



A standard for real-world dates

Peter Fox

day@PeterFox.ukfsn.org

Status of this document

Being tinkered with ie. not definitive.

This is a theoretical study which hasn't been exposed to discussion or informed by attempted implementation.

It should be used only as a guide. Contact the author or visit vulpeculox.net for latest developments.

- Project dormant

Contents

This is the complete **D/a/y** specification. (Tools, test-cases, learning resources and implementation specific details are documented separately.)

I Overview

- 1 Introduction - Dates are not all points in time
- 2 Definitions
- 3 Model
- 4 Encoding schemes
- 5 Operations

II Representation

- 1 General
- 2 Logical signatures
- 3 32bit encoding scheme
- 4 Unix timestamp encoding scheme
- 5 Encoding wrinkles
- 6 Invalid reasons
- 7 Integer sorting
- 8 Constants
- 9 Limits
- 10 Notes

III Functions

- 1 Background
- 2 Functions taking no arguments
- 3 Tests taking a single day argument and returning a boolean
- 4 Functions taking a single day argument and returning a day
- 5 Other single argument functions
- 6 Functions taking two arguments returning boolean
- 7 Functions taking two arguments returning scalar
- 8 Functions taking two arguments returning Day
- 9 Output conversion functions
- 10 Input conversion functions
- 11 Environment and utility functions

IV Internationalisation

- 1 Accessing an appendix
- 2 Logical contents overview
- 3 Physical layout
- 4 Appendix en-uk
- 5 Low level functions

Overview

In this part we describe the model and usage.

1 Introduction - Dates are not all points in time

Being able to specify dates precisely, approximately or abstractly is magic when you come to write real-world applications where data is imperfect or not yet available.

Q: When was some book published?

A: 2007

Q: When was the author born?

A: 14th June 1964

Q: When did they die?

A: Not yet!

Q: When is their next book coming out?

A: February

Q: When did they win the Booker Prize?

A: Unknown

Q: When did they win the Nobel prize for literature?

A: Never

How do you store "Beginning of time", "Unknown", "September", "2004", "Last day of February" and "14th June 2007" using the same metric, integer or database column? Is "June 2007" earlier than "14th June 2007"? - Or later? Or both?

The trouble is we're trying to store abstracts, ranges and points in time all together. The traditional method of storing dates on computer is to model a timeline divided into days, seconds or milliseconds. This falls flat when we want to indicate, perfectly legitimately, a non-point in time. Suppose for example I have a database of assets with acquisition and disposal dates. What do I put in the "date sold" column when I've still got the item? If I put in a dummy date of say the 31st December 2099 any calculations I try to see how frequently I modernise my assets will be bonkers. If I'm planning a year ahead then I might set some activity to happen in "September 2009" (my holiday for example). This shouldn't be appearing as "one day's holiday on the 1st September".

There are more complications when we want to use an interval measure which isn't quite the same thing as either the number of days or a calendar date.

The following is a specification for

- A model for specifying dates with resolutions down to individual days.
- A 32 bit unsigned integer and Unix timestamp compatible encoding schemes
- Method specifications for an implementation
- A framework for helper methods for text input, display and internationalisation.

There is a lot more detail here than most people need to know just to get started with D/a/ys, however it is important to recognise that when we depart from a simple timeline life gets more complicated and simple arithmetic is no longer enough. Still, as that's the way the real world works so we'd better get to grips with it.

2 Definitions

Calendar date

This is 'a date that might appear on a calendar'.

- Just a year
- Just a year and month
- A year month and day

Floating date

This applies to days with unspecified 'more significant' components. ie.

- Only month and day components specified eg. Birthday
- Only day component specified eg. Monthly pay day

Fully specified date

A *calendar date* where the three elements, year, month and day are defined and real.

Interval

A d/a/y object may be used to explicitly represent a time period expressed as \pm, Y, M, D .

- The sign applies to the whole.
- Intervals should not be used to represent dates.

Julian date

The number of days elapsed January 1st 4713 BC. For our purposes a serial number for days which always 'counts one' every 24 hours. This only applies to fully specified dates, but where this holds we can do exact date arithmetic.¹

Period

A date never specifies a moment in time. The least span of time it covers is 24 hours. The Period of a date is the full range of possible dates that it might cover. For example "May 2007" covers a Period from "1st May 2007" to "31st May 2007" inclusive.

Note. Do not get the terminology of Periods mixed up with Intervals. An Interval is a particular type of Day object. A Period is a lack of *Precision*.

Precision

Analogous to numerical precision when dates are specified as Y-M-D with Y being the most and D being least significant.

Signature

A top-level indicator of the type of value contained in a day encoding. Described in part II.

¹ WARNING: Sometimes Julian Date is *misused* to mean the "ordinal date" which is the number of the day in the year.

3 Model

Logical representation

Let us define an object that is capable of representing the following 'dates':

Abstracts

Not a valid date	An error (NV)
Unknown	• Not known (NK)
Beginning of time	• Some unspecified time in the past (BoT)
End of time	• Some unspecified time in the future (EoT)

Floating labels

Some month	eg March
Some day	eg 15 th
Last day of...	eg Last day of February

Ranges

Some year	• eg 2007
Some year+month	eg March 2007

Fully specified dates

Some y+m+d	• eg 4 th August 1976
Last day of y+m	eg Last day of February 2008

(In my personal experience the ones marked • are the most often used.)

We also need to be clear about intervals. 1 year is not the same thing as 365 days and months are worse! (Suppose our planned project is now scheduled to start on the first of March instead of the first of February:- Do we add 1 month or 28 days to all our task start and end dates?)

An interval is composed of four elements. A sign, zero or some years, zero or some months, zero or some days. Intervals are simply a convenient way to express time differences without having to guess how many days to a month or year.

Operations

Firstly we need to recognise that date arithmetic is going to be problematical unless we're working with fully specified days. If we are then we can do things like:

"How many days difference between FSD1 and FSD2 ?"

If we're dealing with floating months or Y+M (both operands being the same) we can sensibly find a difference in months (or years and months). Similarly 2006 minus 1945 gives 61 years.

We can do comparisons with varying types of d/a/y. For example "Is 14th March 2008 before June 2008?" This leads us to a tricky area. Is "March 2008 before 14th March 2008?" and "Is 14th March 2008 before March 2008?" One school of thought goes: "Since we're bound to need an ordering sequence let's use that as our definition of 'before'" A better alternative goes: "We need to look at the meaning we attach to these values more carefully before jumping to a conclusion." Consider what happens if a time-elf travels forward in time during 2008. In order, the elf comes across February, Last day of February, March, March 1st,...March 14th. So the elf can say "I've arrived at 'March' before arriving at '14th March'" and therefore "March is before 14th March". However the same argument applies if the time-elf is going backwards in time from April; as it moves from April to March the elf 'arrives at March' before 'arriving at 14th March". The results so far are:

Is March before 14 th March?	Yes	Is March after 14 th March?	Yes
Is 14 th March before March?	No	Is 14 th March after March?	No.

If we were dealing exclusively with points on a timeline we could say that “is D1 before D2” is true then “is D1 after D2” must be false (and vice versa). But ranges break this logic. Similar considerations apply to years.

However there is another operation we can do: “Is 14th March *contained* by March?”. What this means is that we have to think a little bit differently, taking account of the way dates are used in the real world, in order to work with them and at least get something useful out when we try to manipulate them. For example “How long have you had that plaster cast on your leg?” Our database will contain the date it went on (say 3rd Feb 2008) and the date it came off, which it hasn't yet, of either Unknown or End-of-time. Hopefully the question will turn into “How long was it ...” when the date it came off is now a fully qualified date and we'd be able to do conventional date arithmetic.

```
PlasterDuration = DaysDifference( DateOn , DateOff );
```

So, taking the last case first:

```
PlasterDuration = DaysDifference( '3 Feb 08','18 Mar 08'); // Easy: 44 days
```

Now for the tricky case:

```
PlasterDuration = DaysDifference( '3 Feb 08','End-of-time'); //?
```

What do we mean here? Do we want to return infinity? *Probably* we actually want to return the number of days between 3rd Feb and *today*. So if today is 11th March this will give us a result of 37 days and counting. This makes immediate sense if we want to know how old somebody is whether or not they're still living. This means that the logic for the days difference function (simplified for illustration) goes like:-

Is second date EndOfTime? If so then use Today as the operative date value.²

Converting to linear time

Now we're in real trouble because linear time simply can't represent the concepts we've got in D/a/y-time. We can only relax is if all our dates are *fully qualified*, but that's unlikely as we wouldn't be using this scheme if life was that simple. A D+M+Y can easily (subject to operating system and programming language restrictions) be converted into common linear times such as timestamps as used globally by databases and operating systems.

We are going to need some hack to make the best of a bad job. One way we can approach this is to attempt to get linear time mapped to a sort order. So how do we sort say the following: 2007, 2008, Feb 2008, Feb 1st 2008, Feb 28th 2008 , Feb 29th 2008, Last day of Feb 2008, March 2008, 14th March 2008 and 2009? The order I've given is what I think is sensible, but as we've seen above “March 2008” comes just as much after “March 14th 2008” as before it. Also the leap year makes “Last day of Feb” (a) variable between years and possibly undefined if a year isn't given and (b) *identical* to (in this case) Feb 29th. It's probably going to be convenient if EndOfTime is at the end, but really it ought to be off end of the scale and NotValid shouldn't be there at all!

For practical purposes we want to achieve some interoperability between, lets call them 'timestamps', and d/a/ys. It looks like we need to treat fully qualified dates as standard with hacks for the unusual ones which are slipped in between the standard dates. We can achieve this fairly simply if we recognise that timestamps have a much greater resolution than a day. So for example “1st March 08” will be represented in a timestamp form as “zero milliseconds after midnight on 1st March 08” (or something similar). So What about “March 08”? This wants to come before the 1st if we're to emulate the sorting scheme just

² This eventually leads to an interesting anomaly giving a zero result if we ask what's the difference between BeginningOfTime and EndOfTime!

described. As it stands we have no room before trampling onto "23:59:59 on the 29th Feb". Obviously we'll have to declare something like "standard days will have their 'bits' set to noon" to give us room earlier in the day to put in a fake timestamp for our specials. There are two layers of this with months and years. 1st Jan 2008 can't be zero milliseconds after midnight on 1-1-2008 because that doesn't leave room for plain 2008.

Do we want to be able to convert timestamps into days? Yes. There may be communicating applications that 'talk timestamp'. If they are educated enough to tell us about special cases, particularly the constants BoT, EoT, NK and NV we can give them a some very useful functionality we enjoy with d/a/ys at a stroke.

Model summary

To represent what happens in the real world we need to break away from linear time. We can still manipulate dates but the rules and results are more complex.

4 Encoding schemes

For the purpose of interoperation we need standard encoding schemes. Two schemes are specified.

- a 32 bit unsigned integer
- b Wrapping in a 'Unix' timestamp³

All encodings should return exactly the same sequence when sorted numerically.

General

All encodings are lossless.

The Unix timestamp suffers from limited year range (1902-2037)

The 32 bit encoding gives +/- 4 millennia so for practical purposes there is no need to obtain an offset for the year 'from somewhere'. Spare bits in the encoding are used to store shortcut information.

Intervals are identifiable as being distinct from dates. They should sort numerically in all encodings with from the most negative to the most positive. There is not meant to be any correlation between the encoded values of say the day "12th March" and the interval "+0y3m12d".

Special signatures

We define the following constants as signatures to be used as the most significant bits of the 32bit encoding scheme

Table 1

Mnemonic	Meaning	Binary
NVI	Invalid interval	0 0 0
INT	Interval	0 0 1
NV	Not a date value	0 1 0

³ The purpose of providing a compatibility mode with Unix timestamps is to provide a method of using huge quantities of legacy application code to transfer data without having to rewrite everything from end to end.

FLO	Floating date	0 1 1
NK	Not known	1 0 0
BoT	Beginning of time	1 0 1
Cal	Calender date	1 1 0
EoT	End of time	1 1 1

Encoding schemes overview

The 32 bit encoding uses starts with the signature as the MSBs, year with sign, month and day following. Seven bits are used for other purposes.

The basic Unix timestamp principle is that the DMY components are encoded as you'd expect with the 'h:m:s' part of the timestamp being used for extra information of which the hours units is used to carry the signature and the seconds for offset in amounts of 64 years to extend the year range.

Encoding wrinkles

'Last day of Feb' is encoded as 31st Feb. @@@ Is it?

Table 3

Day	Description	UNIX Timestamp	32 bit
	General date layout	yyyy-mm-dd ??:?:00	sss±yyyy yyyyyyyy mmmmdddd dwww×vf0
	General interval layout	yyyy-mm-dd 0s :ir:zq	001±yyyy yyyyyyyy mmmmdddd d××××v×1
NVI	Not valid interval	<i>Not applicable</i>	000××××× ×××××××× ×××××××× ×r r r r 0×1
NV	Not valid date	1901-12-13 02:0r:00	010××××× ×××××××× ×××××××× ×r r r r 0×0
NK	Not known	1901-12-13 04:00:00	100××××× ×××××××× ×××××××× ××××××××
BoT	Beginning of time	1901-12-13 05:00:00	101××××× ×××××××× ×××××××× ××××××××
EoT	End of time	2038-01-18 07:00:00	111××××× ×××××××× ×××××××× ××××××××
	Year only eg 2009	2009-01-01 03:03:00	01100111 11011000 00000000 0000×100
	Year-month only eg Mar 2009	2009-03-01 03:01:30	01100111 11011000 00110000 0000×100
	Fully specified eg 14 th Mar '09	2009-03-14 06:00:00	11000111 11011000 00110111 0110×110

× ... not used s ... signature w... day of week
 r ... reason code v ... validity f ... fully specified
 i ... signs z ... zero m/d q ... 64-year offset

5 API and methods

Abstract constants

- NV Not a valid day value. An error or uninitialised.
 NK Not known. We 'know we don't know'.
 BoT Beginning of time.
 (a) A day earlier than we're otherwise worried about.
 (b) An unknown past date.
 EoT End of time.
 (a) A day later than we're otherwise worried about

(b) An unknown future date.

Note: BoT and EoT could be in the future or past respectively if we want to use them in that way relative to another in the sense of "some earlier date or "some later date".

Methods

TO BE RE ORGANISED AND CHECKED

Functions taking no arguments

Today()⁴

Tests taking a single day argument and returning a boolean⁵

IsValid(Day)	IsKnown(Day)
IsBoT(Day)	IsEoT(Day)
IsReal(Day)	IsSpecific(Day)
IsRealPeriod(Day)	HasMonth(Day)
HasDay(Day)	HasYear(Day)

Functions taking a single day argument and returning a day

FirstDay(Day)	LastDay(Day)
Next(Day)	Previous(Day)
Month(Day)	YearMonth(Day)

Other single argument functions

DayOfWeek(Day)	MonthNumber(Day)
DayNumber(Day)	YearNumber(Day)

Functions taking two arguments returning boolean

SortsBefore(Day1,Day2)	Before(Day1,Day2)
After(Day1,Day2)	Contains(Day1,Day2)

Functions taking two arguments returning integer

DaysDifference(Day1,Day2)

Functions taking two arguments returning day

Difference(Day1,Day2)	Span(Day1,Day2)
Add(Day1,Day2)	Subtract(Day1,Day2)
Add(Day1,Number of days, Number of months, Number of years)	
Subtract(Day1,Number of days, Number of months, Number of years)	

⁴ It is my convention to capitalise function/method names and also argument placeholder names.

⁵ For the purpose of description these will be illustrated as functions eg IsValid(Day) rather than object methods Day.IsValid(). Implementations may chose whatever scheme suits.

Output conversion functions

ToTimestamp(Day) To32Bits(Day)
 ToString(Day,Format)

Input conversion functions

FromTimestamp(Timestamp) @@@ this list needs checking
 From32Bits(Unsigned 32bit integer)
 FromString(String)⁶
 FromYMD(Y integer,M integer, D integer)

Environment functions

SetLimitToUnix() SetLimitTo32Bit()
 GetLimitEarliestDate() GetLimitLatestDate()
 SetAppendix()

6 Non-linear time problems

Unfortunately for fans of arithmetic, each year doesn't have 13 months of 28 days; instead we have a non deterministic world of guesswork.

Adding intervals to dates

What are the answers to the following (remembering 2008 is a leap year) :

- (a) 1st January 2008 plus 1 month? 1st Feb
- (b) 21st January 2008 plus 1 month? 21st Feb
- (c) 30th January 2008 plus 1 month? 29th Feb or 1st March
- (d) 31st January 2008 plus 1 month? 29th Feb or 2nd March
- (e) 28th January 2009 plus 1 month? 28th Feb or 25th Feb
- (f) 29th January 2009 plus 1 month? 28th Feb or 1st March or 26th Feb
- (g) 30th January 2009 plus 1 month? 28th Feb or 2nd March or 27th Feb
- (h) 31st January 2009 plus 1 month? 28th Feb or 3rd March
- (i) 29th February 2008 plus 1 year? 28th Feb or 1st March 2009

Are we all agreed with (a)? It is pretty reasonable that one month after the first of any month will be the first of the next month. And so for the 2nd, 3rd and so on... What about the last day of the month? Shouldn't a one month addition map to the last day of the next month? By this reasoning (d) and (h) should give the last day of February. (Different because 2008 is a leap year.) What about the 2nd and 3rd from last days? Shouldn't these map to the 2nd and 3rd from last days of the next month? (f) is 2 days before the end of January so surely the result should be two days before the end of February?

How many days in a year?

365 or 366. How can we tell which? If we have a real date to add days to then we can do this by converting the date into a Julian date number adding the days then converting back to YMD. But what happens if we're 'doing arithmetic' that involves juggling years, months and days without a real base point in time? Is 1 year plus 365 days always two years?

Addition rules

In rough summary (details in Part III section 2) when adding a combination of YMD to a date add the years on first, then the months, finally the days. When adding months adjust the day of the result month to reflect any different length so that for example the last day of one month will be hacked to the last day of another. Days are added on using a real

⁶ We will discuss the complications of this in the next section.

calendar.

When adding two intervals together provide alternative methods. Either fudge as scalar using fixed conversion factors between years, months and days or use some base date to act as a surrogate datum for the method used for real dates.

There will be additional rules for rules for floating dates and not fully specified dates but these complications are better dealt with in the specification of the functions involved.

Representation

In this part we

- define logical model components
- define a 32 bit encoding scheme
- define how Unix timestamps can be used, with limitations, in a compatible way.

1 General

All encodings are lossless.

Two representations both using 32bits are specified. The Unix Timestamp version is provided to simplify interoperation with other libraries where it isn't convenient to implement a 'clean sweep'.

The standard Unix timestamp suffers from limited year range (1970-2037) but it is possible to extend this to span from 1806 BC to 5746 AD.

The 32 bit encoding gives +/- 4095 years so for practical purposes there is no need to obtain an offset for the year 'from somewhere'. Spare bits in the encoding are used to store shortcut information.

Intervals are identifiable as being distinct from dates. They should sort numerically in both encodings with from the most negative to the most positive. There is not meant to be any correlation between the encoded values of say the day "12th March" and the interval "0-3-12".

2 Logical signatures

As well as the constants NV,NK,BoT and EoT we define

INT ... Interval

NVI ... Not-valid interval

Table 4

Mnemonic	Meaning	Binary	Decimal
NVI	Invalid interval	000	0
INT	Interval	001	1
NV	Not a date value	010	2
FLO	Floating date	011	3
NK	Not known	100	4
BoT	Beginning of time	101	5
Cal	Calender date	110	6
EoT	End of time	111	7

These *signatures* will appear as high bits embedded in the 32bit encoding and in the units of the 'hh' field of the Unix timestamp.

3 32bit encoding scheme

From most significant to least significant bits:

Table 5

Bits	Size	Usage	Notes
31 - 29	3	Signature	See table 2
29	1	Sign	0:+ve 1:-ve
27 - 16	12	Year	0 ... 4095
15 - 12	4	Month	1:Jan ... 12:Dec 0 is legal
11 - 7	5	Day of month	1 ... 31 0 is legal
6 - 3	4	Day of week (Fully specified)	1:Mon ... 7:Sun 0:Undefined
		Not valid reason code (NV)	See table 7
		Not used	
2	1	Valid flag	1:Is validated
1	1	Fully specified	1:Is fully specified
0	1	Interval	1:Is interval

- The sign is applied to the year for a date or to a whole interval.
- Day of week is for convenience. Should be set to 0 if not applicable or not given
- Note that from an encoding point of view it is possible to have a signature of NV followed by Y,M,D values and a not valid reason. There is a possible use for this where dates are being supplied in bulk by a process that can't reject the data outright.
- Bits 2 - 0 are for quick-reference convenience

4 Unix timestamp encoding scheme

The 'hh:mm:ss' part of the timestamp is hijacked to represent additional information.

- The seconds field is used to store an offset from 1970 in units of 64 years.
- This encoding does not support intervals.⁷

Table 6

Element	Used for	Notes
h-tens	Always 0	
h-units	Signature	See table 2
m-tens	Offset sign	0:+ve 1:-ve

⁷ The purpose of the encoding is to be able to store and exchange dates with legacy stems. It is not intended to be used as a comprehensive storage format.

m-units	YMD Zero flags	Bits set if missing
	Invalid reason (NV only)	See table 7
ss	64-year offset	1970 is base

- The function that 'reads-in' Unix timestamps can be commanded to mask off the hh:mm:ss part of the number where these may contain spurious data. Typically this would occur where existing timestamps generated by another system are being read. With these bits masked-off the normal Unix timestamp limitations apply.
- As the Unix timestamp suffers from a severely truncated year range we use the seconds field to store a count of offsets, each of 64 years, to shift the 'yyyy' value if required. This is based on 1970. For example a 'yyyy' of 7 and an offset of +2 would represent $1970 + (2 * 64) + 7 = 2105$. Similarly 1066 would be represented by an offset of -15 and a year of 56.
- Floating and partially specified dates cause problems because there is no way to specify zero years, months or days in the standard Unix timestamp. This can be handled by using three flags combined to tell us when to ignore the Y,M or D values. For example "March 1988" would be flagged as **001** or 1 minute, "2009" as **011** or 3 minutes.

5 Encoding wrinkles

'Last day of February' is represented by 31st February.

6 Invalid reasons

It may be useful to include more details about the reason why a value is deemed invalid. See Table 7. This feature is provided so that conversion functions don't throw exceptions for data-value related issues. (Exceptions being reserved for programming issues such as an unsuitable type being supplied as an argument.)

7 Integer sorting

The 32-bit encoding will sort (unsigned) firstly in the order of signatures. Intervals will sort with negative before positive and then by magnitude. Dates will sort in 'natural forward order' with 2007 before January 2007 before 1st January 2007.

8 Constants

Encoded abstracts

Table 2

Description	Mnemonic	Timestamp		32 bit
Not valid interval	NVI	Not applicable		1
Not valid	NV	-2147551200	1901-12-13 02:00:00	1073741824
Not known	NK	-2147544000	1901-12-13 04:00:00	2147483652
Beg. of time	BoT	-2147540400	1901-12-13 05:00:00	2684354564
End of time	EoT	2147410800	2038-01-18 07:00:00	3758096388

NV and NVI reason codes are used when a function produces an invalid result and we

want to find out a little bit more about why.

Not valid reason codes

Possible value 0 - 15

Table 7 Not valid reason codes

Value	Procedure	Description
1	DateFromString	An empty string or stupidly long string was provided. (max 40 characters)
	MonthOnly	Sourced from unsuitable date
2	DateFromString	Zero, more than three tokens, or more than one alpha tokens provided.
3	DateFromString	Unable to interpret a non-numeric element
4	DateFromString	Year element is missing, illegal or not 2 or 4 characters. Or month must be followed by year or day.
	LastDay FirstDay RangeEnd	Year was undefined
5	DateFromString	Unable to parse (eg <i>dmmyy</i> needs an 0 at the front.) or can't tell day-month order.
6	DateFromString	Unsuitable number element. (Might be that two-digit years have been outlawed by the environment setting TWO_DIGIT_FIX.)
7		

Where a function emits an integer result and 0 is a valid in-band value the constant **iNV** is returned to indicate an inappropriate/erroneous value. This is declared to be -9999999.

Sometimes it is necessary to scale years, months and days. This involves approximations:

1 year = 365.25 days

1 year = 12 months

1 month = (365.25 / 12) = 30.4375 days

9 Limits

The Unix timestamp compatibility mode may be convenient to use when relating to legacy systems. This is 'harmless' so long as:

- (a) The legacy system makes no practical use of the hh:mm:ss part.
- (b) When reading using the FromTimestamp() function the appropriate bit masking flag is used to ignore or use the hh:mm:ss part.
- (c) If Days are written to a Timestamp they will wrap every 64 years. This means that dates after 2033 won't be correct if subsequently read as a conventional timestamp.

Table 8 Encoding limits

Property	32-bit encoding	Conventional Unix timestamp	Extended Unix timestamp
Earliest date	1 st Jan 4095 BC	13 th Dec 1901	1 st Jan 1806 BC
Latest date	31 st Dec 4095 AD	19 th Jan 2038	31 st Dec 5746 AD
Largest interval	± 4095y 12m 31d		

To be checked

To make things simple, internal checking will only look at the year component and only allow years where the whole year is valid. This means that the conventional Unix range is restricted to 1st Jan 1902 to 31st December 2037.

10 Notes

By convention in everyday calendars there is no year-zero. The year before 1AD is 1BC.

Functions

In this part we define the API and how it is used.

With traditional 'point-in-time' dates there are basic operations that don't need much definition. With a richer object that can be real, abstract, a point-in-time, a period, floating or fixed, an interval or representing some sort of problem we need a richer and more explicitly defined set of functions.

1 Background

D/a/y objects

Throughout the following we define the functions or methods used to manipulate day objects / types / classes. The object or class model is:-

Day	
Date	Sub-class of Day used for calendarish purposes
Interval	Sub-class of Day used for arithmetic and offset purposes

However every Day object knows whether it is a Date or an Interval by simple introspection so some implementers may chose to ignore this level of sophistication if their target language makes it unwieldily to use.

To give implementation programmers the greatest flexibility we have avoided explicit references to objects, constructors and methods although it should be clear from the following specification how these would be implemented.

Where **Day** is specified in the following it could be either an **Interval** or a **Date**.

Conventions

- The format used is functional for clarity. The same API would be used for object methods except typically for the omission of the first argument. For example:
 Function style `bool ← IsValid(Day Day)`
 Method style `bool ← Day.IsValid()`
- For clarity **this style** will be used to indicate objects/types in the text while plain "day" will refer to something like Thursday or 5th August.
- Conventionally functions/methods and argument placeholders are initially capitalised.
- The types of arguments is given after variable labels. The details of various string types have been ignored.
- *For purposes of illustration* many examples are shown with string values where a proper day argument would be used. eg.
`IsValid("37th March 2008")`
 would be used for illustration instead of something like
`var Day d = new Day(2008,3,37);`
`var Bool b = d.IsValid();`

Error handling

There are two ways these functions can 'report something is wrong':

- By throwing an exception
- or returning a value with a specific meaning.

Exceptions are used for 'programming' issues such as inappropriate arguments. Data-related problems such as for example trying to create a date for the "75th January" will normally return NV or NVI, possibly with additional information along the lines of "That's not sensible - so here is a result with NV".

- Boolean results only return true if the test is definitely true. All errors and uncertain results return false.
- Functions that return **Date** will return NV (with a reason code) if a definite answer cannot be computed.
- Functions that return **Interval** will return NVI (with a reason code) if a definite answer cannot be computed.
- Some integer functions return 0 as 'unknown/invalid' for example
DayOfWeek("EoT") → 0
- Where 0 is a valid in-band integer a special value **iNV** is used to indicate inability to return a sensible result.

Where strong typing is available we expect a compiler to reject most inappropriate types of function parameters. However there will be some situations where two arguments may both be a **Date** or both be an **Interval** but cannot be mixed. We leave the exact strategy for dealing with this to the good sense of the programmer. (For example Before() is specified here as taking two **Days** which need to be the same. One way of dealing with this is to specify two Before() functions with different argument patterns, one with both **Dates** the other with both **Intervals**. Alternatively the programmer might test at run-time and raise an exception if mixed arguments were detected.)

Conversion constants

Where it is not possible or desirable to use an actual calendar as the basis for date arithmetic it is necessary to convert years and months to days.

1 year = 365.25 days

1 month = 30.4375 days

2 Methods overview

Dates and intervals

Date objects and interval objects are obviously closely related (in fact they may be implemented as a single class) but are not interchangeable when it comes to usage. It is also important for a programmer to consider when it makes sense to work with the scalar quantity of days and when in DMY intervals. For example adding 30 days is not the same thing as adding one month where an entirely different algorithm is used.

Julian dates

Fully specified dates can be converted to and from Julian dates. A Julian date is just a 'serial day count'. This is ideal for finding the number of days between two events with ease, however should we want to accurately express this in YMD form we need to have an actual calendar to work with so for example we can't use the Julian day method to compute the YMD difference between 1st Feb (year unspecified) and 1st March (year unspecified). Neither can we find the number of days difference between these two floating dates *unless we provide a surrogate year*.

Rules for date arithmetic

In Part I section 6 we discussed the difficulties with adding intervals to dates. Here are the main rules:

- AR1: When adding days alone to a date:
Count according to the real calendar.
- AR2: When adding days alone to an interval select from these methods:
a Convert on the basis of 365.25 days per year and 30.4375 days per month
b Use some real base date as a datum to convert the days into YMD then apply rule R7 then subtract the base date from the result to give an interval.
- AR3: When adding months alone to a date:
Convert the days part of the date into fractions of that month. Then add the months, carrying base 12 as required. Now convert the fraction back into days according to the actual length of the new month.
- AR4: When adding months alone to an interval:
Add to months, carrying base 12.
- AR5: When adding years alone to a date:
If the month component of the date is February and the number of years to add modulo 4 is not 0 then use rule R3 with 12 times the number of months as years otherwise simply add the years.
- AR6: When adding years alone to an interval:
Simply add the year components.
- AR7: When adding a combination of YMD to a date:
Apply R5, R3 and R1 in that order.
- AR8: When adding a combination of YMD to an interval:
Convert both intervals to days then back to YMD using 365.25 days per year and 30.4375 days per month. (Compare with R2b.)

(There are variations and exceptions for floating and partially specified dates but it is probably better to leave those to the detailed method specifications.)

3 Functions taking no arguments

Date ← Today()

Returns a date object with the value of 'today' as understood by the operating system.

Date ← BoT()

Returns a date object with the value of Beginning-of-time.

Date ← EoT()

Returns a date object with the value of End-of-time.

Date ← NK()

Returns a date object with the value of Not-known.

Date ← NV()

Returns a date object with the value of Not-valid and a reason code of 0.

4 Tests taking a single day argument and returning a boolean

bool ← IsValid(Day **Day**)

Returns true if Day is any other value than **NV** or **NVI**.

bool ← IsValidDate(Day **Day**)

Returns true if Day is a **Date** with any other value than **NV**.

Returns false if Day is an **Interval**.

bool ← IsValidInterval(Day **Day**)

Returns true if Day is a **Interval** with any other value than **NVI**.

Returns false if Day is an **Date**.

bool ← IsNotKnown(Date **Date**)

Returns true if Date has the value of **NK**.

bool ← IsBoT(Date **Date**)

Returns true if Date has the value of **BoT**.

bool ← IsEoT(Date **Date**)

Returns true if Date has the value of **EoT**.

bool ← IsGiven(Date **Date**)

Returns true if Date is a *calendar day*, **NK**, **BoT** or **EoT**.

bool ← IsKnown(Date **Date**)

Returns true if Date is a *calendar day* or **BoT** or **EoT**.

bool ← IsSpecific(Date **Date**)

Returns true if Date is *fully specified* with a year, month and date-day.

bool ← IsFloating(Date **Date**)

Returns true if Date is specified without a year.

bool ← IsInterval(Day **Day**)

Returns true if Day has the **INT** signature ie. represents an interval.

bool ← IsRealPeriod(Date **Date**)

Returns true if Date is specified with either

(a) year only

(b) year and month only

bool ← IsCalendar(Date **Date**)

Returns true if Day is specified with either

(a) year only

(b) year and month only

(c) year month and day.

The following examples are *not* calendar days: "15th ", "June", "June 15th ".

Note : Intervals are not calendar days even though they may be fully specified.

bool ← HasMonth(Day **Day**)

Returns true if Day has a month specified.

bool ← HasDay(Day **Day**)

Returns true if Day has a day specified.

bool ← HasYear(Day **Day**)

Returns true if Day has a year specified.

5 Functions taking a single day argument and returning a day

Date ← FirstDay(Date **Date**)

Returns the earliest possible day for the argument. A typical example would be finding the first day of a month.

- If Date is fully specified (ie with D,M and Y) the result will be identical Date
- The constants **NV**, **BoT** and **EoT** return themselves
- The constant **NK** returns **BoT**

Date ← LastDay(Date **Date**)

Returns the latest possible day for the argument. A typical example would be finding the last day of a month.

- If Date is fully specified (ie with D,M and Y) the result will be identical with Date
- The constants **NV**, **BoT** and **EoT** return themselves
- The constant **NK** returns **EoT**

Date ← Next(Date **Date**)

Returns the next day, month or year depending on the precision of the argument. For example "March 2008" would return "April 2008" while "12th March" returns "13th March".

- Calendar dates return the next calendar date
- The constants **NV**, **NK**, **BoT** and **EoT** return themselves

Examples

Next(2007) → 2008

Next(June) → July

Next(June 2007) → July 2007

Next(28 Feb 2007) → 1 Mar 2007

Next(28 Feb 2008) → 29 Feb 2008 (Leap year)

Next(29 Feb 2008) → 1 Mar 2008

Next(15th) → 16th

Date ← Previous(Date **Date**)

Returns the previous day, month or year depending on the precision of the argument. (See Next() for details.)

Date ← MonthOnly(Date **Date**)

Return a date with just the month component of the argument

- Dates that have no month component return **NV**

- The constants **NV**, **NK**, **BoT** and **EoT** return **NV**
- See also MonthNumber().

Date ← YearMonthOnly(Date **Date**)

Return a date with the year and month components of the argument. To be a valid result the argument must have at least one of year and month specified.

- Dates that have no year component and no month component return **NV**
- The constants **NV**, **NK**, **BoT** and **EoT** return **NV**

Date ← YearOnly(Date **Date**)

Return a date with only the year component of the argument

- Dates that have no year component return **NV**
- The constants **NV**, **NK**, **BoT** and **EoT** return **NV**

See also YearNumber().

Interval ← Period(Interval **Interval**)

This has the effect of stripping any negative sign from the interval.

6 Other single argument functions

See also section 11

integer ← Signature(Day **Day**)

Return an integer representing the signature constants

NVI	→ 0
INT	→ 1
NV	→ 2
FLO	→ 3
NK	→ 4
BoT	→ 5
Cal	→ 6
EoT	→ 7

integer ← NVReasonCode(Day **Day**)

Return an integer indicating a reason (if any) associated with NV or NVI. This would most commonly be used to report on failures of FromString().

This is meant to be NV/NVI without a reason → 0

Day is not NV or NVI → **iNV**

See table 7@ for meanings of values 1 .. 7 which vary with the method that created it.

integer ← DayOfWeek(Date **Date**)

Return an integer indexing the day of the week.

Not fully specified. No date-day : → 0

Monday ... Sunday → 1 ... 7

integer ← DayOfMonth(Day **Day**)

Return an integer indexing the day of the month.

Date :

Not fully specified. No day component : → 0

"Last day of February" → 31

Interval:

Day date is 0 : → 0

integer ← MonthNumber(**Day** **Day**)

Return an integer indexing the month.

Not fully specified. No month component : → 0

January ... December → 1 ... 12

integer ← DaySerial(**Date** **Date**)

Return an integer that matches the equivalent Unix timestamp.

Date is not a fully specified date: → **iNV** [**iNV** = 7 Sep 1969 06:13:21]

Dates before 1st January 1970 : → negative

Day is 1st January 1970 : → 0

Dates after 1st January 1970 : → 1...

- Dates after 2037 will continue to increase ignoring any Unix wraparound on Jan 19th 2038.
- Dates before 1901 will continue to be more negative.

See also *ToJulianDayNumber()*.

integer ← YearNumber(**Day** **Day**)

Return an integer for the year component of the day.

Calendar day (AD) : → positive integer

Calendar day (BC) : → negative integer

Day is a floating date or constant : → **iNV**

Interval : → signed year value

float ← AsDays(**Interval** **Interval**)

Return (an approximation to) the number of days represented by the combined components of the interval. The day, month and date-day components are summed using the multipliers :

Each year is 365.25 days

Each month is 30.4375 days

- **iNV** is returned if Interval is **NVI**.
- The result will be negative if the sign of a Interval is negative.

6 Functions taking two arguments returning boolean

Unless specifically allowed do not mix **Dates** and **Intervals**.

bool ← SortsBefore(Day1 **Day**, Day2 **Day**)

Dates:

Returns true if Day1 comes before Day2 in the sorting sequence. Sorting sequence is roughly based on calendar date but with the following enhancements in increasing sort value:

- ↓ NV (lowest)
- ↓ BoT
- ↓ Date-days alone eg "15th " < "16th "
- ↓ Months and date-days only eg "March" < "March 1st " < "March 2nd "
- ↓ NK
- ↓ Days with years specified.
 - Missing month and date-day components are equivalent to 0 in a Y-M-D value system. eg "2007" < "Jan 2007" < "1st Jan 2007".
- ↓ EoT (highest)

Intervals:

Return true if the magnitude of Day1 is smaller than the magnitude of Day2.

The following sequence illustrates the sort order:-

0y 0m 0d < +1y 0m 0d < -2y 0m 0d < -5y 4m 3d

See also *Before()*

SortsBefore() and Before()

These are very different functions and it is important that their different applications are understood. In D/a/y "March" *sorts* before "March 7th" but doesn't *logically* come before it. Negating the result of SortsBefore() is equivalent to switching the order of the arguments but this *is not the case* with Before().

bool ← Before(Day1 **Day**, Day2 **Day**)

Dates:

Returns true if Day1 is *definitely* before Day2.

- Anything except **EoT** before **EoT** will return true
- **BoT** before anything except **BoT** will return true
- All other constants either as Day1 or Day2 will return false
- Non-calendar dates must be the same form to be compared

Examples

Before("April 2007", "1 April 2007") → False

Before("1st April 2007", "April 2007") → False

Before("March 2007", "1st April 2007") → True

Before("2007", "EoT") → True

Before("EoT", "EoT") → False

Before("June", "August") → True

Before("11th June", "August") → True

Before("11th June 2007", "August") → False

The last two examples show that if both days omit the year component we can compare, but if one contains a year they are no longer comparable.

Intervals:

Return true if Day1 comes 'before' Day2 as if years were masquerading as dates starting at Zero AD. Negative values come before positive as illustrated by the following sequence: -5y 4m 3d < -2y 0m 0d < 0y 0m 0d < +1y 0m 0d

See also *SortsBefore()*

bool ← *After*(Day1 **Day**, Day2 **Day**)

Returns true if Day1 is *definitely* after Day2. See *Before()* for details.

- Anything except **BoT** after **BoT** will return true
- **EoT** after anything except **EoT** will return true

bool ← *Contains*(Date1 **Date**, Date2 **Date**)

Returns true if Date2 is the same as, or within a period given by Date1.

- Returns false if any constants appear as any of the arguments.
- Date2 must have the same or greater precision than Date1.

Examples:

```
Contains("2007", "March" ) → False
Contains("2007", "March 2007") → True
Contains("2007", "4th March 2007") → True
Contains("4th March 2007", "4th March 2007") → True
Contains("March", "4th March") → True
Contains("March 2007", "4th March") → False
```

7 Functions taking two arguments returning scalar

These functions are intended to provide scalar time differences between calendar dates. **Days** do not need to be fully specified except that both arguments need to be specified to the same precision.

signed integer ← *DaysDifference*(Date1 **Date**, Date2 **Date**)

Return the number of calendar days between the two dates.

This function relies on converting each date into an integer using *JulianDayNumber()* then performing a simple subtraction.

- If applying *JulianDayNumber()* to either argument gives **iNV** the result is **iNV**...
- ... except if *only* the day component of *both* arguments is set,
- The result can be negative if Date1 is later than Date2
- Both arguments must have the same precision

Note that the *DaysGap()* function works differently and has other restrictions. This function is best only used for fully specified dates.

signed integer ← *DaysGap*(Date1 **Date**, Date2 **Date**)

See *Gap()* for details.

- **iNV** returned when *Gap()* returns **NVI**.

Signed integer ← *Overlap*(Date1 **Date**, Date2 **Date**)

Return the number of days a day object representing the least possible difference between the arguments.

@@@Image somewhere of two adjacent days with gap/span etc.

signed float ← MonthsDifference(Date1 **Date**, Date2 **Date**)

Return (an approximation) to the calendar months between the two dates.

- If any of the arguments are constants the result is **iNV**
- Both arguments must have the same precision
- 1 year is 12 months
- 1 day is 1/(365.25/12) or 1/30.4375 or 0.0328542 months.

signed float ← YearsDifference(Date1 **Date**, Date2 **Date**)

Return (an approximation) to the calendar years between the two days.

- If any of the arguments are constants the result is **iNV**
- Both arguments must have the same precision
- 1 month 1/12 years or 0.083333... years
- 1 day is 1/365.25 years or 0.00327378 years.

Integer ← SmallestOverlap(Date1 **Date**, Date2 **Date**)

This returns the smallest possible number of days for which both dates are concurrent.

If either date is fully specified this will return either 1 or 0.

If one but not both of the arguments is a floating date → **NVI**

Floating dates are converted into real dates based on the current year (and month).

@@@ **LOGIC** @@@

8 Functions taking two arguments returning Day

Day ← Add(Date1 **Date**, Interval2 **Interval**)

Add Interval2 to the Date1.

- If any of **NK**, **NV**, **NVI**, **BoT**, or **EoT** appear in either argument then return **NV** and reason code
- If Interval2 is more precise than Date1 then return **NV** and reason code
- If Date1 is floating then any year values resulting from the addition will be discarded and the result will be floating.
- @@@ Switching???

The order of adding elements is years, months then days. This can make a difference for example if we add +1y 1m 14d to 20th Jan 2008 year first we get intermediate results of 20th Jan 2009, 20th Feb 2009 and finally of 6th March. If we add the days first we get an intermediate result of 3rd Feb 2008, 3rd Mar 2008 and finally 3rd Mar 2009.

Day ← Add(Day1 **Day**, NoYears *int*, NoMonths *int*, NoYears *int*)

Convenience function encapsulating Add(**Date**,**Interval**).

Interval ← Add(Interval1 **Interval**, Interval2 **Interval**, BaseDate **Date**)

Sum the respective components of the intervals, carrying as required.

Note, there are two possible methods for carrying out this operation which arises from the inexact correlation between months and days.

Method 1 : Convert both Intervals to days using AsDays(), add the days together then convert back to Interval with IntervalFromDays()

Method 2 : Find the exact number of calendar days between BaseDate+Interval1 and BaseDate+Interval2

If BaseDate is a *calendar date* then method 2 will be used. (Imprecise BaseDates will use the earliest possible day for full specification.)

```
Add("+0y 1m 20d", "+0y 0m 20d", NV)
    70.4375 days ... 2.314.. months ... 2m 9.56d
    → +0y 2m 10d
```

```
Add("+0y 1m 20d", "+0y 0m 20d", "1 Jan 2007")
    21st Feb 2007 + 20 days ... 13th March 2007
    → +0y 2m 12d
```

```
Add("+0y 1m 20d", "+0y 0m 20d", "1 Jan 2008")
    21st Feb 2008 + 20 days ... 12th March 2008
    → +0y 2m 11d
```

Obviously there is plenty of potential for inappropriate and unintended calculation here.

Day ← Subtract(Date1 **Date**, Interval2 **Interval**)

The same as AddInterval() but with the sign of Interval reversed.

Day ← Subtract(Day1 **Day**, NoYears *int*, NoMonths *int*, NoYears *int*)

Convenience function encapsulating Subtract().

Interval ← Subtract(Interval1 **Interval**, Interval2 **Interval**, BaseDate **Date**)

Same as Add(Interval, Interval, Date) with sign of Interval2 reversed.

Date ← ConcurrencyBeginning(Date1 **Date**, Date2 **Date**)

Return the earliest date where Date1 and Date2 could be concurrent.

Return **NV** if there is no concurrency.

Date ← ConcurrencyEnding(Date1 **Date**, Date2 **Date**)

Return the latest date where Date1 and Date2 could be concurrent.

Return **NV** if there is no concurrency.

Interval ← Gap(Date1 **Date**, Date2 **Date**)

Return a day object representing the least possible difference between the arguments.

@@@

The logic is to find the latest possible date indicated by Day1 and 'subtract it' from the earliest possible date indicated by Day2.

This precision of the result is determined by the most precise argument.

- A positive result indicates there is no overlap. (Illustrated in green in the accompanying figure.)
- A negative result indicates an overlap. (Red in the accompanying figure.)
- Note the important differences between this function and DaysDifference().
 - This function will take mixed precision arguments
 - This function can return **NVI**

If one but not both of the arguments is a floating date → **NVI**

If both dates are floating then convert into real dates based on the current year (and month).

Constants appearing in arguments: (Rules in order of application)

- If either argument is **NV** the result is **NVI+?**
- If either argument is **NK** the result is **NVI+?**
- Anything (including **EoT**) and **EoT** → **NVI+?**
- **BoT** and anything (including **BoT**) → **NVI+?**
- If both arguments are the same: → **+0y0m0d**

Interval ← Span(Date1 **Date**, Date2 **Date**)

This returns the maximum possible span when the arguments are taken in either order. That is from the earliest early date to the latest late date *including both*.

This function is commonly used to calculate the duration of an activity when given a start and end date. For example:

Span("1 April 2008", "3 April 2008") → 3

Take careful note that the result is 3 days *not* 2 days.

Gap() and Span() can give very different results and vary in detail as well.

Gap("2007", "2008") → 0 days ie. 31st Dec 07 to 1st Jan 08

Span("2007", "2008") → 2 years ie. 1st Jan 07 to 31st Dec 08

Note the important difference where both arguments are the same.

The function will try to return the highest precision possible given the possibly varying precisions of the arguments.

If one but not both of the arguments is a floating date → **NVI**

Constants appearing in arguments: (Rules in order of application)

- If either argument is **NV** the result is **NVI+?**
- If either argument is **NK** the result is **NVI+?**
- Anything (including **EoT**) and **EoT** → **NVI+?**
- **BoT** and anything (including **BoT**) → **NK+?**

If both arguments are the same: → + 1 unit of maximum significance. For example:

Span("2005", "2005") → + 1y 0m 0d

Span("Aug 2005", "Aug 2005") → + 0y 1m 0d

Span("18 Aug 2005", "18 Aug 2005") → + 0y 0m 1d

If Date1 is later in the calendar than Date2 the result is negative.

Floating dates are converted into real dates based on the current year (and month).

9 Output conversion functions

32bit integer ← To32Bits(Day **Day**)

Returns the D/a/y 32bit integer encoding.

32bit integer ← ToTimestampExtended(Date **Date**)

Returns the Unix timestamp encoding as extended.

- Use this when it is convenient to overload an existing timestamp field.
- If the date is outside the extended timestamp limits given in table 8 then throw an Out_of_range exception. @@@

32bit integer ← ToTimestampConventional(Date **Date**)

Returns the Unix timestamp encoding keeping within the standard limitations. Use this when another application will be reading this.

- If the date is outside the conventional timestamp limits given in table 8 then throw an Out_of_range exception. @@@

Signed 32bit integer ← ToJulianDayNumber(Day **Day**)

Returns the Julian day number.

- If the Day isn't fully specified then return iNV

@@@ Here or somewhere else?

string ← ToString(Day **Day**, Format *string*)

Returns a string the format of which is controlled by

- The Format argument provided
- and
- The environment settings (see UseAppendix() below)

If Day is an **Interval** then report a string in the following format ignoring any Format argument.

sign years "y" space months "m" space days "d"
eg "-5y 6m 2d" or "+0y 9m 0d"

or

"InvalidInt(" nn ")"

where nn is decimal for a NVI detail code.

eg "InvalidInt(04)"

- The sign is mandatory.
- Interval results will always fit into 14 characters.
- The "y", "m" and "d" characters can be altered using an appendix.

*Comment: In a well structured class world we could have three ToString methods which would throw an exception if trying to format an **Interval** as if it was a **Date** and vice versa. Implementors may wish to do this. The reason for describing this here as a generic **Day** method (or function taking a **Day**) with the quirky handling of the unusual possibility of an **Interval** is to facilitate unambiguous debugging.*

string ← NVReasonString(Day **Day**)

Returns a string version of NVReason() adapted according to the environment settings. (see UseAppendix() below)

10 Input conversion functions

Day ← From32Bits(Data *Unsigned32bit integer*)

Inverse of To32Bits().

- If Data cannot be interpreted as a valid D/a/y object then raise an Illegal_binary_input exception. @@@

Day ← FromTimestamp(Timestamp *32bitInteger*, IsExtended *bool*)

The first argument is a Unix timestamp. The second argument indicates whether the h:m:s part may contain special information as added by ToTimestampExtended().

Day ← FromJulian(JulianDayNumber(*Signed32bit integer*)

Creates a Day object corresponding to the given Julian day number

- If there any inconsistencies encountered then raise an exception. (Details to be finalised.)

Day ← FromString(String *string*)

Convert the supplied string to a day object.

The way this is done depends on the environment settings. See UseAppendix()

If this returns **NV** then NVReasonCode() (See table 7) can be used to discover the reason in more detail.

Normally this will be used to return a Date, but we allow for the format described in ToString() to be used to create Intervals.

Date ← DateFromYMD(Y *integer*, M *integer*, D *integer*)

Create a day object representing a date from three integers.

- Illegal values will return **NV** with Reason code.
- Any argument may be zero
- Y may be negative to indicate BC

Interval ← IntervalFromYMD(Y *integer*, M *integer*, D *integer*)

Create a day object representing an interval from three integers.

- Illegal values will return NVI with Reason code.
- Any argument may be zero
- If M or D arguments are out of the normal range they will be converted into the next higher unit according to @@@ Note that the day to month conversion is approximate so that for example this function could give a different result to IntervalFromDays() which uses a real calendar.
- Only one argument may be negative. If so it is applied to the whole interval. If more than one argument is negative a Too_many_negative_arguments exception is thrown. @@@

Interval ← IntervalFromDays(D *scalar*)

Interval ← IntervalFromDays(D *scalar* , BaseDate **Date**)

Create an Interval from a given number of days.

The BaseDate argument is a *Calendar date* to use as a basis for counting from. If it is omitted Today() is used. If it not fully specified then Earliest(BaseDate) is used. If BaseDate is not a Calendar date then raise an Invalid_base_date exception @@@. [@@@ Allowable julian date range?]

BaseDate can make a difference as shown in the examples below.

IntervalFromDays(32, "1st Jan 2007") → +0y 1m 1d

IntervalFromDays(32, "1st Feb 2007") → +0y 1m 4d

IntervalFromDays(32, "1st Feb 2008") → +0y 1m 3d // leap year

11 Environment and utility functions

These functions are settings that operate globally. (Or are class methods.)

null ← SetLimitTo32Bit()

Allow the full range of date calculations supported by the 32bit encoding.

- This is to be the default condition.

null ← SetLimitToUnixConventional()

Disallow internal **Date** calculations that break the limitations on Unix timestamps. **Interval** calculations are unaffected.

null ← SetLimitToUnixExtended()

Disallow internal **Date** calculations that break the limitations on extended Unix timestamps. **Interval** calculations are unaffected.

Date ← GetLimitEarliestDate()

Return a date representing the earliest calendar date supported by the system as configured by the SetLimit...() functions.

Date ← GetLimitLatestDate()

Return a date representing the latest calendar date supported by the system as configured by the SetLimit...() functions.

Interval ← GetLimitNegativeInterval()

Return the largest negative interval supported by the system as configured by the SetLimit...() functions.⁸

Interval ← GetLimitPositiveInterval()

Return the largest positive interval supported by the system as configured by the SetLimit...() functions.⁹

integer ← ConvertUnixTimestampToDay(TimestampInt *integer*, IsExtended *boolean*)

The 32bit Unix timestamp representation is converted to a 32bit Day representation.

- IsExtended should be false when the Unix timestamp is assumed to be conventional. This filters out all the hh:mm:ss information. Alternatively, if the Unix timestamp is known to have been created as a result of the ToTimestampExtended() function the hh:mm:ss part will contain fake, but essential data and so the flag should be set true.
- This function uses To32Bits() and FromTimestamp() back-to-back and will raise the same exceptions.

integer ← ConvertDayToUnixTimestamp(DayInt *integer*, Extended *boolean*)

The 32bit representation of a Day is converted to a 32bit Unix Timestamp representation.

⁸ This shouldn't be affected by the SetLimit functions as described currently.

- This function uses ToTimestamp...() and From32Bits() back-to-back and will raise the same exceptions.
- If Extended is true use ToTimestampExtended() else use ToTimestampConventional().

integer ← UseAppendix(AppendixName *string*, LocalPath *string*, RemoteLocation *string*)

This function configures the system for input and output translations by giving the name of an 'appendix' to use.

See Part IV for details.

string ← AppxErrorDetail()

If UseAppendix() returned an error code this will give some text details.

string ← AppendixName()

Return the name of the currently selected appendix.

integer ← SetAppxValue(Label *string*, Value *string*)

Customisation of the environment to override settings obtained from an appendix. Return a status code:

- 0 : OK
- 1 : Label not recognised
- 2 : Value not suitable

This can only be used to modify keys existing in the currently loaded appendix. If a new appendix is loaded or the same is re-loaded then the change will be lost.

string ← GetAppxValue(MessageKey *string*)

Return a message indexed by the MessageKey argument. This is used to provide standard messages and interpretations depending on the appendix in use. For example

```
myHelpText = "The first month of the financial year is " +
             GetValue("M4.1");
```

This facility is also used to provide convenient access to localised error conditions that are not strictly to do with day manipulation but are likely to be encountered during form filling. For example "The date must be in the future" or "That is not a valid date here".

array of strings ← AppxShortCuts()

A sorted collection of shortcut keystrokes. The main use of this is to guide the interpretation of user input.

Each element of the array is in the form

```
<keystrokes>=<represents_key>
```

array of strings ← AppxShortCutsExplained()

A sorted collection of shortcut keystrokes with explanations. This can be used to produce a quick reference help screen for example.

Each element of the array is in the form

```
<keystrokes>=<represents_text>
```

array of strings ← AppxMonthsArray(Component *integer*)

Array of month names or abbreviations

- Component selects name style:

- 1 Full name
 - 2 Three letter display
 - 3 Two letter display
- eg `myArray = AppxMonthsArray(2); // myArray[1] -> Jan`

- First element of the array is 'no month'. Second is January.

Internationalisation

In this part we define how internationalisation can be applied to facilitate user interaction. Basically this is a set of strings in a dictionary allowing substitutions. Functions are provided to select a required dictionary, here called an Appendix, and access the specified key values.

By specifying appendices that are independent of the programming system used to implement the 'computing' functions of D/a/y's they can be written and tested once then used anywhere. Also the fiddly bits of UI programming such as for example explaining that a certain date must be in the future, can be dealt with once by the application programmer.

Input and output framework

When we have to display and obtain input we run into problems of locale and language. For example on a system I've been using for years inputting "E" into an edit box results in a display of "End of Time" and inputting "010203" gives 1st of February 2003 displayed as "1 Feb 03", but these are only conventions. Similarly if I input "30 F 03" I'll get an error "February 2003 doesn't have 30 days".

The solution to differing preferences is to provide some scope for 'plugging-in' the convention converters and language translations. Each alternative specification is called an "Appendix". Physically these will be mostly or entirely configuration files accessed by name.

1 Accessing an appendix

The scheme for accessing an appendix will be a hierarchy of

- 1 Embedded configuration data built into the d/a/y library 'at compile time'.
- 2 Local filename
- 3 URL

The API provides the following function:

integer ← *UseAppendix*(AppendixName *string*, LocalPath *string*, RemoteLocation *string*)

Attempts to locate a specified appendix by name.

AppendixName is simply a (case insensitive) IETF language tag as defined in RCF 4646. For example "en", "en-US", or "en-UK". This is used in the first instance to locate the appendix data embedded in the library code.

If LocalPath is supplied this tells the system where on the local file system to look for appendix files.

If RemoteLocation is supplied this gives the base address of a FTP URI to search for a file of the same name as defined above. For example "ftp://somewhere.org/resources/day/appendixes".

This process always looks for an appendix in the order

- 1 Embedded?
- 2 Local file system? (If argument is supplied)
- 3 Remotely available? (If argument is supplied)

and stops when an exact match is found.

There is no cascade or partial matching. If no exact match is found then nothing happens.

If a remote file is located it will attempt (subject to security constraints⁹) to copy the file to the local file system for future reference.

For ease of identification appendix files will, by convention, be named in lowercase with an extension ".dax" or ".daxu". For example `en-us.dax` and `fr.daxu`. The .daxu extension identifies files in unicode.

Function return values :

- 0 ... OK
- +ve ... Line number of basic syntax error
- 1001 ... Appendix not found (embedded)
- 1002 ... Appendix not found (local file)
- 1003 ... Appendix not found (URL)
- 1010 ... Appendix doesn't look like the right content.
 - Unsuitable size (>40Kbytes)
 - Unexpected formatting or control characters
- 1011 ... Unicode not supported
- 10xx ... *more to come*
- 2nnn ... Unsuitable value (nnn = line number)
- 3000 ... Missing required key (See AppxErrorDetail() function)

2 Logical contents overview

An appendix is divided into a number of sections

- 0 Administration
- 1 Lexicon of d/a/y specific terms such as **BoT**
- 2 Lexicon of general terms
- 3 Month names. Various output forms and shortest unique input
- 4 Day names. Various output forms.
- 5 Input parameters. Local variations for example day-month order
- 6 Output user interface. User instructions and requests for re-input.
- 7 Error messages. Advanced and technical error messages.

⁹ Clearly this is not going to be a simple issue.

Individual items are accessed by key, and possibly index. For example

`year.2` → plural of year

`DM_ORDER` → Standard day-month order

All keys in sections 0 to 5 are required although some may have blank values.

- If any keys in sections 6 and 7 are missing from a loaded appendix then they will be taken from the default (compiled-in) appendix.
- `SetAppxValue()` cannot be used to add keys that are not in the currently loaded appendix.

3 Physical layout

The file is a plain text 'configuration file' in the style familiar to generations of programmers.

- Variable number of lines of variable length delineated by LF
- Other control characters and space (0x00-0x20) form white space
- Unicode characters @@@
- Comment lines start with #
- Key/values are indicated by *key = value*
- To embed a new line into text use the token (NEWLINE)
- Blank lines or whitespace-only lines are ignored

The physical order of sections and keys is not to be relied on.

4 Appendix en-uk

It is expected that the English-Great Britain appendix will be 'compiled-in' with the library code and be the default.¹⁰ However developers should always initialise their preferred appendix even if they are firmly in the en-uk zone and not rely on it being the default. (It is expected that other common appendixes will be compiled-in also.)

5 Low level functions

Implementations will need to be able to read and validate raw text files and possibly unicode. This can be wrapped and buffered by `UseAppendix()`.

`Interpret()` and `TweakEnvironment()` give access to the parameters specified in the appendix as far as an application developer is concerned.

`FromString()` will need extensive access optimised for parsing. `ToString()` will need frequent repetitive access.

¹⁰ Amongst other things this allows for accurate conformance testing.