Title

stata.com

datetime translation — String to numeric date translation functions

Description

These functions translate dates and times recorded as strings containing human readable form (HRF) to the desired Stata internal form (SIF). See [D] **datetime** for an introduction to Stata's date and time features.

Also see *Using dates and times from other software* in [D] **datetime**.

Syntax

The string-to-numeric date and time translation functions are

Desired SIF type	String-to-numeric translation function
datetime/c	`clock(`*HRFstr*`,` *mask* [`,` *topyear*] `)`
datetime/C	`Clock(`*HRFstr*`,` *mask* [`,` *topyear*] `)`
date	`date(`*HRFstr*`,` *mask* [`,` *topyear*] `)`
weekly date	`weekly(`*HRFstr*`,` *mask* [`,` *topyear*] `)`
monthly date	`monthly(`*HRFstr*`,` *mask* [`,` *topyear*] `)`
quarterly date	`quarterly(`*HRFstr*`,` *mask* [`,` *topyear*] `)`
half-yearly date	`halfyearly(`*HRFstr*`,` *mask* [`,` *topyear*] `)`
yearly date	`yearly(`*HRFstr*`,` *mask* [`,` *topyear*] `)`

where

HRFstr is the string value (HRF) to be translated,

topyear is described in *Working with two-digit years*, below,

and *mask* specifies the order of the date and time components and is a string composed of a sequence of these elements:

Code	Meaning
`M`	month
`D`	day within month
`Y`	4-digit year
`19Y`	2-digit year to be interpreted as 19*xx*
`20Y`	2-digit year to be interpreted as 20*xx*
`h`	hour of day
`m`	minutes within hour
`s`	seconds within minute
`#`	ignore one element

Blanks are also allowed in *mask*, which can make the *mask* easier to read, but they otherwise have no significance.

Examples of *masks* include

`"MDY"` *HRFstr* contains month, day, and year, in that order.

`"MD19Y"` means the same as `"MDY"` except that *HRFstr* may contain two-digit years, and when it does, they are to be treated as if they are 4-digit years beginning with 19.

`"MDYhms"` *HRFstr* contains month, day, year, hour, minute, and second, in that order.

`"MDY hms"` means the same as `"MDYhms"`; the blank has no meaning.

`"MDY#hms"` means that one element between the year and the hour is to be ignored. For example, *HRFstr* contains values like `"1-1-2010 at 15:23:17"` or values like `"1-1-2010 at 3:23:17 PM"`.

Remarks and examples

stata.com

Remarks are presented under the following headings:

Introduction

The HRF-to-SIF translation functions are used to translate string HRF dates, such as "08/12/06", "12-8-2006", "12 Aug 06", "12aug2006 14:23", and "12 aug06 2:23 pm", to SIF. The HRF-to-SIF translation functions are typically used after importing or reading data. You read the date information into string variables and then the HRF-to-SIF functions translate the string into something Stata can use, namely, an SIF numeric variable.

You use `generate` to create the SIF variables. The translation functions are used in the expressions, such as

```
. generate double time_admitted = clock(time_admitted_str, "DMYhms")
. format time_admitted %tc
. generate date_hired = date(date_hired_str, "MDY")
. format date_hired %td
```

Every translation function—such as `clock()` and `date()` above—requires these two arguments:

1. the *HRFstr* specifying the string to be translated
2. the *mask* specifying the order in which the date and time components appear in *HRFstr*

Notes:

1. You choose the translation function `clock()`, `Clock()`, `date()`, ... according to the type of SIF value you want returned.
2. You specify the mask according to the contents of *HRFstr*.

Usually, you will want to translate an *HRFstr* containing "2006.08.13 14:23" to an SIF datetime/c or datetime/C value and translate an *HRFstr* containing "2006.08.13" to an SIF date value. If you wish, however, it can be the other way around. In that case, the detailed string would translate to an SIF date value corresponding to just the date part, 13aug2006, and the less detailed string would translate to an SIF datetime value corresponding to 13aug2006 00:00:00.000.

Specifying the mask

An argument *mask* is a string specifying the order of the date and time components in *HRFstr*. Examples of HRF date strings and the mask required to translate them include the following:

HRFstr	Corresponding mask	
01dec2006 14:22	`"DMYhm"`	
01-12-2006 14.22	`"DMYhm"`	
1dec2006 14:22	`"DMYhm"`	
1-12-2006 14:22	`"DMYhm"`	
01dec06 14:22	`"DM20Yhm"`	
01-12-06 14.22	`"DM20Yhm"`	
December 1, 2006 14:22	`"MDYhm"`	
2006 Dec 01 14:22	`"YMDhm"`	
2006-12-01 14:22	`"YMDhm"`	
2006-12-01 14:22:43	`"YMDhms"`	
2006-12-01 14:22:43.2	`"YMDhms"`	
2006-12-01 14:22:43.21	`"YMDhms"`	
2006-12-01 14:22:43.213	`"YMDhms"`	
2006-12-01 2:22:43.213 pm	`"YMDhms"`	(see note 1)
2006-12-01 2:22:43.213 pm.	`"YMDhms"`	
2006-12-01 2:22:43.213 p.m.	`"YMDhms"`	
2006-12-01 2:22:43.213 P.M.	`"YMDhms"`	
20061201 1422	`"YMDhm"`	
14:22	`"hm"`	(see note 2)
2006-12-01	`"YMD"`	
Fri Dec 01 14:22:43 CST 2006	`"#MDhms#Y"`	

Notes:

1. Nothing special needs to be included in *mask* to process a.m. and p.m. markers. When you include code `h`, the HRF-to-SIF functions automatically watch for meridian markers.
2. You specify the mask according to what is contained in *HRFstr*. If that is a subset of what the selected SIF type could record, the remaining elements are set to their defaults. `clock("14:22", "hm")` produces 01jan1960 14:22:00 and `clock("2006-12-01", "YMD")` produces 01dec2006 00:00:00. `date("jan 2006", "MY")` produces 01jan2006.

mask may include spaces so that it is more readable; the spaces have no meaning. Thus you can type

```
. generate double admit = clock(admitstr, "#MDhms#Y")
```

or type

```
. generate double admit = clock(admitstr, "# MD hms # Y")
```

and which one you use makes no difference.

How the HRF-to-SIF functions interpret the mask

The HRF-to-SIF functions apply the following rules when interpreting *HRFstr*:

1. For each HRF string to be translated, remove all punctuation except for the period separating seconds from tenths, hundredths, and thousandths of seconds. Replace removed punctuation with a space.
2. Insert a space in the string everywhere that a letter is next to a number, or vice versa.
3. Interpret the resulting elements according to *mask*.

For instance, consider the string

01dec2006 14:22

Under rule 1, the string becomes

01dec2006 14 22

Under rule 2, the string becomes

01 dec 2006 14 22

Finally, the HRF-to-SIF functions apply rule 3. If the mask is `"DMYhm"`, then the functions interpret "01" as the day, "dec" as the month, and so on.

Or consider the string

Wed Dec 01 14:22:43 CST 2006

Under rule 1, the string becomes

Wed Dec 01 14 22 43 CST 2006

Applying rule 2 does not change the string. Now rule 3 is applied. If the mask is `"#MDhms#Y"`, the translation function skips "Wed", interprets "Dec" as the month, and so on.

The `#` code serves a second purpose. If it appears at the end of the mask, it specifies that the rest of *string* is to be ignored. Consider translating the string

Wed Dec 01 14 22 43 CST 2006 patient 42

The mask code that previously worked when "patient 42" was not part of the string, `"#MDhms#Y"`, will result in a missing value in this case. The functions are careful in the translation, and if the whole string is not used, they return missing. If you end the mask in `#`, however, the functions ignore the rest of the string. Changing the mask from `"#MDhms#Y"` to `"#MDhms#Y#"` will produce the desired result.

Working with two-digit years

Consider translating the string 01-12-06 14:22, which is to be interpreted as 01dec2006 14:22:00. The translation functions provide two ways of doing this.

The first is to specify the assumed prefix in the mask. The string 01-12-06 14:22 can be read by specifying the mask `"DM20Yhm"`. If we instead wanted to interpret the year as 1906, we would specify the mask `"DM19Yhm"`. We could even interpret the year as 1806 by specifying `"DM18Yhm"`.

What if our data include 01-12-06 14:22 and include 15-06-98 11:01? We want to interpret the first year as being in 2006 and the second year as being in 1998. That is the purpose of the optional argument *topyear*:

`clock(`*string*`,` *mask* [`,` *topyear*]`)`

When you specify *topyear*, you are stating that when years in *string* are two digits, the full year is to be obtained by finding the largest year that does not exceed *topyear*. Thus you could code

```
. generate double timestamp = clock(timestr, "DMYhm", 2020)
```

The two-digit year 06 would be interpreted as 2006 because 2006 does not exceed 2020. The two-digit year 98 would be interpreted as 1998 because 2098 does exceed 2020.

Working with incomplete dates and times

The translation functions do not require that every component of the date and time be specified.

Translating 2006-12-01 with mask `"YMD"` results in 01dec2006 00:00:00.

Translating 14:22 with mask `"hm"` results in 01jan1960 14:22:00.

Translating 11-2006 with mask `"MY"` results in 01nov2006 00:00:00.

The default for a component, if not specified in the mask, is

Code	Default (if not specified)
`M`	01
`D`	01
`Y`	1960
`h`	00
`m`	00
`s`	00

Thus if you have data recording "14:22", meaning a duration of 14 hours and 22 minutes or the time 14:22 each day, you can translate it with `clock(`*HRFstr*`, "hm")`. See *Obtaining and working with durations* in [D] **datetime**.

Translating run-together dates, such as 20060125

The translation functions will translate dates and times that are run together, such as 20060125, 060125, and 20060125110215 (which is 25jan2006 11:02:15). You do not have to do anything special to translate them:

```
. display %d date("20060125", "YMD")
25jan2006
. display %td date("060125", "20YMD")
25jan2006
. display %tc clock("20060125110215", "YMDhms")
25jan2006 11:02:15
```

In a data context, you could type

```
. generate startdate = date(startdatestr, "YMD")
. generate double starttime = clock(starttimestr, "YMDhms")
```

Remember to read the original date into a string. If you mistakenly read the date as numeric, the best advice is to read the date again. Numbers such as 20060125 and 20060125110215 will be rounded unless they are stored as `doubles`.

If you mistakenly read the variables as numeric and have verified that rounding did not occur, you can convert the variable from numeric to string by using the `string()` function, which comes in one- and two-argument forms. You will need the two-argument form:

```
. generate str startdatestr = string(startdatedouble, "%10.0g")
. generate str starttimestr = string(starttimedouble, "%16.0g")
```

If you omitted the format, `string()` would produce 2.01e+07 for 20060125 and 2.01e+13 for 20060125110215. The format we used had a width that was 2 characters larger than the length of the integer number, although using a too-wide format does no harm.

Valid times

27:62:90 is an invalid time. If you try to convert 27:62:90 to a datetime value, you will obtain a missing value.

24:00:00 is also invalid. A correct time would be 00:00:00 of the next day.

In *hh*:*mm*:*ss*, the requirements are $0 \leq hh < 24$, $0 \leq mm < 60$, and $0 \leq ss < 60$, although sometimes 60 is allowed. 31dec2005 23:59:60 is an invalid datetime/c but a valid datetime/C. 31dec2005 23:59:60 includes an inserted leap second.

30dec2005 23:59:60 is invalid in both datetime encodings. 30dec2005 23:59:60 did not include an inserted leap second. A correct datetime would be 31dec2005 00:00:00.

The clock() and Clock() functions

Stata provides two separate datetime encodings that we call SIF datetime/c and SIF datetime/C and that others would call "times assuming 86,400 seconds per day" and "times adjusted for leap seconds" or, equivalently, UTC times.

The syntax of the two functions is the same:

`clock(`*HRFstr*`,` *mask* `[ ,` *topyear* `])`

`Clock(`*HRFstr*`,` *mask* `[ ,` *topyear* `])`

Function `Clock()` is nearly identical to function `clock()`, except that `Clock()` returns a datetime/C value rather than a datetime/c value. For instance,

$$\begin{aligned}\text{Noon of 23nov2010} &= \text{1,606,132,800,000 in datetime/c}\\ &= \text{1,606,132,824,000 in datetime/C}\end{aligned}$$

They differ because 24 seconds have been inserted into datetime/C between 01jan1960 and 23nov2010. Correspondingly, `Clock()` understands times in which there are leap seconds, such as 30jun1997 23:59:60. `clock()` would consider 30jun1997 23:59:60 an invalid time and so return a missing value.

Why there are two SIF datetime encodings

Stata provides two different datetime encodings, SIF datetime/c and SIF datetime/C.

SIF datetime/c assumes that there are $24 \times 60 \times 60 \times 1000$ ms per day, just as an atomic clock does. Atomic clocks count oscillations between the nucleus and the electrons of an atom and thus provide a measurement of the real passage of time.

Time of day measurements have historically been based on astronomical observation, which is a fancy way of saying that the measurements are based on looking at the sun. The sun should be at its highest point at noon, right? So however you might have kept track of time—by falling grains of sand or a wound-up spring—you would have periodically reset your clock and then gone about your business. In olden times, it was understood that the 60 seconds per minute, 60 minutes per hour, and 24 hours per day were theoretical goals that no mechanical device could reproduce accurately. These days, we have more formal definitions for measurements of time. One second is 9,192,631,770 periods of the radiation corresponding to the transition between two levels of the ground state of cesium 133. Obviously, we have better equipment than the ancients, so problem solved, right? Wrong. There are two problems: the formal definition of a second is just a little too short to use for accurately calculating the length of a day, and the Earth's rotation is slowing down.

As a result, since 1972, leap seconds have been added to atomic clocks once or twice a year to keep time measurements in synchronization with Earth's rotation. Unlike leap years, however, there is no formula for predicting when leap seconds will occur. Earth may be on average slowing down, but there is a large random component to that. As a result, leap seconds are determined by committee and announced 6 months before they are inserted. Leap seconds are added, if necessary, on the end of the day on June 30 and December 31 of the year. The exact times are designated as 23:59:60.

Unadjusted atomic clocks may accurately mark the passage of real time, but you need to understand that leap seconds are every bit as real as every other second of the year. Once a leap second is inserted, it ticks just like any other second and real things can happen during that tick.

You may have heard of terms such as GMT and UTC.

GMT is the old Greenwich Mean Time that is based on astronomical observation. GMT has been replaced by UTC.

UTC stands for coordinated universal time. It is measured by atomic clocks and is occasionally corrected for leap seconds. UTC is derived from two other times, UT1 and TAI. UT1 is the mean solar time, with which UTC is kept in sync by the occasional addition of a leap second. TAI is the atomic time on which UTC is based. TAI is a statistical combination of various atomic chronometers and even it has not ticked uniformly over its history; see http://www.ucolick.org/~sla/leapsecs/timescales.html and especially http://www.ucolick.org/~sla/leapsecs/dutc.html#TAI.

UNK is our term for the time standard most people use. UNK stands for unknown or unknowing. UNK is based on a recent time observation, probably UTC, and it just assumes that there are 86,400 seconds per day after that.

The UNK standard is adequate for many purposes, and when using it you will want to use SIF datetime/c rather than the leap second–adjusted datetime/C encoding. If you are using computer-timestamped data, however, you need to find out whether the timestamping system accounted for leap-second adjustment. Problems can arise even if you do not care about losing or gaining a second here and there.

For instance, you may import from other systems timestamp values recorded in the number of milliseconds that have passed since some agreed upon date. You may do this, but if you choose the wrong encoding scheme (choose datetime/c when you should choose datetime/C, or vice versa), more recent times will be off by 24 seconds.

To avoid such problems, you may decide to import and export data by using HRF such as "Fri Aug 18 14:05:36 CDT 2010". This method has advantages, but for datetime/C (UTC) encoding, times such as 23:59:60 are possible. Some systems will refuse to decode such times.

Stata refuses to decode 23:59:60 in the datetime/c encoding (function `clock()`) and accepts it with datetime/C (function `Clock()`). When datetime/C function `Clock()` sees a time with a 60th second, `Clock()` verifies that the time is one of the official leap seconds. Thus when translating from printable forms, try assuming datetime/c and check the result for missing values. If there are none, then you can assume your use of datetime/c was valid. If there are missing values and they are due to leap seconds and not some other error, however, you must use datetime/C `Clock()` to translate the HRF. After that, if you still want to work in datetime/c units, use function `cofC()` to translate datetime/C values into datetime/c.

If precision matters, the best way to process datetime/C data is simply to treat them that way. The inconvenience is that you cannot assume that there are 86,400 seconds per day. To obtain the duration between dates, you must subtract the two time values involved. The other difficulty has to do with dealing with dates in the future. Under the datetime/C (UTC) encoding, there is no set value for any date more than six months in the future. Below is a summary of advice.

Advice on using datetime/c and datetime/C

Stata provides two datetime encodings:

1. datetime/C, also known as UTC, which accounts for leap seconds
2. datetime/c, which ignores leap seconds (it assumes 86,400 seconds/day)

Systems vary in how they treat time variables. SAS ignores leap seconds. Oracle includes them. Stata handles either situation. Here is our advice:

- If you obtain data from a system that accounts for leap seconds, import using Stata's datetime/C encoding.
    a. If you later need to export data to a system that does not account for leap seconds, use Stata's `cofC()` function to translate time values before exporting.
    b. If you intend to `tsset` the time variable and the analysis will be at the second level or finer, just `tsset` the datetime/C variable, specifying the appropriate `delta()` if necessary—for example, `delta(1000)` for seconds.
    c. If you intend to `tsset` the time variable and the analysis will be coarser than the second level (minute, hour, etc.), create a datetime/c variable from the datetime/C variable (`generate double` *tctime* `= cofC(`*tCtime*`)`) and `tsset` that, specifying the appropriate `delta()` if necessary. You must do that because in a datetime/C variable, there are not necessarily 60 seconds in a minute; some minutes have 61 seconds.

- If you obtain data from a system that ignores leap seconds, use Stata's datetime/c encoding.
    a. If you later need to export data to a system that does account for leap seconds, use Stata's `Cofc()` function to translate time values before exporting.
    b. If you intend to `tsset` the time variable, just `tsset` it, specifying the appropriate `delta()`.

Some users prefer always to use Stata's datetime/c because `%tc` values are a little easier to work with. You can always use datetime/c if

- you do not mind having up to 1 second of error and
- you do not import or export numerical values (clock ticks) from other systems that are using leap seconds, because doing so could introduce nearly 30 seconds of error.

Remember these two things if you use datetime/C variables:

1. The number of seconds between two dates is a function of when the dates occurred. Five days from one date is not simply a matter of adding $5 \times 24 \times 60 \times 60 \times 1000$ ms. You might need to add another 1,000 ms. Three hundred sixty-five days from now might require adding 1,000 or 2,000 ms. The longer the span, the more you might have to add. The best way to add durations to datetime/C variables is to extract the components, add to them, and then reconstruct from the numerical components.
2. You cannot accurately predict datetimes more than six months into the future. We do not know what the datetime/C value of 25dec2026 00:00:00 will be because every year along the way, the International Earth Rotation Reference Systems Service (IERS) will twice announce whether a leap second will be inserted.

You can help alleviate these inconveniences. Face west and throw rocks. The benefit will be transitory only if the rocks land back on Earth, so you need to throw them really hard. We know what you are thinking, but this does not need to be a coordinated effort.

Determining when leap seconds occurred

Stata system file `leapseconds.maint` lists the dates on which leap seconds occurred. The file is updated periodically (see [R] **update**; the file is updated when you `update all`), and Stata's datetime/C functions access the file to know when leap seconds occurred.

You can access it, too. To view the file, type

```
. viewsource leapseconds.maint
```

The date() function

The syntax of the `date()` function is

`date(`*string*`,` *mask* [`,` *topyear*]`)`

The `date()` function is identical to `clock()` except that `date()` returns an SIF date value rather than a datetime value. The `date()` function is the same as `dofc(clock())`.

`daily()` is a synonym for `date()`.

The other translation functions

The other translation functions are

SIF type	HRF-to-SIF translation function
weekly date	`weekly(`*HRFstr*`,` *mask* [`,` *topyear*]`)`
monthly date	`monthly(`*HRFstr*`,` *mask* [`,` *topyear*]`)`
quarterly date	`quarterly(`*HRFstr*`,` *mask* [`,` *topyear*]`)`
half-yearly date	`halfyearly(`*HRFstr*`,` *mask* [`,` *topyear*]`)`

HRFstr is the value to be translated.
mask specifies the order of the components.
topyear is described in *Working with two-digit years*, above.

These functions are rarely used because data seldom arrive in these formats.

Each of the functions translates a pair of numbers: `weekly()` translates a year and a week number (1–52), `monthly()` translates a year and a month number (1–12), `quarterly()` translates a year and a quarter number (1–4), and `halfyearly()` translates a year and a half number (1–2).

The masks allowed are far more limited than the masks for `clock()`, `Clock()`, and `date()`:

Code	Meaning
`Y`	4-digit year
`19Y`	2-digit year to be interpreted as 19*xx*
`20Y`	2-digit year to be interpreted as 20*xx*
`W`	week number (`weekly()` only)
`M`	month number (`monthly()` only)
`Q`	quarter number (`quarterly()` only)
`H`	half-year number (`halfyearly()` only)

The pair of numbers to be translated must be separated by a space or punctuation.
No extra characters are allowed.

Reference

Rajbhandari, A. 2015. A tour of datetime in Stata. *The Stata Blog: Not Elsewhere Classified.* http://blog.stata.com/2015/12/17/a-tour-of-datetime-in-stata-i/.

Also see

[D] **datetime** — Date and time values and variables

[D] **datetime business calendars** — Business calendars

[D] **datetime display formats** — Display formats for dates and times