The *Stata Journal* publishes reviewed papers together with shorter notes or comments, regular columns, book reviews, and other material of interest to Stata users. Examples of the types of papers include 1) expository papers that link the use of Stata commands or programs to associated principles, such as those that will serve as tutorials for users first encountering a new field of statistics or a major new technique; 2) papers that go "beyond the Stata manual" in explaining key features or uses of Stata that are of interest to intermediate or advanced users of Stata; 3) papers that discuss new commands or Stata programs of interest either to a wide spectrum of users (e.g., in data management or graphics) or to some large segment of Stata users (e.g., in survey statistics, survival analysis, panel analysis, or limited dependent variable modeling); 4) papers analyzing the statistical properties of new or existing estimators and tests in Stata; 5) papers that could be of interest or usefulness to researchers, especially in fields that are of practical importance but are not often included in texts or other journals, such as the use of Stata in managing datasets, especially large datasets, with advice from hard-won experience; and 6) papers of interest to those who teach, including Stata with topics such as extended examples of techniques and interpretation of results, simulations of statistical concepts, and overviews of subject areas.

The *Stata Journal* is indexed and abstracted by *CompuMath Citation Index*, *Current Contents/Social and Behavioral Sciences*, *RePEc: Research Papers in Economics*, *Science Citation Index Expanded* (also known as *SciSearch*), *Scopus*, and *Social Sciences Citation Index*.

For more information on the *Stata Journal*, including information for authors, see the webpage

http://www.stata-journal.com

Subscription rates listed below include both a printed and an electronic copy unless otherwise mentioned.

U.S. and Canada		Elsewhere	
Printed & electronic		**Printed & electronic**	
1-year subscription	$115	1-year subscription	$145
2-year subscription	$210	2-year subscription	$270
3-year subscription	$285	3-year subscription	$375
1-year student subscription	$ 85	1-year student subscription	$115
1-year institutional subscription	$345	1-year institutional subscription	$375
2-year institutional subscription	$625	2-year institutional subscription	$685
3-year institutional subscription	$875	3-year institutional subscription	$965
Electronic only		**Electronic only**	
1-year subscription	$ 85	1-year subscription	$ 85
2-year subscription	$155	2-year subscription	$155
3-year subscription	$215	3-year subscription	$215
1-year student subscription	$ 55	1-year student subscription	$ 55

Back issues of the *Stata Journal* may be ordered online at

http://www.stata.com/bookstore/sjj.html

Individual articles three or more years old may be accessed online without charge. More recent articles may be ordered online.

http://www.stata-journal.com/archives.html

The *Stata Journal* is published quarterly by the Stata Press, College Station, Texas, USA.

Address changes should be sent to the *Stata Journal*, StataCorp, 4905 Lakeway Drive, College Station, TX 77845, USA, or emailed to sj@stata.com.

The *Stata Journal* (ISSN 1536-867X) is a publication of Stata Press. Stata, **Stata**, Stata Press, Mata, **Mata**, and NetCourse are registered trademarks of StataCorp LP.

Stata tip 125: Spell boundaries

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

1 Introduction

Identifying spells or runs of observations is a common problem in data management and data summary. A detailed tutorial was given in a "Speaking Stata" column (Cox 2007). That column identified some simple techniques for working with spells in Stata:

1. Mark the start of each spell with an indicator variable. The key is that observations at the start of spells will differ from their predecessors. Care may be needed in handling the very first observation, either in a dataset or in a panel.

2. Use cumulative sums to map start indicators to spell identifiers that are 1 up. It is also useful to identify gaps between spells by 0. Given identifiers, summarizing spell characteristics is then usually straightforward. `egen` functions are particularly useful.

3. Panel datasets are no more difficult than individual series, so long as you use `by:`. Using features allowed after `tsset` or `xtset` is perfectly sensible but not essential.

4. Some spell criteria do require two passes through the data. Typically, spells are reclassified on the second pass, say, to restrict spells to certain lengths or to allow short gaps within spells.

If you are nodding in agreement with these points, do read on. If they seem cryptic, please read (or skim) the 2007 column first.

Here I add further detail on two common problems. The first is when a spell is defined by its end condition. With just a twist, this can be recast easily as a condition for defining its start.

The second problem need not be considered as a question of defining spells but can be seen in that framework. This problem pertains to calculating the time or number of observations since some event, which can be approached directly.

2 The ends define the spells

Many spells are defined just as naturally by when they end as by when they start. Sometimes, the exact time of ending may be known, but the starting time may be

unclear or definable only arbitrarily. For example, an election marks the end of a political campaign. A birth marks the end of a pregnancy. A sale may mark the end of a period of contact between a seller and a potential buyer.

Let's assume that we have, or can create, an indicator variable for the end of a spell, say, `end`. Let's also assume that data are sorted in order of a time or other sequencing variable, possibly within panels in the case of panel or longitudinal data. The criterion for the start of a spell can then be something like

```
. generate begin = end[_n-1] | _n == 1
```

for a single series. For panel data, it may be

```
. bysort id (time): generate begin = end[_n-1] | _n == 1
```

Here the code is short for

```
. generate begin = (end[_n-1] == 1) | (_n == 1)
```

and similarly for the panel case. If the indicator variable is only ever 1 or 0, then `end[_n-1] == 1` yields 1 (true) precisely when `end[_n-1]` is 1 (true).

In general, two possibilities define the start of a spell. Either the previous observation was the end of a spell, or this is the first observation. Here `_n` defines observation number; under the aegis of `by:`, observation numbers are defined within groups; and `|` is the logical operator "or".

Given that indicator variable for the beginning of a spell, a spell identifier is just

```
. generate spellid = sum(begin)
```

Let's see how this works with a small example. There are 7 observations, and the indicator `end` is 1 in observation 4.

```
. list, sep(0)
```

	time	end	begin	spellid
1.	1	0	1	1
2.	2	0	0	1
3.	3	0	0	1
4.	4	1	0	1
5.	5	0	1	2
6.	6	0	0	2
7.	7	0	0	2

Some complications of this basic idea are predictable. With this approach, a spell will always be identified with identifier 1, regardless of whether the event took place. For example, the potential buyer may never have proceeded to purchase. A spell was identified as such because it started with the first observation. In that case, it may be sensible to reclassify the spell because it was incomplete. That is easy. For a single

series, we can `summarize end` and replace `spellid` if its maximum is only 0. After `summarize`, the maximum is accessible as `r(max)`.

```
. summarize end, meanonly
. if r(max) == 0 replace spellid = 0
```

Here we reclassified an incomplete spell to have 0 as an identifier. Another possibility that might appeal is to reclassify to a missing value.

For panel data, we need to examine each panel separately. We can calculate the maximum of `end` for each panel with

```
. bysort id: egen max = max(end)
. replace spellid = 0 if max == 0
```

and thus reclassify the incomplete spell in each panel.

A similar problem arises if the last spell—whether for all the data or for a single panel—is incomplete. Again, "incomplete" means that the last observation has a value of 0 for `end`. For a single series, we reclassify with

```
. replace spellid = 0 if end[_N] == 0 & spellid == spellid[_N]
```

and for panel data, we merely apply that under `by:`,

```
. bysort id (time): replace spellid = 0 if end[_N] == 0 & spellid == spellid[_N]
```

However, notice now that this technique will take care of the first problem too. If the data define a single incomplete spell, then it will also be true that the last observation has value 0 for the end indicator, and the condition that the spell identifier equals the last spell identifier will catch all the relevant data. So you can forget the first technique, or feel good about having two ways of solving the problem.

3 Time since an event

Researchers often want to keep track of the time since some event. Events can be anything deemed to happen at a single time or point in a sequence. A common example on Statalist is an initial public offering or a stock market launch.

In a sense, this problem is a twist on the problem of identifying the previous value of some variable that was different (Cox 2011), but the "Tip" just cited did not spell that out, and the problem fits in here quite well.

It is also a twist on the problem of identifying a counter variable indicating sequence within spells, which is a problem also discussed in Cox (2007). But we can approach it without even identifying spells explicitly, and will do so now.

We again assume an indicator variable for an event and order by time or sequence variable, possibly within panels, as in the previous section. We need not be precise about whether the event marks the start or end of a spell. Hence, we phrase matters

using a neutral variable name such as `event`. `event` will be 1 when an event occurred and 0 otherwise. The times at which events occur can be copied into a new variable with

```
. generate when = time if event
```

which is a valid command for either a single series or panel data. Note that the condition `if event` implies that values will be missing for times other than those when an event took place. But we can copy values downward in a cascade. For a single series, we use

```
. replace when = when[_n-1] if missing(when)
```

and for panel data, we use

```
. bysort id (time): replace when = when[_n-1] if missing(when)
```

If you are not familiar with this trick, it works like this: to begin, `generate` and `replace` use the current order of observations (Newson 2004). So a missing value immediately after a nonmissing value of `when` can be `replace`d with that nonmissing value; a missing value immediately after that can be `replace`d with the same nonmissing value, which is now the previous value. We then continue in a cascade until we reach the next nonmissing value or the end of the panel or the end of the data, whichever comes first.

Now the times since the last event are immediately available by subtraction, as follows:

```
. generate time_since = time - when
```

Sometimes, researchers like to restrict attention or calculation to times within some specified interval of an event, say, within the next 30 days or 2 years. The variable just created will then naturally appear in a condition specified with `if`, such as `time_since <= 30`.

An inevitable side effect of this calculation is that the variables `when` and `time_since` will be returned with missing values for observations before the first event. Typically, that should be considered logical and desirable.

An attraction of this device is that few assumptions are being made. There is no assumption that times are evenly spaced. There is no assumption that a time variable or a panel identifier variable has been declared with `tsset` or `xtset`.

A variant on the problem is that researchers sometimes want to count observations after an event rather than measure the time elapsed. The two are not equivalent whenever times are not evenly spaced or the difference between times is not 1 in whatever time units are being used. An example might be counting patient visits to a clinic after some event, say, an initial consultation.

A technique for this preference is to initialize a counter, as follows:

```
. generate counter = 0 if event
```

We then count upward (as in elementary arithmetic) by adding 1 repeatedly. For single series, use

```
. replace counter = counter[_n-1] + 1 if missing(counter)
```

For panel data, do that within panels. If it makes more sense to regard the event itself as a count of 1, the modification is clear.

References

Cox, N. J. 2007. Speaking Stata: Identifying spells. *Stata Journal* 7: 249–265.

———. 2011. Stata tip 101: Previous but different. *Stata Journal* 11: 472–473.

Newson, R. B. 2004. Stata tip 13: generate and replace use the current sort order. *Stata Journal* 4: 484–485.