

24 Working with dates and times

Contents

- 24.1 Overview
- 24.2 Inputting dates and times
- 24.3 Displaying dates and times
- 24.4 Typing dates and times (datetime literals)
- 24.5 Extracting components of dates and times
- 24.6 Converting between date and time values
- 24.7 Business dates and calendars
- 24.8 References

24.1 Overview

Full documentation on Stata's date and time capabilities—including documentation on relevant functions and display formats—can be found in [D] [datetime](#).

Stata can work with dates such as 21nov2006, with times such as 13:42:02.213, and with dates and times such as 21nov2006 13:42:02.213. You can write these dates and times however you wish, such as 11/21/2006, November 21, 2006, and 1:42 p.m.

Stata stores dates, times, and dates and times as integers such as $-4,102$, 0 , 82 , $4,227$, and $1,479,735,745,213$. It works like this:

1. You begin with the datetime variables in your data however they are recorded, such as 21nov2006 or 11/21/2006 or November 21, 2006, or 13:42:02.213 or 1:42 p.m. The original values are usually best stored in string variables.
2. Using functions we will describe below, you translate the original into the integers that Stata understands and store those values in a new variable.
3. You specify the appropriate display format for the new variable so that, rather than displaying as the integer values that they are, they display in a way you can read them such as 21nov2006 or 11/21/2006 or November 21, 2006, or 13:42:02.213 or 1:42 p.m.

The numeric encoding that Stata uses is centered on the first millisecond of 01jan1960, that is, 01jan1960 00:00:00.000. That datetime is assigned integer value 0.

Integer value 1 is the millisecond after that: 01jan1960 00:00:00.001.

Integer value -1 is the millisecond before that: 31dec1959 23:59:59.999.

By that logic, 21nov2006 13:42:02.213 is integer value $1,479,735,722,213$ or, at least, it is if we ignore the [leap seconds](#) that have been inserted to keep clocks in alignment with astronomical observation. If we account for leap seconds, 21nov2006 13:42:02.213 would be 23 seconds later, namely, $1,479,735,745,213$. Stata can work either way.

Obtaining the number of milliseconds associated with a datetime is easy because Stata provides functions that translate things like 21nov2006 13:42:02.213 (written however you wish) to $1,479,735,722,213$ or $1,479,735,745,213$.

Just remember, Stata records datetime values as the number of milliseconds since the first millisecond of 01jan1960.

Stata records pure time values (clock times independent of date) the same way. Rather than thinking of the numeric value as the number of milliseconds since 01jan1960, however, think of it as the number of milliseconds since the beginning of the day. For instance, at 2 p.m. every day, the airplane takes off from Houston for London. The numeric value associated with 2 p.m. is 50,400,000 because there are that many milliseconds between the beginning of the day (00:00:00.000) and 2 p.m.

The advantage of thinking this way is that you can add dates and times. What is the datetime value for when the plane takes off on 21nov2006? Well, 21nov2006 00:00:00.000 is 1,479,686,400,000 (ignoring leap seconds), and $1,479,686,400,000 + 50,400,000$ is 1,479,736,800,000.

Subtracting datetime values is useful, too. How many hours are there between 21jan1952 7:23 a.m. and 21nov2006 3:14 p.m.? Answer: $(1,479,741,240,000 - (-250,706,220,000))/3,600,000 = 480,679.85$ hours.

Variables that record the number of milliseconds since 01jan1960 and ignore leap seconds are called %tc variables.

Variables that record the number of milliseconds since 01jan1960 and account for leap seconds are called %tC variables.

Stata has seven other kinds of %t variables.

In many applications, calendar dates by themselves are sufficient. The applicant was hired on 15jan2006, for instance. You could use a %tc variable to record that value, assigning some arbitrary time that you would ignore, but it is better and easier to use a %td variable. In %td variables, 0 still corresponds to 01jan1960, but a unit change now represents an entire day rather than a millisecond. The value 1 represents 02jan1960. The value -1 represents 31dec1959. When you subtract %td variables, you obtain the number of days between dates.

In a financial application, you might use %tq variables. In %tq, 0 represents the first quarter of 1960, 1 represents the second quarter, and -1 represents the last quarter of 1959. When you subtract %tq variables, you obtain the number of quarters between dates.

Stata understands nine %t formats:

| Format | Base | Units | Comment |
|--------|-----------|--------------|-------------------------------------|
| %tc | 01jan1960 | milliseconds | ignores leap seconds |
| %tC | 01jan1960 | milliseconds | accounts for leap seconds |
| %td | 01jan1960 | days | calendar date format |
| %tw | 1960-w1 | weeks | 52nd week may have more than 7 days |
| %tm | jan1960 | months | calendar month format |
| %tq | 1960-q1 | quarters | financial quarter |
| %th | 1960-h1 | half-years | 1 half-year = 2 quarters |
| %ty | 0 A.D | year | 1960 means year 1960 |
| %tb | - | days | user defined |

All formats except %ty and %tb are based on the beginning of January 1960. The value 0 means the first millisecond, day, week, month, quarter, or half-year of 1960, depending on format. The value 1 is the millisecond, day, week, month, quarter, or half-year after that. The value -1 is the millisecond, day, week, month, quarter, or half-year before that.

Stata's %ty format records years as numeric values and it codes them the natural way: rather than 0 meaning 1960, 1960 means 1960, and so 2006 also means 2006.

24.2 Inputting dates and times

Dates and time variables are best read as strings. You then use one of the string-to-numeric conversion functions to convert the string to an appropriate `%t` value:

| Format | String-to-numeric conversion function |
|------------------|---------------------------------------|
| <code>%tc</code> | <code>clock(string, mask)</code> |
| <code>%tC</code> | <code>Clock(string, mask)</code> |
| <code>%td</code> | <code>date(string, mask)</code> |
| <code>%tw</code> | <code>weekly(string, mask)</code> |
| <code>%tm</code> | <code>monthly(string, mask)</code> |
| <code>%tq</code> | <code>quarterly(string, mask)</code> |
| <code>%th</code> | <code>halfyearly(string, mask)</code> |
| <code>%ty</code> | <code>yearly(string, mask)</code> |

The full documentation of these functions can be found in [D] [datetime translation](#).

In the above table, *string* is the string variable to be translated, and *mask* specifies the order in which the components of the date and/or time appear in *string*. For instance, the *mask* in `%td` function `date()` is made up of the letters M, D, and Y.

`date(string, "DMY")` specifies *string* contains dates in the order of day, month, year. With that specification, `date()` can translate 21nov2006, 21 November 2006, 21-11-2006, 21112006, and other strings that contain dates in the order day, month, year.

`date(string, "MDY")` specifies *string* contains dates in the order of month, day, year. With that specification, `date()` can translate November 21, 2006, 11/21/2006, 11212006, and other strings that contain dates in the order month, day, year.

You can specify a two-digit prefix in front of Y to handle two-digit years. `date(string, "MD19Y")` specifies *string* contains dates in the order of month, day, and year, and that if the year contains only two digits, it is to be prefixed with 19. With that specification, `date()` could not only translate November 21, 2006, 11/21/2006, and 11212006, but also Feb. 15 '98, 2/15/98, and 21598. (There is another way to deal with two-digit years so that 98 becomes 1998 while 06 becomes 2006; it involves specifying an optional third argument. See [Working with two-digit years](#) in [D] [datetime translation](#).)

Let's consider some `%td` data. We have the following raw-data file:

```
----- begin bdays.raw -----
Bill  21 Jan 1952  22
May   11 Jul 1948  18
Sam   12 Nov 1960  25
Kay   9 Aug 1975   16
----- end bdays.raw -----
```

We could read these data by typing

```
. infix str name 1-5 str bday 7-17 x 20-21 using bdays
(4 observations read)
```

We read the date not as three separate variables but as one variable. Variable `bday` contains the entire date:

```
. list
```

| | name | bday | | | x |
|----|------|------|-----|------|----|
| 1. | Bill | 21 | Jan | 1952 | 22 |
| 2. | May | 11 | Jul | 1948 | 18 |
| 3. | Sam | 12 | Nov | 1960 | 25 |
| 4. | Kay | 9 | Aug | 1975 | 16 |

The data look fine, but if we set about using them, we would quickly discover there is not much we could do with variable `bday`. Variable `bday` looks like a date, but it is just a string. We need to turn `bday` into a `%t` variable that Stata understands:

```
. gen birthday = date(bday, "DMY")
. list
```

| | name | bday | | | x | birthday |
|----|------|------|-----|------|----|----------|
| 1. | Bill | 21 | Jan | 1952 | 22 | -2902 |
| 2. | May | 11 | Jul | 1948 | 18 | -4191 |
| 3. | Sam | 12 | Nov | 1960 | 25 | 316 |
| 4. | Kay | 9 | Aug | 1975 | 16 | 5699 |

New variable `birthday` is a `%td` variable. The problem now is that, whereas the new variable is perfectly understandable to Stata, it is not understandable to us. Naturally enough, a `%td` variable needs a `%td` format:

```
. format birthday %td
. list
```

| | name | bday | | | x | birthday |
|----|------|------|-----|------|----|-----------|
| 1. | Bill | 21 | Jan | 1952 | 22 | 21jan1952 |
| 2. | May | 11 | Jul | 1948 | 18 | 11jul1948 |
| 3. | Sam | 12 | Nov | 1960 | 25 | 12nov1960 |
| 4. | Kay | 9 | Aug | 1975 | 16 | 09aug1975 |

Using our new `%td` variable, we can create a variable recording how old each of these subjects was on 01jan2000:

```
. gen age2000 = (td(1jan2000)-birthday)/365.25
. list
```

| | name | bday | | | x | birthday | age2000 |
|----|------|------|-----|------|----|-----------|----------|
| 1. | Bill | 21 | Jan | 1952 | 22 | 21jan1952 | 47.94524 |
| 2. | May | 11 | Jul | 1948 | 18 | 11jul1948 | 51.47433 |
| 3. | Sam | 12 | Nov | 1960 | 25 | 12nov1960 | 39.13484 |
| 4. | Kay | 9 | Aug | 1975 | 16 | 09aug1975 | 24.39699 |

`td()` is a function that makes it easy to type `%td` dates. There are also functions `tc()`, `tC()`, `tw()`, `tm()`, `tq()`, and `th()` for the other `%t` formats; see [\[D\] datetime](#).

Let's consider one more example. We have the following data:

```
. use http://www.stata-press.com/data/r13/datexmpl2
. list
```

| | id | timestamp | action |
|----|------|------------------------------|--------|
| 1. | 1001 | Tue Nov 14 08:59:43 CST 2006 | 15 |
| 2. | 1002 | Wed Nov 15 07:36:49 CST 2006 | 15 |
| 3. | 1003 | Wed Nov 15 09:21:07 CST 2006 | 11 |
| 4. | 1002 | Wed Nov 15 14:57:36 CST 2006 | 16 |
| 5. | 1005 | Thu Nov 16 08:22:53 CST 2006 | 12 |
| 6. | 1001 | Thu Nov 16 08:36:44 CST 2006 | 16 |

Variable `timestamp` is a string which we want to convert to a `%tc` variable. From the table above, we know we will use function `clock()`. The *mask* in `clock()` uses the letters D, M, Y, and h, m, s, which specify the order of the day, month, year and hours, minutes, seconds. `timestamp` contains more than that and so cannot directly be converted using `clock()`. First, we must create a variable that `clock()` understands:

```
. gen str ts = substr(timestamp, 5, 15) + " " + substr(timestamp, 25, 4)
. list ts
```

| | ts |
|----|----------------------|
| 1. | Nov 14 08:59:43 2006 |
| 2. | Nov 15 07:36:49 2006 |
| 3. | Nov 15 09:21:07 2006 |
| 4. | Nov 15 14:57:36 2006 |
| 5. | Nov 16 08:22:53 2006 |
| 6. | Nov 16 08:36:44 2006 |

New variable `ts` can be translated using `clock(ts, "MD hms Y")`. "MD hms Y" specifies that the order of the components in `ts` is month, day, hours, minutes, seconds, and year. There is no meaning to the spaces; we could just as well have specified `clock(ts, "MDhmsY")`. You can specify spaces when they help to make what you type more readable.

Because `%tc` values can be so large, whenever you use the function `clock()`, you must store the results in a double, as we do below:

```
. gen double dt = clock(ts, "MD hms Y")
. list id dt action
```

| | id | dt | action |
|----|------|-----------|--------|
| 1. | 1001 | 1.479e+12 | 15 |
| 2. | 1002 | 1.479e+12 | 15 |
| 3. | 1003 | 1.479e+12 | 11 |
| 4. | 1002 | 1.479e+12 | 16 |
| 5. | 1005 | 1.479e+12 | 12 |
| 6. | 1001 | 1.479e+12 | 16 |

Don't panic. New variable `dt` contains numeric values, and large ones, which is why it was so important that we stored the values as doubles. That output above just shows us what a `%tc` variable

looks like with default formatting. If we wanted to see the numeric values better, we could change `dt` to have a `%20.0gc` format. We would then see that the first value is 1,479,113,983,000, the second 1,479,195,409,000, and so on. We will not do that. Instead, we will put a `%tc` format on our `%tc` variable:

```
. format dt %tc
. list id dt action
```

| | id | dt | action |
|----|------|--------------------|--------|
| 1. | 1001 | 14nov2006 08:59:43 | 15 |
| 2. | 1002 | 15nov2006 07:36:49 | 15 |
| 3. | 1003 | 15nov2006 09:21:07 | 11 |
| 4. | 1002 | 15nov2006 14:57:36 | 16 |
| 5. | 1005 | 16nov2006 08:22:53 | 12 |
| 6. | 1001 | 16nov2006 08:36:44 | 16 |

Variable `dt` is a variable we can use. Say we wanted to know how many hours it had been since the previous action:

```
. sort dt
. gen hours = hours(dt - dt[_n-1])
(1 missing value generated)
. format hours %9.2f
. list id dt action hours
```

| | id | dt | action | hours |
|----|------|--------------------|--------|-------|
| 1. | 1001 | 14nov2006 08:59:43 | 15 | . |
| 2. | 1002 | 15nov2006 07:36:49 | 15 | 22.62 |
| 3. | 1003 | 15nov2006 09:21:07 | 11 | 1.74 |
| 4. | 1002 | 15nov2006 14:57:36 | 16 | 5.61 |
| 5. | 1005 | 16nov2006 08:22:53 | 12 | 17.42 |
| 6. | 1001 | 16nov2006 08:36:44 | 16 | 0.23 |

We subtracted the previous value of `dt` from `dt`, which results in the number of milliseconds. Converting milliseconds to hours is easy enough: we just have to divide by $60 \times 60 \times 1,000 = 3,600,000$. It is easy to forget or mistype that constant, so we used Stata's `hours()` function, which converts milliseconds to hours. `hours()`, and other useful functions, are documented in [D] [datetime](#).

24.3 Displaying dates and times

A `%td` variable should have a `%td` format, a `%tc` variable should have a `%tc` format, and so on for all the other `%t` variable–format pairs. You do that by typing `format varname %td`, `format varname %tc`, etc.

Formats `%tc`, `%tC`, `%td`, `%tw`, `%tm`, `%tq`, `%th`, and `%ty` are called the default `%t` formats. By specifying codes following them, you can control how the variable is to be displayed.

In the previous example, we started with a string variable that contained a time stamp and looked like “Tue Nov 14 08:59:43 CST 2006”. After creating a `%tc` variable from it, and putting the default `%tc` format on it, our datetimes looked like “14nov2006 08:59:43”. Below we specify a `%tc` format that makes our new variable look just like the original:

```
. format dt %tcDay_Mon_DD_HH:MM:SS!C!S!T_CCYY
. list id dt action hours
```

| | id | dt | action | hours |
|----|------|------------------------------|--------|-------|
| 1. | 1001 | Tue Nov 14 08:59:43 CST 2006 | 15 | . |
| 2. | 1002 | Wed Nov 15 07:36:49 CST 2006 | 15 | 22.62 |
| 3. | 1003 | Wed Nov 15 09:21:07 CST 2006 | 11 | 1.74 |
| 4. | 1002 | Wed Nov 15 14:57:36 CST 2006 | 16 | 5.61 |
| 5. | 1005 | Thu Nov 16 08:22:53 CST 2006 | 12 | 17.42 |
| 6. | 1001 | Thu Nov 16 08:36:44 CST 2006 | 16 | 0.23 |

%t display formats are documented in [D] [datetime display formats](#).

24.4 Typing dates and times (datetime literals)

You will sometimes need to type dates and times in expressions. When we needed to calculate the age of subjects as of 01jan2000 in a previous example, for instance, we typed,

```
. gen age2000 = (td(1jan2000)-birthday)/365.25
```

although we could just as well have typed,

```
. gen age2000 = (14610-birthday)/365.25
```

because 14,610 is the %td value of 01jan2000. Typing `td(1jan2000)` is easier and less error-prone.

Similarly, if we needed 10:55 a.m. on 01jan1960 as a %tc value, rather than typing 39,300,000, we could type `tc(01jan1960 10:55)`. See [Conveniently typing SIF values](#) in [D] [datetime](#) for details.

24.5 Extracting components of dates and times

Once you have a %t variable, you can use the extraction functions to obtain components of the variable. For instance, the following functions are appropriate for use with %td variables:

| | |
|-----------------------------|--|
| <code>year(date)</code> | returns four-digit year; e.g., 1980, 2002 |
| <code>month(date)</code> | returns month; 1, 2, . . . , 12 |
| <code>day(date)</code> | returns day within month; 1, 2, . . . , 31 |
| <code>halfyear(date)</code> | returns the half of year; 1 or 2 |
| <code>quarter(date)</code> | returns quarter of year; 1, 2, 3, or 4 |
| <code>week(date)</code> | returns week of year; 1, 2, . . . , 52 |
| <code>dow(date)</code> | returns day of week; 0, 1, . . . , 6; 0 = Sunday |
| <code>doy(date)</code> | returns day of year; 1, 2, . . . , 366 |

There are other functions useful with other %t variables. See [Extracting time-of-day components from SIFs](#) and [Extracting date components from SIFs](#) in [D] [datetime](#).

24.6 Converting between date and time values

You can convert between %t values. For instance, the `cofd()` function converts a %td value to a %tc value. `cofd()` of 17,126 (21nov2006) returns 1,479,686,400,000 (21nov2006 00:00:00). Function `dofc()` of 1,479,736,920,000 (21nov2006 14:02) returns 17,126 (21nov2006).

There are other functions for converting between other %t values; see [SIF-to-SIF conversion](#) in [D] [datetime](#).

24.7 Business dates and calendars

In addition to the built-in date types above, such as `%tc` and `%td`, Stata provides a type you can define, denoted as `%tb` and called business dates and calendars.

A business calendar is like an ordinary calendar with some dates crossed out. The crossed-out dates correspond to the dates on which the business is closed:

| November 2011 | | | | | | |
|---------------|----|----|----|----|----|----|
| Su | Mo | Tu | We | Th | Fr | Sa |
| | | 1 | 2 | 3 | 4 | X |
| X | 7 | 8 | 9 | 10 | 11 | X |
| X | 14 | 15 | 16 | 17 | 18 | 19 |
| X | 21 | 22 | 23 | X | 25 | X |
| X | 28 | 29 | 30 | | | |

With respect to a business date, *yesterday* is the last day the business was open, and *tomorrow* is the next day the business will be open.

Consider `date = 25nov2011`. If `date` is a regular (`%td`) date variable,

$$yesterday = date - 1 = 24nov2011$$

$$tomorrow = date + 1 = 26nov2011$$

If `date` is a business (`%tb`) date variable,

$$yesterday = date - 1 = 23nov2011$$

$$tomorrow = date + 1 = 28nov2011$$

That is important because variables containing dates are often used with Stata's lag and lead operators; see [U] 13.9 **Time-series operators**. If variable `trading_date` is an ordinary `%td` variable, then `L.trading_date` really is yesterday and `F.trading_date` really is tomorrow. But if `trading_date` has an appropriately defined `%tb` format, `L.trading_date` is the previous trading date and `F.trading_date` is the next trading date.

You can use `bcal create` to create a business calendar based on the current dataset. Alternatively, you can create a file named `calname.stbcal`, such as `nyse.stbcal`. After that, Stata deeply understands the new format `%tbnyse`. For more information, see [D] **datetime business calendars**.

Business dates work just like regular dates, it is just that some dates are crossed out.

24.8 References

- Cox, N. J. 2006. *Speaking Stata: Time of day*. *Stata Journal* 6: 124–137.
- . 2010. *Stata tip 68: Week assumptions*. *Stata Journal* 10: 682–685.
- . 2012. *Stata tip 111: More on working with weeks*. *Stata Journal* 12: 565–569.
- Samuels, S. J., and N. J. Cox. 2012. *Stata tip 105: Daily dates with missing days*. *Stata Journal* 12: 159–161.