Holiday section. It is used to name certain days or times, but there are a few important differences:

Events can be assigned to any time and duration

All holidays are exactly 1 day long. They are assigned to a period of time from midnight to midnight.

Events can be based at any time of the day, and may be of any duration.

Events don't affect business mode calculations

Unlike holidays, events are completely ignored when doing business mode calculations.

Whereas holidays were added with business mode math in mind, events were added with calendar and scheduling applications in mind.

Every line in the events section is of the form:

```
EVENT = NAME
```

where NAME is the name of the event, and EVENT defines when it occurs and its duration. An EVENT can be defined in the following ways:

```
Date
Date*

Date  ; Date
Date  ; Delta
```

Here, Date* refers to a string containing a Date with NO TIME fields (Jan 12, 1/1/2000, 2010−01−01) while Date does contain time fields. Similarly, Recur* stands for a recurrence with the time fields all equal to 0) while Recur stands for a recurrence with at least one non-zero time field.

Both Date* and Recur* refer to an event very similar to a holiday which goes from midnight to midnight.

Date and Recur refer to events which occur at the time given and with a duration of 1 hour.

Events given by ''Date ; Date'', ''Date ; Delta'', and ''Recur ; Delta'' contain both the starting date and either ending date or duration.

Events given as three elements ''Date ; Delta ; Delta'' or ''Recur ; Delta ; Delta'' take a date and add both deltas to it to give the starting and ending time of the event. The order and sign of the deltas is unimportant (and both can be the same sign to give a range of times which does not contain the base date).

KNOWN PROBLEMS

The following are not bugs in Date::Manip, but they may give some people problems.

Unable to determine Time Zone

Perhaps the most common problem occurs when you get the error:

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

Date::Manip tries hard to determine the local time zone, but on some machines, it cannot do this (especially non-Unix systems). To fix this, just set the TZ variable, either at the top of the Manip.pm file, in the DateManip.cnf file, or in a call to Date_Init. I suggest using the form ''EST5EDT'' so you don't have to change it every 6 months when going to or from daylight saving time.

Windows NT does not seem to set the time zone by default. From the Perl−Win32−Users mailing list:

```
> How do I get the TimeZone on my NT?
>
>      $time_zone = $ENV{'TZ'};
>
You have to set the variable before, WinNT doesn't set it by
default.  Open the properties of "My Computer" and set a SYSTEM
variable TZ to your time zone.   Jenda@Krynicky.cz
```

This might help out some NT users.

A minor (false) assumption that some users might make is that since Date::Manip passed all of its tests at install time, this should not occur and are surprised when it does.

Some of the tests are time zone dependent. Since the tests all include input and expected output, I needed to know in advance what time zone they would be run in. So, the tests all explicitly set the time zone using the TZ configuration variable passed into Date_Init. Since this overrides any other method of determining the time zone, Date::Manip uses this and doesn't have to look elsewhere for the time zone.

When running outside the tests, Date::Manip has to rely on its other methods for determining the time zone.

Missing date formats

Please see the Date::Manip::Problems document for a discussion.

Complaining about getpwnam/getpwuid

Another problem is when running on Micro$oft OS's. I have added many tests to catch them, but they still slip through occasionally. If any ever complain about getpwnam/getpwuid, simply add one of the lines:

```
$ENV{OS} = Windows_NT
$ENV{OS} = Windows_95
```

to your script before

```
use Date::Manip
```

Date::Manip is slow

The reasons for this are covered in the SHOULD I USE DATE::MANIP section above.

Some things that will definitely help:

Version 5.21 does run noticeably faster than earlier versions due to rethinking some of the initialization, so at the very least, make sure you are running this version or later.

ISO-8601 dates are parsed first and fastest. Use them whenever possible.

Avoid parsing dates that are referenced against the current time (in 2 days, today at noon, etc.). These take a lot longer to parse.

```
Example:  parsing 1065 dates with version 5.11 took 48.6 seconds, 36.2
seconds with version 5.21, and parsing 1065 ISO-8601 dates with version
5.21 took 29.1 seconds (these were run on a slow, overloaded computer with
little memory... but the ratios should be reliable on a faster computer).
```

Business date calculations are extremely slow. You should consider alternatives if possible (i.e. doing the calculation in exact mode and then multiplying by 5/7). Who needs a business date more accurate than ''6 to 8 weeks'' anyway, right :-)

Never call Date_Init more than once. Unless you're doing something very strange, there should never be a reason to anyway.

Sorting Problems

If you use Date::Manip to sort a number of dates, you must call Date_Init either explicitly, or by way of some other Date::Manip routine before it is used in the sort. For example, the following code fails:

```
use Date::Manip;
# Date_Init;
sub sortDate {
    my($date1, $date2);
    $date1 = ParseDate($a);
    $date2 = ParseDate($b);
    return (Date_Cmp($date1,$date2));
}
@dates = ("Fri 16 Aug 96",
          "Mon 19 Aug 96",
          "Thu 15 Aug 96");
@i=sort sortDate @dates;
```

but if you uncomment the Date_Init line, it works. The reason for this is that the first time you call Date_Init, it initializes a number of items used by Date::Manip. Some of these have to be sorted (regular expressions sorted by length to ensure the longest match). It turns out that Perl has a bug in it which does not allow a sort within a sort. At some point, this should be fixed, but for now, the best thing to do is to call Date_Init explicitly. The bug exists in all versions up to 5.005 (I haven't tested 5.6.0 yet).

NOTE: This is an EXTREMELY inefficient way to sort data (but read the 2nd note below for an easy way to correct this). Instead, you should parse the dates with ParseDate, sort them using a normal string comparison, and then convert them back to the format desired using UnixDate.

NOTE: It has been reported to me that you can still use ParseDate to sort dates in this way, and be quite efficient through the use of Memoize. Just add the following lines to your code:

```
use Date::Manip;
use Memoize;
memoize('ParseDate');
...
@i=sort sortDate @dates;
```

Since sortDate would call ParseDate with the same data over and over, this is a perfect application for the Memoize module. So, sorting with ParseDate is no longer slow for sorting.

RCS Control

If you try to put Date::Manip under RCS control, you are going to have problems. Apparently, RCS replaces strings of the form ''$Date...$'' with the current date. This form occurs all over in Date::Manip. To prevent the RCS keyword expansion, checkout files using ''co −ko''. Since very few people will ever have a desire to do this (and I don't use RCS), I have not worried about it.

KNOWN BUGS

Daylight Saving Times

Date::Manip does not handle daylight saving time, though it does handle time zones to a certain extent. Converting from EST to PST works fine. Going from EST to PDT is unreliable.

The following examples are run in the winter of the US East coast (i.e. in the EST time zone).

```
print UnixDate(ParseDate("6/1/97 noon"),"%u"),"\n";
=> Sun Jun  1 12:00:00 EST 1997
```

June 1 EST does not exist. June 1st is during EDT. It should print:

```
=> Sun Jun  1 00:00:00 EDT 1997
```

Even explicitly adding the time zone doesn't fix things (if anything, it makes them worse):

```
print UnixDate(ParseDate("6/1/97 noon EDT"),"%u"),"\n";
=> Sun Jun  1 11:00:00 EST 1997
```

Date::Manip converts everything to the current time zone (EST in this case).

Related problems occur when trying to do date calculations over a time zone change. These calculations may be off by an hour.

Also, if you are running a script which uses Date::Manip over a period of time which starts in one time zone and ends in another (i.e. it switches form Daylight Saving Time to Standard Time or vice versa), many things may be wrong (especially elapsed time).

These problems will not be fixed in Date::Manip 5.xx. Date::Manip 6.xx has full support for time zones and daylight saving time.

BUGS AND QUESTIONS

Please refer to the Date::Manip::Problems documentation for information on submitting bug reports or questions to the author.

SEE ALSO

Date::Manip − main module documentation

LICENSE

AUTHOR

Sullivan Beck (sbeck@cpan.org)