AgEcon SEARCH
RESEARCH IN AGRICULTURAL & APPLIED ECONOMICS

The World's Largest Open Access Agricultural & Applied Economics Digital Library

This document is discoverable and free to researchers across the globe due to the work of AgEcon Search.

Help ensure our sustainability.

Give to AgEcon Search

AgEcon Search
http://ageconsearch.umn.edu
aesearch@umn.edu

*Papers downloaded from **AgEcon Search** may be used for non-commercial purposes and personal study only. No other use, including posting to another Internet site, is permitted without permission from the copyright owner (not AgEcon Search), or as allowed under the provisions of Fair Use, U.S. Copyright Act, Title 17 U.S.C.*

The Stata Journal

Editor
H. Joseph Newton
Department of Statistics
Texas A & M University
College Station, Texas 77843
979-845-3142; FAX 979-845-3144
jnewton@stata-journal.com

Editor
Nicholas J. Cox
Geography Department
Durham University
South Road
Durham City DH1 3LE UK
n.j.cox@stata-journal.com

Associate Editors

Christopher Baum
Boston College

Rino Bellocco
Karolinska Institutet, Sweden and
Univ. degli Studi di Milano-Bicocca, Italy

David Clayton
Cambridge Inst. for Medical Research

Mario A. Cleves
Univ. of Arkansas for Medical Sciences

William D. Dupont
Vanderbilt University

Charles Franklin
University of Wisconsin, Madison

Joanne M. Garrett
University of North Carolina

Allan Gregory
Queen's University

James Hardin
University of South Carolina

Ben Jann
ETH Zurich, Switzerland

Stephen Jenkins
University of Essex

Ulrich Kohler
WZB, Berlin

Jens Lauritsen
Odense University Hospital

Stanley Lemeshow
Ohio State University

J. Scott Long
Indiana University

Thomas Lumley
University of Washington, Seattle

Roger Newson
Imperial College, London

Marcello Pagano
Harvard School of Public Health

Sophia Rabe-Hesketh
University of California, Berkeley

J. Patrick Royston
MRC Clinical Trials Unit, London

Philip Ryan
University of Adelaide

Mark E. Schaffer
Heriot-Watt University, Edinburgh

Jeroen Weesie
Utrecht University

Nicholas J. G. Winter
Cornell University

Jeffrey Wooldridge
Michigan State University

Stata Press Production Manager Lisa Gilmore
Stata Press Copy Editors Gabe Waggoner, John Williams

Copyright Statement: The Stata Journal and the contents of the supporting files (programs, datasets, and help files) are copyright © by StataCorp LP. The contents of the supporting files (programs, datasets, and help files) may be copied or reproduced by any means whatsoever, in whole or in part, as long as any copy or reproduction includes attribution to both (1) the author and (2) the Stata Journal.

The articles appearing in the Stata Journal may be copied or reproduced as printed copies, in whole or in part, as long as any copy or reproduction includes attribution to both (1) the author and (2) the Stata Journal.

Written permission must be obtained from StataCorp if you wish to make electronic copies of the insertions. This precludes placing electronic copies of the Stata Journal, in whole or in part, on publicly accessible web sites, fileservers, or other locations where the copy may be accessed by anyone other than the subscriber.

Users of any of the software, ideas, data, or other materials published in the Stata Journal or the supporting files understand that such use is made without warranty of any kind, by either the Stata Journal, the author, or StataCorp. In particular, there is no warranty of fitness of purpose or merchantability, nor for special, incidental, or consequential damages such as loss of profits. The purpose of the Stata Journal is to promote free communication among Stata users.

The *Stata Journal*, electronic version (ISSN 1536-8734) is a publication of Stata Press, and Stata is a registered trademark of StataCorp LP.

The Stata Journal (2006)
6, Number 1, pp. 124–137

Speaking Stata: Time of day

Nicholas J. Cox
Department of Geography
Durham University
Durham City, UK
n.j.cox@durham.ac.uk

Abstract. Many problems in statistical analysis include time-of-day variables, but Stata offers limited support for time-of-day calculations. Support is needed for dates with times, times alone, and durations or timings. This article presents two new programs as general utilities to convert back and forth between string and numeric representations.

Keywords: dm0018, ntimeofday, stimeofday, time of day, time series, calendar

1 Introduction

Many problems in statistical analysis include time-of-day variables. Economists, medical statisticians, and environmental scientists are just three groups who often deal with such data. However, Stata offers limited official support for time-of-day calculations. Here I will discuss the general problems that time of day poses and introduce two utilities for conversion back and forth between numeric representations (in some units such as days or seconds) and composite string representations showing the information separated into elements (and often decorated with units or other information).

This column is the first in a series with the general theme of circular arguments. Time of day is a circular scale, as one midnight or noon is followed by another a day later. Later columns will examine seasonal data and directions (angles, orientations) defined on more general circular outcome spaces.

2 From time to time of day

Support for time and time variables has been part of official Stata for several versions, including

Formats. Daily date and other date formats (`%d` and `%t` formats).

Functions. Various functions for date definitions and extractions (`date()`, `d()`, `mdy()` and many others).

`tsset`. Allows the declaration of datasets as individual time series or longitudinal or panel data.

Operators. Time-series operators such as `L.` and `D.` allow the automation of lagging, differencing, etc.

© 2006 StataCorp LP

Specific commands. Various commands intended mostly or entirely for time series, whether individual time series or longitudinal or panel data, as surveyed in the [TS] and [XT] manuals.

If most or all of this is new to you, start with [U] **24 Dealing with dates**.

Use of single time variables, let alone of `tsset`, is not compulsory in Stata for all time-series manipulations. When working with daily rainfall data, I have often used a date definition based on three separate variables, for year, month, and day of month, or one based on two separate variables, for year and day of year. This system is less wasteful than may at first sight appear, as year and month (or day of year) variables are essential for many analyses. Nor is the sort order defined by year, month, and day in turn the only natural sort order. An order defined by month, day, and year has several uses when examining seasonal fluctuations.

Some of the facilities for time data are necessary because time has a direction defining a unique order of observations. Most of the programming complexities arise, however, because of the calendar. Richards (1998) gives an entertaining and informative survey of the variety of calendars. Even the Western (Gregorian) calendar that most readers will regard as standard has a complicated structure of years, months, weeks, and days, including leap year corrections. Some data, especially in finance or economics, also refer to half-years or quarters. There are also alternative calendars; weekends; public holidays and other special days; years or quarters linked to origins other than 1 January; and other details not explicitly supported in Stata. Calendar problems continue to pose a variety of computing challenges. Reingold and Dershowitz (2001) is the standard reference.

Data do often arrive using some calendar convention. But even if you receive a Stata dataset in which time has been set up according to Stata's own conventions, you must be able to report results for time data using the standard calendar.

Put together, these facilities for time variables ease management and analysis of data recorded on various resolutions from yearly to daily. Stata's support does not apply to years before 100 CE, so any archaeologists, ancient historians, or climatologists with interests around or before that time would need to fend for themselves. Evidently, some scientists do work with data over even longer time spans. Geologists and astronomers routinely think in thousands, indeed millions, of years. However, they can avoid the idiosyncrasies of the calendar. Geologists estimating the date of a meteor impact do not trouble themselves with whether it was during a leap year: that resolution is impossible to attain. At the other end of the spectrum, some scientists do work on fast reactions in which times may be measured in nanoseconds. As with very long time scales, it is not clear that any special support in Stata is needed for time over such resolutions.

One feature that is almost missing from official Stata, however, is support for time of day. As I type this sentence, a little display in the corner of my computer screen shows 17:16, which you will recognize as a 24-hour time. Many datasets go beyond daily resolution to register when an event occurred or when some property was recorded.

As with the calendar, the complexity in handling time of day arises from using hours, minutes, and seconds. Although familiar to all readers, this system emerged only in stages over a long period. The brief notes here are based mostly on details given by Richards (1998, 44) and Whitrow (1975, 62). The Egyptians divided the day into 24 periods, but astronomers from Hipparchus (second century BCE) onward took the further step of defining those periods to have equal length. For several centuries more, most other subdivisions of the day were based on subdivisions of daylight and nighttime periods, which thus varied seasonally. Division of hours into 60 minutes was introduced by the astronomer Ptolemy in the second century CE, but seconds came only much later. Whitrow reports seconds' being used for theoretical calculations in 1345—before they were used as units of measurement.

I said that support for time of day is "almost" missing because the documentation of the official `split` command introduced in Stata 8 does include an example showing how time data in *hh*`:`*mm*`:`*ss* form may be split into component variables. More generally, various general Stata commands or functions can be useful if you are dealing with times of day, as will be discussed in the next section. Other than that, support for time of day is limited to various user-written programs. Perhaps Stata groups in financial institutions or government agencies have privately developed such programs, but all those known to me are accessible from the SSC archive by using the `ssc` command; see [R] **ssc**. The programs concerned are `fodstr`, `fod2str`, and `mkfrac` by William Gould and various `egen` functions written by Christopher F. Baum and myself.

For me, the need for time-of-day support arises mostly from looking at data from tipping-bucket rain gauges. People have been measuring rainfall for thousands of years. At root, it is easy. Just after a storm you look in a pot or a bucket that was left out in the open and see how deep the water is. Naturally there are many complications (e.g., splashing, evaporation, snow), but the instrument design and the measurement protocol have moved on accordingly. Tipping-bucket rain gauges all center on recording the times by which successive units of rainfall have accumulated. The name comes from tipping and emptying of a little bucket each time it is full, meaning that it is holding the specified unit. One great attraction of modern designs is that gauges can be left in wild or remote areas without needing much attention from scientists.

This column can be read on two levels. First, it introduces two new commands, `ntimeofday` and `stimeofday`, for basic time-of-day manipulations. The first letters `n` and `s` flag a major distinction. `n` stands for numeric, and `ntimeofday` is for generating numeric time-of-day variables from string time-of-day variables. `s` stands for string, and `stimeofday` is for generating string time-of-day variables from numeric time-of-day variables. You may be interested primarily to know whether these commands provide the functionality you need.

Second, and in many ways more interesting, are some issues that arise in implementing these programs, ranging from user needs to syntax choices. In brief, some support for time of day is within the reach of the user–programmer, but some functionality requires StataCorp action.

3 Needs and possibilities

First, let us look at what is needed and to what extent user–programmers can provide it. In programming any extension of Stata, you can write what you like, so long as it is legal syntax. That said, many questions of style do also arise (Cox 2005). In the long term, it pays off, and users will thank you more, if what you write is as close to official Stata style as possible.

In supporting time of day, a natural aim is to mesh as closely as possible with existing support for dates, especially where dates and times of day are likely to touch. Let us recap on how Stata handles dates. If you read the string

"21 January 2006"

you recognize a date immediately. Stata in this case will agree with you, so that

```
. display d(21 January 2006)
```

will elicit a display of 16822, which is the number of days after Stata's origin 0 at 1 January 1960. Conversely, Stata's smartness about dates is finite, and it is fooled by

```
. display d(January 21 2006)
```

although not by

```
. display d(21/1/2006)
```

In general, dates may be shown in Stata as strings that have meaning but are subject to arbitrary misunderstanding, or as numeric values, meaning integers, counting time units from their origin in 1960, which will be understood correctly so long as Stata is told what it is dealing with. Although shown here only for daily dates, the concept holds for dates varying from weeks through months, quarters, and halves to years.

You can bridge the gap between numeric and string values, which can be collected in variables, in various ways. Value labels are possible but would be inefficient for most time-series problems given the many possible distinct values. A neater way is using formats. Thus

```
. display %dM_d_CY 16822
```

reverses the process. The format `%dM_d_CY` can be thought of as an instruction to produce something like a value label on demand, although unlike value labels the text cannot be completely arbitrary. That catch is not much of a problem with date formats, as the little language defining them is rich enough for most purposes.

Times of day are similar to dates in many ways. We should distinguish carefully among

1. Times in the strict sense, such as "17:16" mentioned earlier. The colon convention used here is common but not universal. If you ask Stata to show the current time by

```
. display "$S_TIME"
```

or

```
. di c(current_time)
```

it will give a colon-separated time of the form $hh:mm:ss$.

2. *Date–times*, in which both a daily date and a time are combined, such as "21 January 2006 17:16".

3. Durations or timings. An Internet search reveals that the women's marathon record is, at the time of writing, 2:15:25—2 hours, 15 minutes, 25 seconds—held by the British athlete Paula Radcliffe. This time's association with a daily date, 13 April 2003, does not make it a date–time. Thus we can imagine a dataset in which times to run a maze or to complete some other task are presented without times of observation, so that timings without a date or time context can make perfect sense. Timings are just differences between two clock times, even though those clock times are usually unstated. (They might be interesting, as when time of day affects an experiment, but that is another story.) If we provide functionality for times, we also provide it for timings. That said, in practice scientists producing timings will usually choose a convenient time unit and record times as a multiple thereof rather than reporting them as composite units. But we want to be able to handle composite units easily when they occur.

No special Stata formats are available for times, date–times, and timings. Moreover, user-defined formats are not within the scope of Stata, at least at present. We might hope that in due course official Stata extends formats to include time-of-day formats. Value labels are impractical for the reason already given, the many distinct values that might be needed.

The most practical approach to times of day for user–programmers is to handle them as both numbers and strings. The numeric form will be needed for most calculations, such as calculating time differences, rates, or summary measures; in graphics; and so forth. Few time calculations are amenable to string manipulations, but string representations are needed for display or presentation. Users might want to show colons, spaces, month information, and extra text such as `h`, `m`, or `s`, and `am` or `pm`.

Units and precision now need discussion. For numeric representation, users might want a variety of time units for times of day, date–times, and timings. Those that seem most likely—from a user's point of view—are days, hours, minutes, seconds, or milliseconds.

Let us review some basic time arithmetic. There are 24 hours in a day, 60 minutes in an hour, and 60 seconds in a minute. Thus there are 1,440 minutes in a day, 3,600 seconds in an hour, and 86,400 seconds in a day. Imagine that time is measured in exact hours (minutes, seconds, respectively) and that no date component is used. Then an `int` variable will be adequate for up to 32,740 time units, that is, 32,740 hours (over 545 hours, just over 9 hours, respectively). Thus for coarse resolutions, or short time spans,

an `int` might be adequate. Conversely, if dates are also used, or the time resolution is shorter, then other data types should be used. The choice in practice is between `double` and `long`. `long` will be fine for large integers, but for greater generality in being able to handle any fractional part, even down to, say, milliseconds, `double` is likely the better choice.

If you had a numeric representation of time of day using one unit and wished to convert to another (say, hours and fractions of hours to days and fractions of days), then no special code is needed; `generate` and the appropriate conversion constant will suffice.

If you had variables containing the individual elements of times of day and wanted to use them to produce a composite, then that too is relatively simple. Suppose that you have separate numeric variables giving `hours`, `minutes`, and `seconds`. A numeric composite, with units of seconds, could be yielded by

```
. gen ntime = 3600 * hours + 60 * minutes + seconds
```

and a string composite by

```
. egen stime = concat(hours minutes seconds), p(:)
```

In the last case, there is no harm in doing it with more fundamental features:

```
. gen stime = string(hours) + ":" + string(minutes) + ":" + string(seconds)
```

Easy though such manipulations are, having individual variables like this appears uncommon in practice. The inverse may be more common. Thus given the time 17:16, $60 \times 17 + 16$ yields 1,036 (minutes after midnight). How would that numeric calculation be reversed? In Stata terms, `floor(1036 / 60)` yields 17 and `mod(1036, 60)` yields 16. More generally, other such calculations require a duet of `floor()` (a bass voice) and `mod()` (a soprano). For more on the mathematics of the floor and mod functions, see Knuth (1997) or Graham, Knuth, and Patashnik (1994). For more on the uses of `floor()` in Stata, see Cox (2003).

Here `floor()` never rounds up and is thus simpler than `int()`, which rounds noninteger toward zero, down when positive, and up when negative. Using `int()` rather than `floor()` could lead to bugs when dealing with Stata daily dates less than 0, i.e., before 1 January 1960.

Given a string representation such as "17:16", the elements could instead be extracted by using `substr()` or the `split` command.

To summarize, many simple time-of-day calculations reduce to simple manipulations using one or more of

- the commands `generate`, `replace`, and `split`,
- the functions `floor()`, `mod()`, and `substr()`, and
- the `egen` function `concat()`.

This system still leaves considerable scope for commands that take care of these details for you.

4 Two new commands

Two new commands have been written: `ntimeofday` for converting from string times of day (or timings) to numeric versions of the same and `stimeofday` for conversion in the opposite direction. The two commands are not quite mirror images of each other, as a few details need attention in one case but not both. The naming convention is that the first letter of the command name indicates the result that will be produced, `n` for numeric and `s` for string.

Designing two programs, not one, is partly a judgment call and partly based on standard Stata logic. Commands and functions in Stata that do conversions usually specify one kind of input and thus one kind of output. The opposite conversion is done by something with a different name. For example, general string-numeric functionality comes in pairs: `encode` and `decode`, `destring` and `tostring`, and `real()` and `string()`.

A simple but powerful principle here is that whenever an operation and its inverse both make sense, both should be supported. `tostring` is a case in point. It was written, in the very first instance, because the operation clearly made sense. It became clear only later that several users really did need it for specific problems. In any case, having two programs gives the developer a set of consistency tests, checking that the output of one program can be translated back to the input of the other.

4.1 ntimeofday

`ntimeofday` generates numeric time-of-day variables from string time-of-day variables. It is designed as a general tool in various senses:

1. It can take several variables and correspondingly produce several new variables. They must all follow the same specification, as explained shortly. Generation is all or nothing; there is no scope for a partial mess that needs to be cleaned up by the user.

2. The user may specify as units for the new numeric variables any one of days, hours, minutes, or seconds. The required option `string()` should be used to specify the sequence of time or date elements followed in each of the string variables supplied. Each element specified must occur just once in any value. The minimum requirement is one element. Elements should be separated by spaces, punctuation, or other nonnumeric characters. The little language used for specification allows daily dates (day, month, year) in any order allowed by `date()` and elements specifying separately one or more of years, months, days, hours, minutes, and seconds. However, certain combinations of year, month, and day elements are regarded as invalid, because they give insufficient information to define a daily date part of a date and time. Thus years or months alone is insufficient. So also are years

and months without days, and months and days without years. Days alone are acceptable, as they must be to support timings. Years and days together are acceptable, as a service to those who use years and day of year running from 1 to 365 or 366: the two elements are interpreted as daily dates using `mdy(1,1,`*year*`)` + (*day* `- 1`).

An `extremes` option produces a report on the range of each date or time element, thus permitting a partial check on expectations. In particular, as far as `ntimeofday` is concerned, the number of hours, minutes, and seconds may reasonably be or exceed 24, 60, and 60, respectively, depending on what is supplied.

3. Text strings indicating times a.m. and p.m. are allowed, but the user must say what they are. Empirical survey shows small variations in use of upper or lower case and periods or no periods, so the onus is placed on the user to spell out what applies. This is really a feature, as an error will result if the data are not as it is thought they should be.

4. A variety of separators are allowed if specified through a `parse()` option. The `split` command is used within `ntimeofday` to produce individual elements, and so the argument of `parse()` is passed to `split`. The default separators are spaces and colons.

5. A `type()` option may be used to specify that numeric variables should be generated as a specified variable type. By default, variables are created as `double`. `type()` is a rarely used option. Specifying `type(long)` will result in loss of information whenever the times or dates contain fractional parts in the units specified.

   Conversely, `ntimeofday` has some explicit limitations.

6. `ntimeofday` neither ignores nor allows for indicators of time zones (e.g., CET) or days of the week (e.g., Mon). These should be removed first with, say, `subinstr()`.

7. It does not support run-together dates such as 20060121 or run-together times such as 1234. For an official comment on how to tackle run-together dates, see http://www.stata.com/support/faqs/data/dateseq.html and for a user-written program, use `ssc` to install `todate`.

8. It makes no adjustments for leap seconds or other astronomical subtleties. Every day in every year is presumed to have 24 hours, each being 60 minutes and each minute in turn being 60 seconds.

Let us look at a few examples. Suppose that a typical date–time is `21 Jan 2006 12:34:56`. Then specify `string(day mo yr h mi s)`. No arguments need be supplied to `parse()`, as the separators of space and colon are its default. The full command might then be

```
. ntimeofday datetime, gen(ndatetime) s(day mo yr h mi s) n(day)
```

which in this case produces a daily date with an extra fractional part.

Or suppose that a typical date–time is `21jan06 12:34:56`. Then specify `string(dmy h mi s) cend(2050)`. Again no arguments need be supplied to `parse()`, as the separators of space and colon are its default. Thus we might have

```
. ntimeofday datetime, gen(ndatetime) s(dmy h mi s) n(day) cend(2050)
```

Note the use of a century-end option here, `cend()`, to spell out that years like `06` are within the century ending in 2050.

Or suppose that timings are specified as `12h 34m 56s`. Then specify `string(h mi s)`. The separators here are `parse(h m s)`. `split` automatically trims the remaining spaces. We can also ask for a display of extremes:

```
. ntimeofday timing, gen(ntiming) s(h mi s) n(day) parse(h m s) extremes
```

Syntax

`ntimeofday` *strvarlist* [*if*] [*in*] `,` `generate(`*newvarlist*`)` `numeric(`*units*`)`
`string(`*specification*`)` [`am(`*am_strings*`)` `pm(`*pm_strings*`)` `cend(`*century_end*`)`
`extremes` `parse(`*parse_strings*`)` `type(`*variable_type*`)`]

Options

`generate(`*newvarlist*`)` specifies a list of new names for the new numeric variables. There must be as many new names as there are existing composite string variables. `generate()` is required.

`numeric(`*units*`)` indicates units for the new numeric variables. One of days, hours (or hrs), minutes, or seconds must be specified but may be abbreviated in any way. `numeric()` is required.

`string(`*specification*`)` specifies the sequence of time or date elements followed in each of the string variables supplied. Each element specified must occur just once in any value. The minimum requirement is one element. Elements should be separated by spaces, punctuation, or other nonnumeric characters. See also the `parse()` option below. `string()` is required. See `help ntimeofday` for more details of *specification*.

`am(`*am_strings*`)` and `pm(`*pm_strings*`)` specify any text used in the string variables to indicate whether times are a.m. or p.m. If two or more different text indicators are used, all should be specified. Thus `am(am)` specifies that at least some times a.m. are flagged with the text "am". Times a.m. are naturally not adjusted, but the use of such indicators should be declared. Similarly, `pm(pm p.m.)` specifies that at least some times p.m. are flagged with one of those indicators. Times p.m. are adjusted by adding 12 to hours from 1 to 11. A p.m. time given as, say, 23:45 pm will not therefore be adjusted. Again the use of such indicators should be declared.

`cend(`*century_end*`)` specifies a century-end to apply to two-digit years whenever the element is one of `ymd`, `ydm`, `myd`, `mdy`, `dym`, or `dmy`, and such dates are to be interpreted with `date("`*date*`", "`*element*`",` *century_end*`)`. See the `date()` function in [D] **functions**.

`extremes` requests a display of the minimum and maximum of each date element. `ntimeofday` performs some checks, and invalid or unintelligible dates will yield missing values. However, the user is advised to apply more, drawing upon knowledge of what the data should be. In particular, as far as `ntimeofday` is concerned, the number of hours, minutes, and seconds may reasonably be or exceed 24, 60, and 60, respectively, depending on what is supplied. Inspection of extremes is thus advised.

`parse(`*parse_strings*`)` supplies parse strings, that is, separators, to `split` (see [D] **split**), which divides string variables into components before they are reassembled. By default, `ntimeofday` causes `split` to parse on spaces and colons.

`type(`*variable_type*`)` specifies that numeric variables be generated as a specified variable type. By default, variables are created as `double`s in `ntimeofday`. `type()` is a rarely used option. Specifying `type(long)` will result in loss of information whenever the times or dates contain fractional parts in the units specified.

4.2 stimeofday

`stimeofday` generates string time-of-day variables from numeric time-of-day variables. It also is designed as a general tool in various senses:

1. Point 1 above also applies; that is, the command will operate on several variables within one call.

2. All the numeric variables supplied must share the same units, one of days, hours, minutes, or seconds. The option `string()` specifies the sequence of time or date elements followed in each of the string variables to be generated. One of the following specifications should be given.

   ```
   days hours minutes seconds
   days hours minutes
   days hours
   hours minutes seconds
   hours minutes
   minutes seconds
   ```

   Thus single elements are ruled out, as requiring only trivial work to produce them, although it might be argued that this exclusion makes the command incomplete. The rule can be stated in one: elements must compose a subsequence of the sequence days, hours, minutes, and seconds. Thus a choice such as hours and seconds is ruled out as capricious. This is a judgment call, and only experience will indicate whether there are demands for forms that are presently regarded as unlikely choices.

3. Similarly to point 3 above, `am()` and `pm()` options may be used to specify text indicating whether times are a.m. or p.m.

4. A `timeonly` option specifies that only the time-of-day part of a date with a time of day be used. Thus given `1453466096` with specified unit seconds, which is `21 Jan 2006 12:34`, then only the time-of-day part would be used: i.e., `mod(1453466096, 86400)` would be used to produce a representation of the time of day.

5. A set of options may be used to tune the presentation of individual date or time elements. Thus `dformat()`, `hformat()`, `mformat()`, and `sformat()` specify formats for day, hour, minute, and second elements. Commonly, but not necessarily, the day format will be specified by using a date format. This option is at the user's discretion. The default format is the default numeric format, given that the number of days could be (part of) a timing. The default formats for hours, minutes, and seconds are `%02.0f`, `%02.0f`, and `%05.2f`, respectively, so that leading zeros apply by default with numbers less than 10. Using default formats may lead to considerable loss of information.

   Similarly, `dsymbol()`, `hsymbol()`, `msymbol()`, and `ssymbol()` specify symbols for day, hour, minute, and second elements. Each symbol occurs after the associated element. The defaults are space, colon, colon, and empty string, respectively, except that the default symbol is always an empty string for the last element specified.

6. Similarly to point 6 above, `stimeofday` does not indicate time zones (e.g., CET) or days of the week (e.g., Mon). If desired, these elements should be added otherwise.

7. Similarly to point 7 above, `stimeofday` does not produce run-together dates or times.

8. Point 8 above also applies; that is, the command makes no adjustments for leap seconds or other astronomical subtleties.

Suppose that you want a typical date–time to look like `21 Jan 2006 12:34:56`. Then specify `string(d h m s) dformat(%dd_m_CY) sformat(%02.0f)`. By default, without these format statements, the same date–time would appear as `16822 12:34:56.00`.

```
. stimeofday ntime, gen(stime) n(s) s(d h m s) dformat(%dd_m_CY) sformat(%02.0f)
```

Or suppose that you want just the time of day from such a date and an indication of both a.m. and p.m. Then specify, e.g., `string(h m s) timeonly am("am") pm("pm")`.

```
. stimeofday ntime, gen(stime) n(s) s(h m s) timeonly am("am") pm("pm")
```

Syntax

stimeofday *numvarlist* [*if*] [*in*], generate(*newvarlist*) numeric(*units*)
string(*specification*) [am(*am_strings*) pm(*pm_strings*) timeonly
dformat(*format*) hformat(*format*) mformat(*format*) sformat(*format*)
dsymbol(*symbol*) hsymbol(*symbol*) msymbol(*symbol*) ssymbol(*symbol*)]

Options

generate(*newvarlist*) specifies a list of new names for the new string variables. There must be as many new names as there are existing numeric variables. generate() is required.

numeric(*units*) indicates units for the numeric variables. One of days, hours (or hrs), minutes, or seconds must be specified but may be abbreviated in any way. numeric() is required.

string(*specification*) specifies the sequence of time or date elements followed in each of the string variables to be generated. One of the following specifications should be given.

days hours minutes seconds	(a)
days hours minutes	(b)
days hours	(c)
hours minutes seconds	(d)
hours minutes	(e)
minutes seconds	(f)

hrs is allowed as an alternative to hours.

string() is required.

am(*am_string*) and pm(*pm_string*) specify text used in the string variables to indicate whether times are a.m. or p.m. Thus am(am) specifies that times a.m. are to be flagged with the text "am". Times a.m. are naturally not adjusted. Similarly, pm(" p.m.") specifies that times p.m. are to be flagged with that text, including the leading space. Times p.m. are adjusted by subtracting 12 from hours from 13 to 23. By default, no such text is used. One of am() and pm() may be specified without the other. am() and pm() are deemed incompatible with (c) and (f) above. Assigning am or pm text when the number of hours is 24 or more will cause an error.

timeonly specifies that only the time-of-day part of a date with a time of day be used. Thus given 1453466096 with specified unit seconds, which is 21 Jan 2006 12:34, then only the time-of-day part would be used; i.e., mod(1453466096, 86400) would be used to produce a representation of the time of day.

`dformat(`*format*`)`, `hformat(`*format*`)`, `mformat(`*format*`)`, and `sformat(`*format*`)` specify formats for day, hour, minute, and second elements. Commonly, but not necessarily, the day format will be specified by using a date format; see [U] **12.5.3 Date formats**. This choice is at the user's discretion. The default format is the default numeric format, given that the number of days could be (part of) a timing. The default formats for hours, minutes, and seconds are `%02.0f`, `%02.0f`, and `%05.2f`, respectively, so that leading zeros apply by default with numbers less than 10; see [D] **format**. Using default formats may lead to considerable loss of information, so you should exercise due care.

`dsymbol(`*symbol*`)`, `hsymbol(`*symbol*`)`, `msymbol(`*symbol*`)`, and `ssymbol(`*symbol*`)` specify "symbols" for day, hour, minute, and second elements. Each symbol occurs after the associated element. The defaults are `" "` (space), `:` (colon), `:` (colon), and `""` (empty string), respectively, except that the default symbol is always an empty string for the last element specified (namely, hours given (c) above and minutes given (b) or (e) above).

5 Conclusion

The two programs here are offered as one step forward in providing support for time of day, allowing the user to try to have the best of both worlds in both calculating with and suitably displaying time, date–time, or duration data. Most of the hard work—extending date formats to include times and in allowing time-series commands to work where appropriate on such data—is yet to be done.

6 Acknowledgments

Kit Baum, William Gould, and Vince Wiggins produced programs or comments that were helpful in identifying user needs and possible solutions. Jeff Warburton, Louise Bracken, Alona Armstrong, and Mark Smith provided experiences with rain gauge data and advice on athletics conventions.

7 References

Cox, N. J. 2003. Stata tip 2: Building with floors and ceilings. *Stata Journal* 3: 446–447.

———. 2005. Suggestions on Stata programming style. *Stata Journal* 5: 560–567.

Graham, R. L., D. E. Knuth, and O. Patashnik. 1994. *Concrete Mathematics: A Foundation for Computer Science*. Reading, MA: Addison–Wesley.

Knuth, D. E. 1997. *The Art of Computer Programming. Volume I: Fundamental Algorithms*. Reading, MA: Addison–Wesley.

Reingold, E. M., and N. Dershowitz. 2001. *Calendrical Calculations: The Millennium Edition*. Cambridge: Cambridge University Press. [See also http://www.calendarists.com for errata and extra material.]

Richards, E. G. 1998. *Mapping Time: The Calendar and its History*. Oxford: Oxford University Press. [See also http://www.users.zetnet.co.uk/egrichards/book.htm for errata and extra material.]

Whitrow, G. J. 1975. *The Nature of Time*. Harmondsworth: Penguin.

About the author

Nicholas Cox is a statistically minded geographer at Durham University. He contributes talks, postings, FAQs, and programs to the Stata user community. He has also coauthored 15 commands in official Stata. He was an author of several inserts in the *Stata Technical Bulletin* and is an editor of the *Stata Journal*.