The Stata Journal (2007)
7, Number 2, pp. 249–265

Speaking Stata: Identifying spells

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

Abstract. Spells in time series (and more generally in any kind of one-dimensional series) may be defined as sequences of observations that are homogeneous in some sense. For example, a categorical variable may remain in the same state, or values of a measured variable may satisfy the same true–false condition. Devices for working with spells in Stata include marking the start of each spell with indicator variables and tagging spells with integer codes. Panel data are easy to handle with the `by:` prefix. Some kinds of spell identification require two passes through the data, as when only spells of some minimum length are of interest or short gaps are tolerable within spells. Many questions concerning spells are easy to answer given careful use of `by:` and appropriate sort order, selection of just 1 observation from each panel or spell, and appreciation of the many functions written for `egen`. Gaps before, between, and after spells can also be important, and I suggest a convention for handling them.

Keywords: dm0029, spells, runs, time series, data management

1 Introduction

Researchers with time-series data often want to identify *spells* in their data, periods that are homogeneous in some sense. Within a spell, some condition holds over a sequence of observations, short or long as the case may be. The spell ends when that condition no longer holds (or, conventionally, at the end of the available data). The term *run* has also long been in use, especially whenever there is comparison of run numbers or lengths with various simple probabilistic models. The focus here is on data management and data summary, contexts in which talk of spells seems much more frequent.

The aim of this column is to show you how to handle spell problems with complete control over your specifications. I will explain how to combine various basic Stata commands to identify spells and then calculate their properties.

2 What is a spell?

Interest in spells can arise with both categorical and measured data. You might have data on employment, so that members of a panel experience spells in and out of work. Or you might have data on women who gave birth on various dates, on elections that returned different parties to power, or on eruption and earthquake times. A birth, an election, an eruption, and an earthquake all can be thought of as initiating new periods or spells. Or you might have daily data on rainfall or share prices and want to determine periods in which it rained at least 10 mm every day or in which share prices were increasing. Given one or more objective criteria that define a spell, you can automate identification of spells.

The idea of spell here is both broad and narrow. There may be gaps between spells, so that it need not rain every day, or share prices might decrease. However, spells are disjoint and may not overlap. That is not really a restriction; you can always identify different sets of spells by different criteria and then examine whether those sets overlap.

Although the idea of spells arises most obviously in the analysis of time-series data, it also applies to other sequences, including one-dimensional spatial series for transects or profiles. "Time" in such contexts means position or order in a spatial sequence. Such series may not have a natural direction, unlike time series, so that what is regarded as "previous" or "following" depends on some convention about the recording of data. However, any indeterminacy over the direction of spatial data is not usually problematic for identification of spells.

The focus here is on deterministic definitions: an observation unequivocally is or is not within a particular spell. There are also statistical definitions in which spells are relatively homogeneous in some sense, but some within-spell variability is allowed. From one point of view, the problem is then one of cluster analysis, but with a special constraint that only contiguous observations may be put in a given cluster (Hartigan 1975). From another point of view, the problem is one of detecting one or more change points, at which the level of some variable may be thought to jump. Many solutions have been offered for both problems, but they lie beyond the scope of this column.

3 Spells in Stata

The central question in this column is how to identify such spells for yourself in Stata. There are three main devices. First, define indicator variables that mark the beginning of each spell. Second, define identifier variables that tag distinct spells. Third, make full use of `by:` together with `_n` and `_N`. An earlier column (Cox 2002) gave an extended tutorial on these last features. Having read that column, or knowing about `by:` already, will make this column easier to understand, but I will assume neither here. Reminding yourself of the syntax of `by:` by reading the manual entry or the online help may be a good idea at some point.

It is no restriction to assume that you will have a variable that indicates time or spatial order. At worst, you may not yet have such a time variable, but you do know that your data are in time order. The data might be the results of a series of psychological tests administered to a person in time order. If so, create a time variable straight away:

```
. gen time = _n
```

As some of the examples so far given do indicate, spell identification is often wanted with panel or longitudinal data. Spells are to be determined separately for different members of a panel. One of the several advantages of `by:` is that panel structure is just as easy to deal with as individual time series, as we will see shortly.

People coming to Stata from some other language might fairly guess that identification of spells is some kind of looping problem. That is, we need to look in turn at each observation and decide whether it resembles its predecessor, and so belongs in the same spell, or it differs, and so is the start of a new spell. This guess is in essence correct, but you do not need to write programs with a loop construct such as `while`, `forvalues`, or `foreach`. Stata's ability to refer to previous observations by using subscripting and its `by:` prefix are typically sufficient, and a few commands used interactively solve most spell problems.

4 Start at the beginning

Begin with spell problems by focusing on the times at which spells themselves begin. Here is a toy dataset, for one person or object. We will soon get to consider full panel structure.

`year`	`state`
1995	A
1996	A
1997	A
1998	B
1999	B
2000	B
2001	C
2002	C
2003	C
2004	C

These data are, as expected, in time order. Identifying spells hinges on data's being in time order for each time series. That should not seem surprising. It is always a good idea to `sort` explicitly by, say,

```
. sort year
```

Such sorting does no harm at the best of times, and it will catch situations in which your data somehow got out of order.

These toy data fall into three distinct spells, in which `state` is in turn A, B, and C. `state` could here be a string variable or a numeric variable with value labels. The general idea of a spell does not depend on what kind of variables you have.

At the start of each new spell, `state` differs from its previous value. Recall that Stata uses the system variable `_n` for observation numbers. In years 1998 and 2001, `state` (which always can be thought of in its fuller form as `state[_n]`) is not equal to its previous value, `state[_n-1]`. This finding leads to the idea of an indicator variable marking the start of each spell:

```
. gen byte begin = state != state[_n-1]
```

The simple device of marking the start of each spell is the basis for solving all spell problems. We are using `generate` to create a new variable. The expression specifying an inequality, `state != state[_n-1]`, will be true when `state` differs from its previous value and false otherwise, when `state` is the same as its previous value. Stata returns 1 when a logical expression is true and 0 when it is false.

It is good practice to specify that the new indicator variable should be of `byte` type. `byte` variables, as Stata's most compact numeric variable type, suffice to hold 0s and 1s. Using `byte` variables will also help if you are short of memory. The results will be

year	state	begin
1995	A	1
1996	A	0
1997	A	0
1998	B	1
1999	B	0
2000	B	0
2001	C	1
2002	C	0
2003	C	0
2004	C	0

What happened for the first observation needs special attention. `state != state[_n-1]`—or equivalently `state[_n] != state[_n-1]`—for the first observation is `state[1] != state[0]`, as `_n` is 1 for the first observation. `state[0]` is before the start of the data. Stata does not know what it should be, any more than whoever compiled the data, perhaps yourself. Stata's way of saying "I don't know" is to return missing. In particular, whenever Stata is asked to determine a value in an observation that it does not have, it returns missing. Missing will mean numeric missing (`.`) with numeric variables and string missing (`""`) with string variables. Either way, `state[0]`, returned as missing, will not be the same as `state[1]`, here with the value or value label A. This is why `begin` is returned as 1 for 1995, indicating 1995 as the first year in a spell.

Almost always, this rule of Stata will give the behavior that we want for identifying spells. Occasionally, we will need to be a little more careful. Suppose that interest is in identifying spells of missing data. The first values in spells of missing values are then given by

```
. gen byte begin = missing(state) & (state != state[_n-1])
```

except that this is not right whenever the first value observed is missing. Here `state[1]`, which is missing, will be equal to `state[0]`, which as explained is deemed to be missing too. This small problem is easy to fix. The compound condition that defines the start of a spell needs tweaking to

```
. gen byte begin = missing(state) & ((state != state[_n-1]) | (_n == 1))
```

So, an alternative to values of `state` differing from the values previously recorded is that the observation be the first. Either of two conditions is sufficient for the start of a spell of missing values:

(a) This value of `state` is missing, and the previous value is not.

(b) This value of `state` is missing, and this is the first value. Recall that `_n == 1` identifies the first observation.

In examples like this, parenthesizing the code aggressively to spell out the logic, not only to Stata but also to yourself and anybody else reading it, should cause no embarrassment. You need not assume knowledge of Stata's precedence rules that determine interpretation when several operators are used in one expression. More importantly, you may avoid some horrible little bugs.

We now have an indicator or marker variable for the start of each spell, and that itself may be useful. If we want to `summarize` some variables, taking snapshots only at the start of each spell, then the qualifier `if begin == 1`—or even more concisely `if begin`—identifies those observations only.

Here are some useful details for stipulating logical conditions. Whenever an `if` condition includes a numeric variable name only, Stata looks inside each value of that variable and treats nonzero values as true and zero values as false. Here `if begin` is true whenever `begin` is 1 and false whenever `begin` is 0, and so `if begin` is equivalent in practice to `if begin == 1`. Also, `if !begin` is equivalent to `if begin == 0`, as negation using `!` flips nonzero to 0 and 0 to 1.

5 From indicators to identifiers

An indicator variable for spell starts is one step away from something we usually want, an identifier variable that tags distinct spells. This variable can most easily be just the cumulative sum, given by the `sum()` function:

```
. gen spell = sum(begin)
```

(*Continued on next page*)

Here are the results in our toy example:

`year`	`state`	`begin`	`spell`
1995	A	1	1
1996	A	0	1
1997	A	0	1
1998	B	1	2
1999	B	0	2
2000	B	0	2
2001	C	1	3
2002	C	0	3
2003	C	0	3
2004	C	0	3

Evidently, as we are summing 1s and 0s, the identifier changes only when we start a new spell, exactly as we would desire.

Now let us consider a common complication. Suppose that spells are defined by being in state B. We consider states A and C, whatever they are, to define gaps between spells. Our start criterion is now a compound condition. If `state` is a string variable, we can say

```
. gen byte begin = (state == "B") & (state != state[_n-1])
```

and if it is a numeric variable with value label attached, we would usually specify the appropriate numeric value, such as

```
. gen byte begin = (state == 2) & (state != state[_n-1])
```

Let us work through that possible problem with the first observation. Whenever the first value is indeed `"B"` or 2, it will differ from the zeroth value, the one before, which Stata will deem to be missing. Hence we have no need for an extra condition like `_n == 1` in defining the start of the spell.

What about the identifier variable?

```
. gen spell = sum(begin)
```

will not do by itself, as `sum(begin)` will not only be 1 as soon as the first B is met but also remain that way at the end of the spell when `state` becomes C, or indeed anything other than B. Only when a new B is observed will the identifier change.

This in turn is easy to fix. We have been using a simple and natural convention: the first, second, third, and any following spells have been assigned identifiers 1, 2, 3, and so on. A convenient extension is to label gaps between spells by 0s. Normally, we will not care much about those gaps, except that they exist, so lumping them together will lose us little. If we do later decide that they are interesting, or at least useful, we can define them as spells of a different kind. This convention can be implemented in two steps. Consider the string variable case:

```
. gen spell = sum(begin)
. replace spell = 0 if state != "B"
```

Even better: it can be implemented in one step, by exploiting the helpful function `cond()`:

```
. gen spell = cond(state == "B", sum(begin), 0)
```

Even if `cond()` is new to you, its operation should seem clear from this example. If `state` is B, we use `sum(begin)` for our identifier; otherwise, we use 0. For a tutorial on `cond()`, see Kantor and Cox (2005).

The results will look like this:

year	state	begin	spell
1995	A	0	0
1996	A	0	0
1997	A	0	0
1998	B	1	1
1999	B	0	1
2000	B	0	1
2001	C	0	0
2002	C	0	0
2003	C	0	0
2004	C	0	0

The words "gap" and "between" should be treated elastically. A gap, meaning a sequence of observations not in a spell, can exist before the first spell (as in 1995–1997) or after the last spell (as in 2001–2004).

A convention of identifying gaps by 0 allows another useful shortcut. The condition that observations are within *some* spell (but not specifying *which* spell) is just `if spell > 0` or even more concisely `if spell`.

So far, we have been looking at examples in which some condition, either the value of `state` or whether `state` is B, remaining constant was the definition of a spell (or implicitly of gaps between spells). The opening examples included spells begun by some kind of event (births, elections, earthquakes, eruptions). Usually, we suppose that each such spell lasts until the next such event. We do not need any more tricks for this, as we already know that given an indicator variable for the start of a spell we can move directly to an identifier variable.

When the criterion for a spell is quantitative, it should be specified, whenever possible, in terms of an equivalent true–false criterion. Suppose that we have daily rainfall data and want spells in which rainfall was at least 10 mm every day. We could go

```
. gen byte begin = (inrange(rain,10,.) & !inrange(rain[_n-1],10,.)
. gen spell = cond(inrange(rain,10,.), sum(begin), 0)
```

`inrange()` here, as elsewhere, conveniently excludes missing values (Cox 2006). The apparently simpler solution

```
. gen byte begin = (rain > 10) & !(rain[_n-1] > 10)
```

is the wrong criterion for the first observation as `rain[0]` is deemed missing (and thus ≥10 mm). As earlier, this could be fixed by allowing `_n == 1` as an extra criterion.

You might prefer to be a little more long-winded for clarity:

```
. gen byte wet = inrange(rain, 10, .)
. gen byte begin = wet & !wet[_n-1]
. gen spell = cond(wet, sum(begin), 0)
```

Binary or dichotomous division of quantitative ranges is naturally not the only possibility. You can categorize ranges by using whatever intervals or bins you want. You should usually worry at least a little about how far any categorization is arbitrary. One that can seem natural for many time series is categorizing into periods of increase, stability, and decrease.

6 Panel structure

The extension from one series to several defined for each of several panels is also easy, so long as we let `by:` flex its muscles. The appropriate `sort` order to work with is first by identifier and then by time or position, say,

```
. sort id time
```

If you are using `tsset`, then

```
. tsset
```

will ensure that same sort order. Using `tsset` is not essential for spell identification, but it is a good idea generally. All that we need to add to our technique is using `by:` as a prefix when generating indicators and identifiers. Our first two examples would be generalized to

```
. by id: gen byte begin = state != state[_n-1]
. by id: gen spell = sum(begin)
```

and

```
. by id: gen byte begin = (state == "B") & (state != state[_n-1])
. by id: gen spell = cond(state == "B", sum(begin), 0)
```

Evidently, calculations must be done separately for each panel, but this is not a problem: it is exactly what we want. When `by:` is specified, `_n` as the observation number is determined separately for each panel, so that (for example) `_n == 1` always identifies the first observation in each panel. The spell identifiers produced by this recipe also start at 1 for each panel. This is not a restriction either. If we ever wanted to lump together spells from different panels, they can be given unambiguous identifiers by

```
. egen SPELLID = group(id spell) if spell, label
```

which, as a side effect, assigns missing values to any gaps for which `spell` is 0. We could in turn replace those missings with zeros:

```
. replace SPELLID = 0 if spell == 0
```

7 Useful results are now at hand

7.1 Spell properties

Once you have indicators for the start of spells and identifiers for being within spells (or within gaps), almost everything else to do with spells is easy. We might want summary statistics for each spell, say, the mean of some response. The functions written for `egen` can be invaluable for this purpose:

```
. by id spell, sort: egen mean = mean(response)
```

There are just two details to be careful about. First, as above, a spell is in general specified jointly by a panel identifier and a spell identifier. If there is just one panel, we need worry only about a spell identifier. Second, when we calculate summaries and put them in new variables, each summary will be repeated for every observation in a panel. Usually we should want to use each summary in some further analysis just once, once for each spell. With an indicator variable already at hand marking the start of each spell, this selection of 1 observation from each spell is yielded by `if begin`.

`egen` has many other uses for spells. The time at which the first spell for each panel started is

```
. by id, sort: egen firstspellstart = min(cond(begin, time, .))
```

What is going on here? We are exploiting the fact that the `egen` function `min()`—like several other `egen` functions—can feed on an expression, which can be more complicated than one variable name. The expression `cond(begin, time, .)` will return the `times` at which spells began for observations that are the first in each spell and missings otherwise. The `egen` function `min()` will find the minimum over those and thus ignore the missings—unless a panel experienced no spells, where missing will be returned, which is fair enough.

Similarly, the time at which the last spell ended for each panel is

```
. by id, sort: egen lastspellend = max(cond(spell, time, .))
```

This command may seem more surprising if you are thinking that missings in Stata are always treated as arbitrarily large. But the `egen` function `max()`, like `min()`, ignores missings unless they are everywhere in sight. (The ordinary functions `min()` and `max()` do the same.)

Perhaps more obvious alternatives are

```
. by id, sort: egen firstspellstart = min(time) if begin
```

and

```
. by id, sort: egen lastspellend = max(time) if spell
```

However, the resulting variables will be sprinkled with missings for those observations that do not satisfy the `if` conditions, which is more awkward for later work. It can in turn be fixed:

```
. by id (firstspellstart), sort: replace firstspellstart = firstspellstart[1]
. by id (lastspellend), sort: replace lastspellend = lastspellend[1]
```

Sorting within panels pushes nonmissing values to the start of each panel so that they can then be used to overwrite every value for each panel. But we will still need to fix the sort order before doing anything else. Evidently, being able to get to the desired point in one step is more attractive.

7.2 Spell lengths

The length of spells is a property of much interest. Depending on what is most sensible for your problem, this could be

```
. by id spell, sort: gen length = _N
```

or

```
. by id spell (year), sort: gen length = year[_N] - year[1] + 1
```

or

```
. by id spell (time), sort: gen length = time[_N] - time[1]
```

Let us back up and look at those one-liners more slowly. A principle used in all is that under `by:`, `_N` is the number of observations in each distinct group of observations, here each spell. (So, as with `_n`, `_N` is interpreted with respect to its own group, not the dataset as a whole.) Thus, in the first one-liner, we just count observations separately. This could be a sensible measure of length in many situations, especially whenever observations are equally spaced (e.g., daily, monthly, yearly).

In the second and third one-liners, we stipulate first that within panels and then within spells, values are sorted by our time variable. (Sorting by panels and spells alone does not ensure this; assuming that it does is a source of nasty small bugs.) Given such sorting, `year[_N]` is the last year within each spell and `year[1]` is the first year within each spell. Adding 1 in the second one-liner matches the fact that (for example) a spell starting in 1995 and ending in 2004 is 10 years long, not 9. The third one-liner is for situations in which `time` is measured so finely that we do not customarily add one unit to get a correct answer.

7.3 Spell ends and sequence numbers

You may need recipes for indicating or marking the last observation at the end of each spell,

```
. by id spell (year), sort: gen byte end = _n == _N
```

and for assigning sequence identifiers within a spell,

```
. by id spell (year), sort: gen seq = _n
```

Again, let us look at those more slowly. The versatility of `by:` really is impressive, even when you appreciate it, but there is correspondingly need for minute care in working out the choreography. Consider again a toy example, but now imagine two panels instead of one:

year	id	state	begin	spell
1995	1	A	1	1
1996	1	A	0	1
1997	1	A	0	1
1998	1	B	1	2
1999	1	B	0	2
2000	1	B	0	2
2001	1	C	1	3
2002	1	C	0	3
2003	1	C	0	3
2004	1	C	0	3
1997	2	C	1	1
1998	2	C	0	1
1999	2	A	1	2
2000	2	A	0	2
2001	2	A	0	2
2002	2	B	1	3
2003	2	B	0	3
2004	2	C	1	4

Incidentally, the fabricated example here is designed to underscore the generality of what we are doing. For example, there are no hidden assumptions about the lengths of panels or the number of spells being the same in each panel.

That detail aside, the last observation in each spell must be identified in steps, first within panel and then within spell. Again we specify that within spells, observations are sorted on the time variable:

```
. by id spell (year), sort:
```

That stipulated, the last, or end, observation in each spell is marked by the observation number in that group of observations being the same as the total number of observations in that group. For example, if there are 7 observations in a spell, then the last is the seventh.

```
. by id spell (year), sort: gen byte end = _n == _N
```

The logical expression `_n == _N` will be true (evaluated as 1) whenever the observation number `_n` is the same as the number of observations `_N` and false otherwise (0). Similarly, observations within each spell can be labeled in sequence by

```
. by id spell (year), sort: gen seq = _n
```

This command would label gaps sequentially, too; remember our convention that they have identifier, here `spell`, of 0. You can leave that as it is, or insist that gap sequence identifiers be zero, or insist that they be missing:

```
. by id spell (year), sort: gen seq = cond(spell, _n, 0)
. by id spell (year), sort: gen seq = cond(spell, _n, .)
```

The beauty of this approach is that you are in charge and can get exactly what you want.

7.4 Number of spells

Another common need is the number of spells in each panel. This is

```
. by id, sort: egen spellno = max(spell)
```

or (just to show that there is more than one way to do it)

```
. by id, sort: egen spellno = total(begin)
```

as the number of spells is the same as the number of spell starts. With this command, the number of spells in each panel, held in the variable `spellno`, is necessarily identical for every observation in each panel. Typically, you would want to use `spellno` just once for each panel. For that, you need to tag just 1 observation in each panel. There are two systematic ways to do that, to use the first observation or the last. To see this, consider that panels could be as short as 1 observation. Then the rule of choosing the first and the rule of choosing the last would still both work, but any other rule, such as choosing the second, would fail. (Being arbitrary is, by comparison, of little consequence.) Let us choose the first in each panel,

```
. by id (time), sort: gen byte tag = _n == 1
```

and then do further work on the number of spells in each panel like this:

```
. tab spellno if tag
```

You may know that the `egen` function `tag()` uses exactly this idea.

8 Spells of consecutive observations

A different spell problem arises whenever there are gaps in panel data. You may then want to look systematically at spells of consecutive observations, especially the length of the longest spell in each panel. Some researchers restrict analyses to the longest spell available for each panel, `drop`ping others from the dataset. This problem yields to the same methods as before.

Suppose that we have observations for one panel at `time`

1, 2, 3, 5, 6, 7, 8, 9, 11, 12

so that we have three spells of consecutive observations

1, 2, 3;   5, 6, 7, 8, 9;   11, 12

as there are gaps before the observations at times 5 and 11. (The term *gap* is here used differently from before, which should not be too confusing.)

At the start of each spell, `time - time[_n-1]` is more than 1. The first observation is no exception, as `time[1] - time[0]` is evaluated as missing, which is more than 1.

Thus the start indicator becomes

```
. by id (time), sort: gen byte begin = (time - time[_n-1]) > 1
```

and the identifier is

```
. by id: gen spell = sum(begin)
```

and the lengths are

```
. by id spell, sort: gen length = _N
```

How do we select the longest spell? Within each panel, `sort` the longest spell to the end. Arbitrarily, let us choose the latest if there are ties for longest spell. Other choices are clearly possible, including random selection.

```
. by id (length time), sort: keep if spell == spell[_N]
```

9 Two passes may be needed

Sometimes, two passes through the data may be the best way—indeed, the only way—to identify spells. We will look at two problems of this kind.

9.1 Spells must be at least so long

Often researchers are interested only in spells of at least a certain length. Scientifically and practically, it is often extended periods in which a condition persists that are important. Misery arises from long-lasting troubles, as life and history make all too clear.

However, you do not know how long a spell lasts until you have reached its end, which makes two passes necessary.

Assume again that we have a panel identifier, `id`; a time variable, `time`; a start indicator, `begin`; and a spell identifier, `spell`. Recall that we can get spell lengths (here we fix on counting observations) with

```
. by id spell, sort: gen length = _N
```

We might want to insist that spells be, say, of at least length 7. Then being within a spell is redefined as a logical or indicator variable:

```
. gen byte inspell = (spell > 0) & (length >= 7)
```

Spells must satisfy whatever criteria they satisfied before and be at least so long. A start indicator variable for these longer spells is then

```
. by id (time), sort: gen byte longbegin = inspell & !inspell[_n-1]
```

and an identifier variable for the same spells is then

```
. by id: gen longspell = cond(inspell, sum(longbegin), 0)
```

9.2 Short gaps are allowed within spells

The opposite of a strict definition (spells must be at least so long) is a generous definition: gaps of up to a certain length are allowed and regarded as parts of each spell. One technique is to follow spell identification on a first pass with a treatment of the gaps between spells as another kind of spell on a second pass.

Assume again that we have a panel identifier, `id`; a time variable, `time`; a start indicator, `begin`; and a spell identifier, `spell`. Recall our convention that within gaps the spell identifier should be 0.

A start indicator variable for gaps is then

```
. by id (time), sort: gen byte gapbegin = !spell & spell[_n-1]
```

We are leaning once more on Stata's rules for true and false. If `spell` is 0, that is logically false, but its negation `!spell` is 1 and logically true. Conversely, `spell[_n-1]` will be treated as logically true whenever it is nonzero. A check will show that this will work as desired for the first observation in each panel, as `spell[0]` will always be deemed missing, which is nonzero. Thus our criterion for the start of a gap is shorthand for

```
(spell == 0) & (spell[_n-1] > 0)
```

An identifier variable for gaps is then

```
. by id: gen gap = cond(!spell, sum(gapbegin), 0)
```

and the lengths of gaps are

```
. by id gap, sort: gen gaplength = cond(gap, _N, 0)
```

Suppose that we are happy to tolerate gaps that are at most length 3. Then being within a spell is redefined with an indicator variable

```
. gen byte inspell = (spell > 0) | (gaplength <= 3)
```

A start indicator variable for these more liberal spells is then

```
. by id (time), sort: gen byte libbegin = inspell & !inspell[_n-1]
```

and an identifier variable for the same spells is then

```
. by id: gen libspell = cond(inspell, sum(libbegin), 0)
```

This liberal spell definition includes, possibly, one gap before the first spell in each panel. Suppose that to you is one step too far: you will tolerate gaps between spells but not a gap before the first spell. (As before, the word *gap* is strained here, but that is a mere point of terminology.) Then you need to specify a further restriction: no gap can be reclassified that occurred before the first spell start. We have already seen how to calculate the time of the first spell start. The revised criterion is then

```
. gen byte inspell = (spell > 0) | ((gaplength <= 3) & (time > firstspellstart))
```

and the code is otherwise identical. A similar issue may arise with any gap after the last spell, and the solution is similar: insist that no gap observation can be reclassified if it occurs after the last spell end.

10 Some questions answered

10.1 Irregular spacing

What if your time series are irregularly spaced, so that some gaps occur between observations?

Sometimes the answer is to define the start of spells by using the lag operator `L.`, not the previous value indicated by the subscript `[_n-1]`. Doing so requires that the data be `tsset`. If the previous time is not present in the data for the same panel, then `L.`*varname* will be deemed missing, just like *varname*`[0]`, and this will typically lead to an indication that a new spell has started.

Often—perhaps more often—a pragmatic answer is that we should work from the data that we have. Thus if a panel was in state B at times 1, 2, and 4, but time 3 is not given in the data, we will not usually presume that the panel jumped to a different state at time 3 and then back again.

10.2 if and in conditions

What about extra `if` and/or `in` conditions?

That is up to you. You are in charge and should specify what your definition is. But be careful. Selecting entire panels, or ignoring some of them, will usually be a simple matter. Selecting some times within panels will often be problematic, as the previous observations, in general, may or may not be selected, too. Normally, extra conditions should not override the principle that spells consist of contiguous sequences of values.

10.3 Missing values

What about missing values?

Same answer, really. Again, be careful.

10.4 Censoring

How do I flag that spells are censored? I want to flag that spells may begin or end only in an artificial sense, as I have data only within a particular time window.

A left-censored spell starts at the first relevant observation (so it might have started earlier, except that we have no data to determine that). A right-censored spell ends at the last relevant observation (so it might have ended later, except that we have no data to determine that). Indicator variables for censoring are easily produced. The first problem is when the first observation in each panel has been declared the start of a panel:

```
. by id (time), sort: gen byte censoredleft = spell & (_n == 1)
```

The second problem is when the last observation in a panel is within a spell:

```
. by id (time), sort: gen byte censoredright = spell & (_n == _N)
```

These variables flag only the beginning or end of each censored spell. The entire first spell is flagged as censored left with

```
. by id (time), sort: gen byte censoredleft = spell == 1 & spell[1]
```

and similarly the entire last spell is flagged as censored right with

```
. by id (time), sort: gen byte censoredright = (spell == spell[_N]) & spell[_N]
```

11 Conclusion

We have looked at some simple techniques for working with spells in Stata.

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 first observation, either in a dataset or in a panel.

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

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

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.

12 Acknowledgments

Kit Baum, Jan Dehn, Richard Goldstein, Stephen Jenkins, Philippe Van Kerm, Gary Longton, Vince Wiggins, and Fred Wolfe have made helpful contributions to my understanding of spells.

13 References

Cox, N. J. 2002. Speaking Stata: How to move step by: step. *Stata Journal* 2: 86–102.

———. 2006. Stata tip 39: In a list or out? In a range or out? *Stata Journal* 6: 593–595.

Hartigan, J. A. 1975. *Clustering Algorithms*. New York: Wiley.

Kantor, D., and N. J. Cox. 2005. Depending on conditions: A tutorial on the `cond()` function. *Stata Journal* 5: 413–420.

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 wrote several inserts in the *Stata Technical Bulletin* and is an editor of the *Stata Journal*.