

Demystifying Intervals

Derek Morgan, PAREXEL International, Billerica, MA

ABSTRACT

Intervals have been a feature of base SAS® for a long time, allowing SAS users to work with commonly (and not-so-commonly) defined periods of time such as years, months, and quarters. With the release of SAS 9, there are more options and capabilities for intervals and their functions. This paper will first discuss the basics of intervals in detail, and then we will discuss several of the enhancements to the interval feature, such as the ability to select how the INTCK() function defines interval boundaries and the ability to create your own custom intervals beyond multipliers and shift operators.

BASICS OF SAS INTERVALS

Intervals are a way that we track periods of time that may or may not vary in duration, such as quarters, months, or years. The SAS System has had the capacity to work with dates and times in these terms, and it is useful when you cannot just substitute a given number of days. As an example, most of the time, 30 days isn't a calendar month, so adding 30 days to a date will give you a date 30 days in the future, but not necessarily a date (or the desired date) in the next calendar month. When this distinction is important to your results, SAS intervals are there to help, saving you the cumbersome task of programming the distinction yourself. Manipulating data with intervals differs from the simple math you can employ with date data; instead of mathematical operations, you manipulate date and time data with a series of built-in SAS functions that operate on dates, times, and datetimes using the intervals that you specify. The SAS System has several intervals already defined to correspond to these standard definitions, and a host of functions designed to be used with them. A complete list of the built-in SAS intervals, the shift points for each, and the default starting point for each interval, is in Appendix 1 at the end of the paper.

Two of the most powerful capabilities of SAS intervals are the ability to alter their starting points easily, and to create custom intervals based on multiples of these standard intervals. For example, if you want your year to start on July 1 instead of January 1 (as in a fiscal year), you can do that without extensive programming. If you need an interval of 10 years (corresponding to a decade), again, there is little programming required. You create an interval based on multiples of existing SAS intervals by using an interval **multiplier**. You move the starting point of an interval by using a **shift index**. You can use one or both to customize SAS intervals to suit your needs. "YEAR100" is an example of an interval with a multiplier: this example created a 100-year interval, or a century. "YEAR.7" is an example of an interval with a shift index. This moves the start of the YEAR interval to the seventh month, July 1 instead of January 1. For the purposes of this paper, when the term "*interval*" is used in a function definition, it means a SAS interval name, plus an optional multiplier and/or shift index.

The mainstays of the SAS interval facility have been, and continue to be, the two interval functions: INTCK() and INTNX(). The INTCK() function counts the number of times that an interval boundary is crossed between two given dates. We will discuss the concept of "interval boundary" in more detail in a moment. Essentially, you would use this function to determine how many calendar months, or some other interval has elapsed between two dates. The INTNX() function will take a date that you give it and move it the number of intervals you specify. This is useful to project future (or past) dates from a given reference point. Version 9 has seen some enhancements to these functions. In particular, the INTCK() function has a brand new parameter which can alter the way that it works, which will affect the results it gives.

WHEN IS A YEAR NOT A YEAR?

This brings us to the first enhancement in intervals. The syntax for the INTCK() function has changed in version 9. It is now: **INTCK(interval,start-of-period,end-of-period,method)**; *interval* must be in quotes. This function still calculates the number of **intervals** between the dates represented by **start-of-period** and **end-of-period**. However, the **method** argument has been added. SAS can now count interval boundaries in two ways: CONTINUOUS (C) or DISCRETE (D). The **method** argument must be enclosed in quotes. If you do not specify **method**, DISCRETE (or D) is the default, and is the way that the INTCK()

function has calculated intervals since its inclusion in SAS. One simple way of thinking about the CONTINUOUS method is that it counts intervals the way we intuitively think of periods of time such as months or years. However, a great deal of existing code relies on the DISCRETE method, and I strongly advise against running through all of it and changing it to the more intuitive method.

To illustrate the potential for problems with SAS intervals, consider the following program and result in bold:

```
DATA _NULL_;
  v1 = INTCK('YEAR', '01jan2016'd, '01jan2017'd);
  v2 = INTCK('YEAR', '31dec2016'd, '01jan2017'd);
  PUT v1= +3 v2=;
RUN;
v1=1      v2=1
```

Wait a minute. We know that a year from December 31, 2016, is not January 1, 2017. What happened? SAS intervals are not a shortcut to doing the math. When the DISCRETE method is used (either explicitly or by default,) the INTCK() function counts the number of times that the period *interval* begins between *start-of-period* and *end-of-period*. It does not count the number of complete intervals between *start-of-period* and *end-of-period*. Therefore, for any date in 2016, the first year period starts on January 1, 2016. Therefore, given any starting date in 2016, the number of YEAR intervals that INTCK() counts will be one for any given ending date in 2017. There is an enormous potential for bad results if you misunderstand how INTCK() calculates using the DISCRETE method. The important thing to remember about the default with intervals is that they are based on the starting point of the interval for the *start-of-period* date given to the function, and not the *start-of-period* date itself.

Let's try that same program with the CONTINUOUS method using the alias "C" in the program below:

```
DATA _NULL_;
  v1 = INTCK('YEAR', '01jan2016'd, '01jan2017'd, 'C');
  v2 = INTCK('YEAR', '31dec2016'd, '01jan2017'd, 'C');
  PUT v1= +3 v2=;
RUN;
v1=1      v2=0
```

This looks much more as you would expect. One year elapsed from January 1, 2016 to January 1, 2017, but not between December 31, 2016 and January 1, 2017. The CONTINUOUS method calculates continuous time from the *start-of-period* date itself. If you want to know how many full calendar months it has been since your *start-of-period* and *end-of-period* dates, then you would use the CONTINUOUS method. You could get different answers for the same number of days between your *start-of-period* and *end-of-period* dates because the calendar isn't evenly spaced. 28 days will be equal to one month when *start-of-period* is in the month of February, and it isn't a leap year. However, 28 days will not be considered one month when it is a leap year, or *start-of-period* is in any month other than February. The INTCK() function will not count a month between January 1 and January 31, which is 30 days, but it will count the 30 days between April 1 and May 1 as one month.

The CONTINUOUS method is useful for calculating anniversaries and milestones tied to dates, times and datetimes. While the CONTINUOUS method may seem more intuitive than the traditional, and still default, way in which the INTCK() function handles intervals, you will have to be cautious in choosing which of the two methods you select. Again, it is not recommended that you start going through your legacy code that uses the INTCK() function and changing the method just because the CONTINUOUS method makes more sense in the way that the INTCK() function counts. That legacy code was written to take advantage of the DISCRETE method, and as seen above, the two are not equivalent.

With that in mind, Example 1 provides a good look at the INTCK() function using its default setting. The "D" argument shown for *method* is optional in the following examples.

| Function Call | Result |
|---|--------|
| INTCK('DAY', '31dec2016'd, '06jan2017'd, 'D') | 6 |
| INTCK('WEEK', '31dec2016'd, '06jan2017'd, 'D') | 1 |
| INTCK('MONTH', '31dec2016'd, '06jan2017'd, 'D') | 1 |
| INTCK('YEAR', '31dec2016'd, '06jan2017'd, 'D') | 1 |

Example 1: Sample INTCK() Function Calls Using DISCRETE Method

Although 7 days have elapsed, only six DAY interval boundaries have been crossed since December 31, 2016 (01/01/2017, 01/02, 01/03, 01/04, 01/05, and 01/06.) The start of the WEEK interval for January 6, 2017 is January 4 (weeks start on Sunday by default, although you can change this with a shift index.) The MONTH, and YEAR intervals for January 6, 2017 start on January 1, 2017, so one boundary has been crossed in each of these cases. The INTCK() function can also count backwards: when **end-of-period** is a date prior to **start-of-period**, the INTCK() function will return a negative number. One thing that the INTCK() function will not do is return a non-integer value, because there is no such thing as a partial interval boundary. Intervals cannot be subdivided, although you can create the equivalent by using a multiplier of a shorter base interval.

To summarize, you use the INTCK() function to count the number of interval boundaries between two dates, times, or datetimes. As of SAS 9, there are two different methods of counting them, CONTINUOUS and DISCRETE. DISCRETE is the default, and it is the way that SAS has counted intervals since the inception of the function. The CONTINUOUS method is good for counting anniversaries and milestones, and can be thought of as the way most people think of elapsed time. The methods are NOT interchangeable, and it is a very bad idea to change existing code to use the new method without careful consideration.

PROJECTING DATES BASED ON INTERVALS

The INTNX() function advances a given date, time or datetime by a specified number of intervals. The syntax for this function is **INTNX(interval,start-date,number-of-increments,alignment)**; where **interval** is one of the SAS intervals from Appendix 1 (again in quotes), **start-date** is the starting date, and **number-of-increments** is how many intervals you want to add or subtract. **number-of-increments** should be an integer, but if it is not, the function will only use the integer portion of a decimal argument, because there is no such thing as a partial interval. **alignment** will adjust the result of INTNX() relative to the interval given. It can be 'B', 'M', or 'E' (quotes necessary) for the beginning, middle, or end of the interval, respectively. There is also the 'S' alignment value, which will adjust the result to the same day as given in the **start-date** argument. To illustrate how INTNX() and alignment works, Example 2 provides a sample program that adds six months to March 20, 2017, with varying alignment values. The result is in bold.

```

DATA _NULL_;
a = INTNX ('MONTH', '20MAR2017'd, 6);
b = INTNX ('MONTH', '20MAR2017'd, 6, 'B');
c = INTNX ('MONTH', '20MAR2017'd, 6, 'M');
d = INTNX ('MONTH', '20MAR/2017'd, 6, 'E');
e = INTNX ('MONTH', '20MAR/2017'd, 6, 'S');
PUT "A) 6 months from 3/20/2017 with default alignment = " a mmdyy10.;
PUT "B) 6 months from 3/20/2017 aligned with beginning of MONTH interval= " b
mmdyy10.;
PUT "C) 6 months from 3/20/2017 aligned with middle of MONTH interval= " c
mmdyy10.;
PUT "D) 6 months from 3/20/2017 aligned with end of MONTH interval= " d mmdyy10.;
PUT "E) 6 months from 3/20/2017 aligned with same day in MONTH interval= " e
mmdyy10.;
RUN;

A) 6 months from 3/20/2017 with default alignment = 09/01/2017
B) 6 months from 3/20/2017 aligned with beginning of MONTH interval= 09/01/2017
C) 6 months from 3/20/2017 aligned with middle of MONTH interval= 09/15/2017
D) 6 months from 3/20/2017 aligned with end of MONTH interval= 09/30/2017
E) 6 months from 3/20/2017 aligned with same day in MONTH interval= 09/20/2017

```

Example 2: INTNX() Function with Different Alignments

A) and B) set the result to the first day of the month, because that is the beginning of the MONTH interval. This is the basic calculation that the INTNX() function performs every time that it is called. From here, the alignment operators kick in. C) moves the date from 09/01/2017 to the middle of the month (14th for February in non-leap years, 15th for months with 29 or 30 days, and the 16th for months with 31 days.) D) shows that the result is now the last calendar day of the month, and E) takes the day used in the original argument. Even though the results may in some cases look as if the function just calculated those dates directly by using partial intervals, it is important to remember that the INTNX() function first advances the date supplied by the number of interval boundaries given, and then it adjusts the result based on the **alignment** argument. Once again, there is no such thing as a partial interval boundary.

BUT I DON'T WANT MY YEAR TO START ON JANUARY 1

Since the default starting point of an interval is at the beginning of it, SAS seems to have a blind spot when it comes to figuring out intervals that do not coincide with the standard definition. There is a way to shift the starting point of any given interval by creating your own interval definition. For example, what if you wanted to know the number of YEAR intervals between two dates, but instead of calculating calendar years, you wanted to calculate it using your company's fiscal year, which starts on July 1? You tell SAS how many periods to shift. Each interval has a shift unit. For years, the shift unit is months, so you will tell SAS to shift the starting point of the year in terms of months. A shifted interval is the interval name, followed by a period, and the number of periods to shift. To shift the start of the YEAR interval to July 1, you would use the interval "YEAR.7".

It is important to remember that when you count the number of periods to shift, you need to include the beginning of the period. We are not shifting the start of the year interval by six months to move to July; we are moving the start of the year to the seventh month. If you do not account for this, you will be off by one unit. Another handy way to think of it is that the YEAR.1 interval is the same as the YEAR interval. You can shift an interval by the maximum number of sub-intervals it contains. For example, you cannot shift a DAY interval, because the boundary is a DAY. However, you can shift a WEEK because the boundary is also a DAY. You can shift a WEEK by up to 7 days, which would start the week on Saturday. Refer to Appendix 1 at the end of the paper for the shift point associated with each type of interval.

I NEED AN INTERVAL THAT IS NOT IN SAS. WHAT CAN I DO?

SAS will allow you to define non-standard intervals in two different ways. One has always been available to you with intervals, and that is with interval multipliers. An interval multiplier essentially allows you to create an interval that is an integer multiple of the existing SAS interval definitions. For example, if you

want an interval where the boundaries occur every 10 years, use 'YEAR10' as the interval name. A 'MONTH4' interval would provide you with trimesters. Any interval you create using a multiplier functions the same as those that are already defined in SAS. The interval boundaries are set at the beginning of these intervals and you can shift the starting point of the interval using a shift index that is based on the shift point of the original SAS interval. Let's take the MONTH4 interval as an example. Instead of having the interval boundary occur on January 1 of each year, we want to set the interval so that the annual boundary falls on July 1 of each year. First, we need to figure out where the default boundaries for our interval lay. Handy trick: if you ever want to determine where an interval boundary starts relative to any date, use the INTNX() function with zero as the number of increments. Here, we're going to show all interval boundaries for 4 trimesters relative to January 1, 2016.

| Function Call | Date |
|------------------------------------|----------------------------|
| INTNX('MONTH4','01jan2016'd,0,'B') | Wednesday, January 1, 2016 |
| INTNX('MONTH4','01jan2016'd,1,'B') | Thursday, May 1, 2016 |
| INTNX('MONTH4','01jan2016'd,2,'B') | Monday, September 1, 2016 |
| INTNX('MONTH4','01jan2016'd,3,'B') | Thursday, January 1, 2017 |

Example 3: Interval Boundaries for Custom Intervals

It looks as if our MONTH4 starts on January 1 of the year. How does SAS figure out what point in time it should start calculating our intervals? Our old friend, January 1, 1960. This is an important point to be aware of for more esoteric intervals, but for intervals that easily fit into other SAS intervals, it won't be perceived as making a difference. So how do we move our interval so that one of the trimesters will always start on July 1? We can't shift the interval that starts in January by 7 months, as we would for the YEAR interval to move the start of it to July 1. You can only move an interval the number of shift-points that are contained in it. There are three shift points in our MONTH4 interval before we move on to the next MONTH4 interval. Given that, it makes sense to shift from May to July. Remember that a shift index of one has no effect on the starting point of an interval; therefore, we need to shift the start of the MONTH4 interval by 3 months (May, June, July). That results in our interval being MONTH4.3. as shown in Example 4:

| Function Call | Date |
|--------------------------------------|----------------------------|
| INTNX('MONTH4.3','01jan2017'd,0,'B') | Friday, November 1, 2016 |
| INTNX('MONTH4.3','01jan2017'd,1,'B') | Saturday, March 1, 2017 |
| INTNX('MONTH4.3','01jan2017'd,2,'B') | Tuesday, July 1, 2017 |
| INTNX('MONTH4.3','01jan2017'd,3,'B') | Saturday, November 1, 2017 |

Example 4: Using the SHIFT Index to Customize Interval Boundaries

Now, July 1 will always be an interval boundary for our MONTH4 interval. If you are wondering about the starting date of the interval, bear in mind that SAS is calculating intervals from the first interval boundary AFTER January 1, 1960. The first complete MONTH4.3 interval ending in 1960 ends in March of 1960. Move the date back by 4 months and you get November 1, 1959. Therefore, trimesters based on our MONTH4.3 interval will always start in the months of November, March, and July. This has allowed SAS programmers to create contiguous periods of time that are not already defined by SAS, and has historically been the only way to modify interval definitions.

The enhancement to customizing intervals is now you have the ability to create your own intervals using a SAS dataset. User-defined intervals now allow you to create intervals that do not correspond to shifted and/or multiplied standard SAS intervals. Intervals with irregular boundaries, or intervals that do not encompass an entire pre-defined SAS interval, or intervals that may be specific to your industry or company. As an example, you can create an HOUR8 interval to account for three eight-hour working shifts; furthermore, you can use HOUR8.7 to create intervals that correspond with shift start times of 6 a.m., 2 p.m., and 10 p.m. It's an easy way to count shifts. What happens if your working shifts are 10 hours long, with a mandatory 4-hour break while the machinery goes through an automated cleaning

cycle and reset? Although this may be an extreme example, try accommodating this with standard SAS intervals using multipliers and a shift index.

Another possibility is to create a user-defined interval that changes the shift point of an existing standard SAS interval. Consider that the shift point for a DAY is DAY, which starts at 12 midnight. If you want your "day" to start at 6 a.m. instead of midnight, it is not easy to do. You could use a DTHOUR24.7 interval, but that will only work with datetime values, and you would have to convert all of your date data to datetimes.

THE MECHANICS OF USER-DEFINED INTERVALS

Defining an interval requires a SAS dataset that has at least one variable named **begin**. It may also have two additional variables, **end** and **season**. If end is not included in the dataset, the end of one period in your interval is considered to be one less than the begin value of the next period in that interval. In general terms, if your user-defined interval is based on days, the endpoint of one period would be the day before the begin date of the next period. The optional season variable is for the concept of seasonality used in time-series analyses. The final requirement is that the **begin** (and **end**, if present) variables must be associated with an appropriate date, time, or datetime format in the dataset when it is created.

Once you have created your interval dataset, you need to tell SAS to use it as a user-defined interval as follows:

```
OPTIONS INTERVALDS (semester, custom.semester, proddays, custom.anyname) ;
```

This tells SAS that an interval named SEMESTER is created from the dataset CUSTOM.SEMESTER, and the interval PRODDAYS is created from the dataset CUSTOM.ANYNAME. I would strongly recommend naming your interval definition dataset the same as the interval it defines. If you need more user-defined intervals, just add more arguments to the INTERVALDS option statement. The rules for naming your intervals are standard: it must adhere to standard SAS naming conventions (32 characters or less, must start with a letter, etc.), and it cannot be a SAS reserved word—including names of standard SAS intervals. This prevents you from changing the definition of the DAY interval. You can create your own parallel interval, but it cannot be named DAY.

While this may be cause for much rejoicing in some quarters, as always, care must be taken when it comes to defining and using your own intervals, because unexpected results are lurking just around the corner. To demonstrate, let's take the example of a small, family-owned manufacturing plant. They are in production 5 days a week, not including weekends or the standard holidays. In addition, the generous owners give their employees the following additional days off: Christmas Eve, the day after Thanksgiving, the Friday before Easter, and the company's anniversary, which is August 6 of each year. In the event that this date (or Christmas Eve, Christmas, or New Year's Day) falls on a Saturday or Sunday, the day off is moved to Friday or Monday as appropriate. They also do not have production days the week before July 1 while they do inventory. Got all that?

Why such a complicated scenario? The company is known for high quality components and their ability to deliver just-in-time sourcing to clients at a very good price (with a 25% discount if they do not meet the stated delivery date.) Therefore, it is crucial the company knows how long production takes to the exact day. Appendix 2 holds the annotated code to build the dataset used to create the PRODDAYS interval.

First, let's look at the actual days, number of weekdays, and the number of production days for the calendar year 2017.

| start | stop | Actual Days | Weekdays | Production Days |
|-----------|-----------|-------------|----------|-----------------|
| 01JAN2017 | 01JAN2018 | 365 | 261 | 243 |

You can see the slight difference in the number of weekdays vs. the number of days that the production lines are working. To build and deliver each of the company's six product lines requires a different number of production days. A new customer comes in with an order on Friday, June 23, 2017, and wants to know what the delivery date would be for each type of product. The company determines how many production days it will take to fill the order, and then adds it to the order date. Since the contract calls for a substantial discount if the delivery date is not met, the company must be precise in its calculations. An

approximate delivery date is calculated with the INTNX() function, using the WEEKDAY interval, but the exact one will use our PRODDAYS interval.

| Product | Days to Produce Customer Order | Approximated Delivery Date Using WEEKDAY Interval | Actual Delivery Date Using PRODDAYS Interval |
|------------------|--------------------------------|---|--|
| Std Product 1 | 23 | Wednesday, July 26, 2017 | Thursday, August 3, 2017 |
| Std Product 2 | 32 | Tuesday, August 8, 2017 | Thursday, August 17, 2017 |
| Std Product 3 | 35 | Friday, August 11, 2017 | Tuesday, August 22, 2017 |
| Custom Product 1 | 33 | Wednesday, August 9, 2017 | Friday, August 18, 2017 |
| Custom Product 2 | 42 | Tuesday, August 22, 2017 | Thursday, August 31, 2017 |
| Custom Product 3 | 56 | Monday, September 11, 2017 | Thursday, September 21, 2017 |

Example 5: Delivery Dates using User-Defined Intervals

This is an example of what you can do with user-defined intervals. However, there is one specific problem when you define your own intervals. If there is no record for a specific date, time, or datetime in the interval dataset, and an interval function makes a calculation that results in a value you have not defined, you will get an unexpected result. This unexpected result may be a missing value, or, worse, it may not. Continuing with our small, family-owned company, and their PRODDAYS interval, what happens when the interval is defined through the end of 2015, but not beyond? Here is an example of using the INTCK() function to count the number of PRODDAYS from December 22, 2017:

| Obs | startDate | endDate | result |
|-----|-----------------------------|------------------------------|--------|
| 1 | Thursday, December 21, 2017 | Friday, December 22, 2017 | 0 |
| 2 | Thursday, December 21, 2017 | Saturday, December 23, 2017 | 0 |
| 3 | Thursday, December 21, 2017 | Sunday, December 24, 2017 | 0 |
| 4 | Thursday, December 21, 2017 | Monday, December 25, 2017 | 0 |
| 5 | Thursday, December 21, 2017 | Tuesday, December 26, 2017 | 1 |
| 6 | Thursday, December 21, 2017 | Wednesday, December 27, 2017 | 2 |
| 7 | Thursday, December 21, 2017 | Thursday, December 28, 2017 | 3 |
| 8 | Thursday, December 21, 2017 | Friday, December 29, 2017 | 4 |
| 9 | Thursday, December 21, 2017 | Saturday, December 30, 2017 | 4 |
| 10 | Thursday, December 21, 2017 | Sunday, December 31, 2017 | 4 |
| 11 | Thursday, December 21, 2017 | Monday, January 1, 2018 | 4 |
| 12 | Thursday, December 21, 2017 | Tuesday, January 2, 2018 | 4 |
| 13 | Thursday, December 21, 2017 | Wednesday, January 3, 2018 | 4 |
| 14 | Thursday, December 21, 2017 | Thursday, January 4, 2018 | 4 |
| 15 | Thursday, December 21, 2017 | Friday, January 5, 2018 | 4 |

Example 6: Result of Undefined Days in a User-Defined Interval

From this, we can see that the first four days after December 21 are not production days. This makes sense; the company is closed on Christmas Eve (holiday moved to Friday from Sunday), over the weekend, and Christmas Day. The remainder of the following week counts as you would expect until the weekend, followed by January 1, 2018, another company holiday. The following days aren't counted as production days. Shouldn't they be? Yes, but the dataset we used to create the interval only provides dates through December 31, 2017. Since those dates in 2018 aren't in the dataset, they aren't counted. This makes sense, after all, how did we remove days from the interval in the first place? We did not put them in the interval dataset, so they were not counted as a part of the PRODDAYS interval.

Of course, there are limitations with user-defined intervals: you cannot use a multiplier and/or a shift index, and their functionality is limited to the domain of interval functions. You can't use your user-defined interval to define axes for graphics, for example. Nonetheless, this is a new weapon in the arsenal of SAS intervals. It fills the gap of what to do when the standard SAS intervals along with multipliers and a shift index can't get you the results you need with intervals.

MORE ENHANCEMENTS

With version 9, SAS has added several more functions that work with, on, or provide information about, intervals. Many of these are going to be of particular interest to those who work with time series analysis, and some are of particular note for application developers. I will provide a brief compendium in this paper, but for more in-depth discussion, go to support.sas.com and look at the examples they provide to fuel your imagination as to how these new functions can benefit you.

INTERVAL FUNCTIONS ABOUT INTERVALS

INTFIT(*argument-1,argument-2,type*) This function will take two date, datetime, or observation numbers as the first two arguments and return the interval between them deemed to be "the best fit." This function assumes that the alignment is "SAME", so that the exact values are matched between the two arguments. *type* depends on the arguments. When the arguments are SAS date values, type should be "D". If the arguments are SAS datetime values, type should be "DT". If you are looking for the interval between two observations by providing observation numbers, function will return an interval with the base name of "OBS". You can use this function to give you the interval between two points in time. The intervals returned contain multipliers and a shift index, so the result returned may not be exactly what you were looking for. As you can see in the table below, some of the intervals are not as you might anticipate.

| | start | end | result |
|----------|-------------------------------|------------------------------|------------------------|
| A | Tuesday, April 1, 2014 | Wednesday, April 2, 2014 | DAY |
| B | Tuesday, April 1, 2014 | Thursday, April 3, 2014 | DAY2 |
| C | Tuesday, April 1, 2014 | Friday, April 4, 2014 | DAY3.3 |
| D | <i>Tuesday, April 1, 2014</i> | <i>Sunday, April 6, 2014</i> | <i>DAY5.5</i> |
| E | 5/1/2014 9:00 AM | 5/1/2014 2:00 PM | DTHOUR5 |
| F | <i>5/1/2014 9:00 AM</i> | <i>5/1/2014 8:45 PM</i> | <i>DTMINUTE705.136</i> |
| G | <i>5/1/2014 9:00 AM</i> | <i>5/2/2014 3:30 AM</i> | <i>DTMINUTE1110.61</i> |

Example 7: The INTFIT() Function

The italicized rows show examples of intervals that may be puzzling. Remember that the function provides the exact intervals between the pair of dates or datetimes, and the lowest common denominator will be used for base interval. In addition, the dates are measured using the SAMEDAY alignment, which means that the interval is measured from the beginning of the interval, and then adjusted. At first glance, the DAY5.5 result in row D doesn't seem to be right; Tuesday is only the third day of the week, so shouldn't the shift be 3 as it is for the DAY3 and DAY4 intervals? The start of DAY 5 intervals are Sunday, then Friday. If you want to measure a DAY5 interval to a Sunday from a Tuesday, then you will have to measure from the Friday, which gives you a shift of 5. Rows F and G demonstrate that it may become even more confusing with times and/or datetimes. The base interval shift points are now minutes and/or seconds, so the resulting shift index and multipliers may be large enough that you lose the context of the original reference point.

The following table shows a sample result of using the INTFIT() function with the 'OBS' type. The records are from a sequential dataset with Wednesday, April 2, 2014 as the first date. Since the dates are keyed by observation, the result of the INTFIX() function shows the interval as the relationship between observations, not the actual date values themselves.

| Sample Function Call | Date | Result |
|----------------------|--------------------------|---------|
| INTFIT(1,1,'OBS') | Wednesday, April 2, 2014 | |
| INTFIT(1,4,'OBS') | Saturday, April 5, 2014 | OBS3.2 |
| INTFIT(1,5,'OBS') | Sunday, April 6, 2014 | OBS4.2 |
| INTFIT(1,10,'OBS') | Friday, April 11, 2014 | OBS9.2 |
| INTFIT(1,14,'OBS') | Tuesday, April 15, 2014 | OBS13.2 |
| INTFIT(1,16,'OBS') | Thursday, April 17, 2014 | OBS15.2 |

Example 8: The INTFIT() Function with the OBS Argument

INTFMT('interval', 'size') This function will take the interval you give and return a suggested format for date, time, or datetime values using this interval. *interval* can be any standard SAS interval (with multipliers and/or a shift index if desired), or a user-defined interval that you have created. *size* refers to whether the format it returns will have a two-digit year ('short'/'s') or a four-digit year ('long'/'l'). This function is most useful when you use the INTFIT() or INTGET() functions to determine the interval dynamically, so that you will always display a result in the proper context. As of version 9.4, there is one important usage note about the *size* argument. If you do not use lower case, the function may return an unpredictable result (see Row A) in the example table below:

| | Sample Function Call | Result | Comment |
|---|------------------------|-------------|--------------------------------|
| A | INTFMT('WEEK','LONG') | | size argument is in upper case |
| B | INTFMT('WEEK','short') | WEEKDATX15. | |
| C | INTFMT('MONTH','long') | MONYY7. | |

Example 9: The INTFMT() Function

INTGET(argument1,argument2,argument3) This function will determine an interval from three date or datetime values you provide. The arguments can be variables or date/datetime literals, but they must be of the same type; you cannot mix dates and datetimes as arguments. The function calculates all intervals between the first two arguments, and then between the second and third arguments. If the intervals are the same, it will return that interval. If the intervals are not the same, then the function will test the interval between the second and third arguments to see if it is a multiple of the interval between the first two arguments. If this is true, then the function will return the smaller of the two intervals. If neither of these cases are true, then the INTGET() function will return a missing value.

| | Sample Function Call | Result |
|---|--|-----------|
| A | INTGET('05SEP2013'd,20SEP2013'd,05OCT2013'd) | SEMIMONTH |
| B | INTGET('15JAN2014'd,15APR2014'd,15OCT2015'd) | QTR |
| C | INTGET('15JAN2014'd,15APR2015'd,15OCT2015'd) | |

Example 10: The INTGET() Function

In row A, the period between the dates is 15 days, which corresponds to the SEMIMONTH interval. Row B demonstrates that the dates do not have to have the same interval. The gap between January and April is one quarter, but the gap between April and October of the following year is five quarters. Since the interval for the second pair of arguments would be 'QTR5', since it is a multiple of the QTR interval, the function returns the interval without a multiplier. What happened in row C? Simply, the interval between January 2014 and April 2015 is now QTR5, but the interval between April and October of 2015 is QTR2. Since QTR2 is not a multiple of QTR5, the result is missing.

INTSHIFT('interval') This function will take the interval you give and return the shift point for that interval. *interval* can be any standard SAS interval (with multipliers and/or a shift index if desired.) If *interval* is represented by a character string, then it must be enclosed in quotes, but you may use a character variable containing the name of an interval instead. If you try to use this function with a user-defined interval you have created, the function will return a missing value.

INTTEST('interval') This function will take the interval you give and return a 1 if it is a valid interval name, or 0 if it is not. You can use this to determine if you have created a valid interval with multipliers and/or a shift index. *interval* can be any standard SAS interval (with multipliers and/or a shift index if desired.) If *interval* is represented by a character string, then it must be enclosed in quotes, but you may use a character variable containing the name of an interval instead. If you try to use this function with a user-defined interval you have created, the function will return a missing value.

| Sample Function Call | Result | Comment |
|----------------------|--------|---|
| INTTEST('WEEK') | 1 | |
| INTTEST('QTR.1') | 1 | You can shift a quarter by up to 3 months (counting the initial starting point as .1). |
| INTTEST('QTR3.13') | 0 | The shift point for quarters is months, so the maximum number of months you can shift in a 3-quarter period would be 9. |
| INTTEST('YEAR.7') | 1 | |

Example 11: The INTTEST() Function

RETAIL CALENDAR INTERVALS

SAS has added intervals that are specifically designed for the retail industry. These intervals are ISO 8601-compliant, and can be used with any of the interval functions. They facilitate comparisons across years, because the weeks remain consistent between years. In order to facilitate this, some years will have leap weeks. Year definitions are based on the ISO 8601 definition of a week, which is the first Monday preceding January 4, which in some cases may place the beginning of the week in December. These intervals allow you to define the structure of your 52-week year, expecting that for the first 13-week period of your interval, there will be one month that has 5 weeks in it. This means that you can set the month pattern to 5-4-4, 4-5-4, or 4-4-5. You can work with years, months, or quarters in this fashion. The full list of retail intervals is in Appendix 3.

SEASONALITY FUNCTIONS

Seasonality is used in time-series analysis, and can be used in SAS/ETS®. It helps to account for normal seasonal variations in patterns inside of an analysis. While it is not strictly a date and time matter, it does use intervals; therefore, the seasonality functions are documented here, but without context. For more information on seasonality and its application, you can refer to the SAS/ETS documentation, and support.sas.com is a great place to find more help on this topic.

INTINDEX('interval',argument) This function returns the index of the seasonal cycle based on *interval* for *argument*, where *argument* is a SAS date, time or datetime value. *interval* can be any standard SAS interval (with multipliers and/or a shift index if desired).

INTCYCLE('interval',seasonality) This function returns the interval of the seasonal cycle. *interval* can be any standard SAS interval (with multipliers and/or a shift index if desired. *seasonality* is an optional argument that allows you to define seasonal cycles, and it can be a number or a cycle (such as 'YEAR'). For example, you can use the *seasonality* argument to specify your year as having 53 weeks instead of 52.

INTINDEX('interval',argument,seasonality) This function returns the seasonal index when given an interval, a SAS date, time, or datetime. *interval* can be any standard SAS interval (with multipliers and/or a shift index if desired. *argument* is a SAS date, time, or datetime value. Note that the interval specified must be appropriate for the *argument*. *seasonality* is an optional argument that allows you to define seasonal cycles, and it can be a number or a cycle (such as 'YEAR'.) The INTINDEX function returns the seasonal index, while the INTCINDEX function returns the cycle index.

INTSEAS('interval',seasonality) This function returns the number of intervals in a seasonal cycle. *interval* can be any standard SAS interval (with multipliers and/or a shift index if desired.) *seasonality* is an optional argument that allows you to define seasonal cycles. This is a good function to be aware of if

you are new to concepts of seasonality, because the number of intervals in a seasonal cycle may not be intuitive. As an example, while you might expect the number of intervals for the QTR interval to be 4 (because there are 4 quarters in a year), the number of intervals for a DAY interval is 7, for the number of days in the week. This can help you check your expectations.

SUMMARY

Version 9 of SAS has seen some significant additions to its interval capacity. The addition of a CONTINUOUS method to the INTCK function allows you to count elapsed time in a more intuitive way, although the default DISCRETE still has its place, and you shouldn't be too quick to adopt the new method for all situations. The upgrade to the INTNX function has been the addition of the SAMEDAY adjustment, which lets you calculate to a date (or datetime), and find the same day as in the original argument within that interval. One of the biggest enhancements to intervals is the ability to define your own interval, which expands your control over the interval you need by expanding well beyond the previous boundaries of creating multiples of existing intervals and shifting starting points.

Automation has been considered as well, with a series of functions about intervals themselves. Now you can find an interval between two or three points, and get a format for a value based on the interval you're using. A function now exists to test if your interval is valid, allowing you to take graceful action to handle an exception within an application rather than crashing the process because of an invalid interval name. You can also determine whether a shift-index is valid or not, so a user can't try to shift the YEAR interval by 14 months.

For those of you in retail, SAS has added intervals just for your industry to help with those analyses and projections, and for the SAS/ETS user, there are now a group of functions to help with the concept of seasonality, which may also be accounted for in any user-defined intervals you create. While these enhancements may not be a major tidal change in the way many of us deal with intervals on a day-to-day basis, I'm sure that some of us have already taken advantage of them, and many more will do so in the near future.

REFERENCES

Morgan, Derek P. 2014, *The Essential Guide to SAS® Dates and Times Second Edition*. Cary, NC: SAS Institute Inc.

CONTACT INFORMATION:

Further inquiries are welcome to:

Derek Morgan
E-mail: mrdatesandtimes@gmail.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. Other brand and product names are trademarks of their respective companies.

APPENDIX 1: SAS INTERVALS AND THEIR STARTING POINTS

| Interval Name | Definition | Default Starting Point |
|------------------------|---|---|
| DAY | Daily intervals | Each day |
| WEEK | Weekly intervals of seven days | Each Sunday |
| WEEKDAY <i>daysW</i> | Daily intervals with Friday-Saturday-Sunday counted as the same day (five-day work week with a Saturday-Sunday weekend). <i>days</i> identifies the individual numbers of the weekend day(s) by number (1=Sunday ... 7=Saturday). By default, <i>days</i> ="17", so the default interval is WEEKDAY17W. | Each day |
| TENDAY | Ten-day intervals (a U.S. automobile industry convention) | 1 st , 11 th , and 21 st of each month |
| SEMIMONTH | Half-month intervals | 1 st and 16 th of each month |
| MONTH | Monthly intervals | 1 st of each month |
| QTR | Quarterly (three-month) intervals | 1-Jan 1-Apr 1-Jul 1-Oct |
| SEMIYEAR | Semi-annual (six-month) intervals | 1-Jan 1-Jul |
| YEAR | Yearly intervals | 1-Jan |
| DTDAY | Daily intervals used with datetime values | Each day |
| DTWEEK | Weekly intervals of seven days used with datetime values | Each Sunday |
| DTWEEKDAY <i>daysW</i> | Daily intervals with Friday-Saturday-Sunday counted as the same day (five-day work week with a Saturday-Sunday weekend.) <i>days</i> identifies the individual weekend days by number (1=Sunday ... 7=Saturday.) By default, <i>days</i> ="17", so the default interval is DTWEEKDAY17W. This interval is only used with datetime values. | Each day |
| DTTENDAY | Ten-day intervals (a U.S. automobile industry convention) used with datetime values | 1 st , 11 th , and 21 st of each month |
| DTSEMIMONTH | Half-month intervals used with datetime values | 1 st and 16 th of each month |
| DTMONTH | Monthly intervals used with datetime values | 1 st of each month |
| DTQTR | Quarterly (three-month) intervals used with datetime values | 1-Jan 1-Apr 1-Jul 1-Oct |
| DTSEMIYEAR | Semiannual (six-month) intervals used with datetime values | 1-Jan 1-Jul |
| DTYEAR | Yearly intervals used with datetime values | 1-Jan |
| DTSECOND | Second intervals used with datetime values | Seconds |
| DTMINUTE | Minute intervals used with datetime values | Minutes |
| DTHOUR | Hour intervals used with datetime values | Hours |
| SECOND | Second intervals used with time values | Seconds |
| MINUTE | Minute intervals used with time values | Minutes |
| HOUR | Hourly intervals used with time values | Hours |

APPENDIX 2: CREATING THE DATASET TO MAKE THE PRODDAYS INTERVAL

```
1  OPTIONS INTERVALDS=(proddays=proddays);
2  DATA proddays (KEEP=begin);

3  /* Only need BEGIN variable, assume end-of-day as same day */
4  start = '01JAN2017'D;
5  stop  = '31DEC2017'D;
6  nweekdays = INTCK('WEEKDAY',start,stop);
7  DO i = 0 TO nweekdays;
8      begin = INTNX('WEEKDAY',start,i); /* weekdays between start/stop */
9      year = YEAR(begin);

10     /* Calculate inventory days */
11     invend = MDY(6,30,year);
12     IF 1 < WEEKDAY(invend) < 7 THEN
13         invstrt = INTNX('WEEKDAY',invend,-4);
14     ELSE
15         invstrt = INTNX('WEEKDAY',invend,-5);

16     /* Company-specific holidays */
17     nye = HOLIDAY('NEWYEAR',year);
18     xthanks = HOLIDAY("THANKSGIVING",year) + 1; /* Thanksgiving Friday*/
19     xmas = HOLIDAY('CHRISTMAS',year); /* Christmas */
20     xmaseve = xmas - 1; /* Christmas Eve */
21     sprng = HOLIDAY("EASTER",year) - 2; /* Friday before Easter */

22     /* Founders Day - If on weekend, move forward/back as appropriate */
23     founders = MDY(8,6,year);
24     SELECT(WEEKDAY(founders));
25         WHEN(7) founders = founders - 1;
26         WHEN(1) founders = founders + 1;
27         OTHERWISE founders = founders;
28     END;

29     /* Also need to adjust New Year, Christmas, and Christmas Eve if */
30     /* they fall on weekend days
31     SELECT(WEEKDAY(xmaseve)); /* Move Christmas Eve to preceding Fri */
32         WHEN(7) xmaseve = xmaseve - 1;
33         WHEN(1) xmaseve = xmaseve - 2;
34         OTHERWISE xmaseve = xmaseve;
35     END;
36     SELECT(WEEKDAY(xmas)); /* Move to following Mon */
37         WHEN(7) xmas = xmas + 2;
38         WHEN(1) xmas = xmas + 1;
39         OTHERWISE xmas = xmas;
40     END;
41     SELECT(WEEKDAY(nye)); /* Move to following Mon */
42         WHEN(7) nye = nye + 2;
43         WHEN(1) nye = nye + 1;
44         OTHERWISE nye = nye;
45     END;
46
```

```

47 /* Exclude "normal" holidays, company-added holidays and inventory time */
48     IF begin NE nye AND
49         begin NE HOLIDAY("MLK",year) AND
50         begin NE HOLIDAY("USPRESIDENTS",year) AND
51         begin NE HOLIDAY("MEMORIAL",year) AND
52         begin NE HOLIDAY("USINDEPENDENCE",year) AND
53         begin NE HOLIDAY("LABOR",year) AND
54         begin NE HOLIDAY("VETERANS",year) AND
55         begin NE HOLIDAY("THANKSGIVING",year) AND
56         begin NE xmas AND
57         begin NE xmaseve AND
58         begin NE xthanks AND
59         begin NE sprng AND
60         begin NE founders AND
61         (begin < invstrt or begin > invend) THEN OUTPUT;
62     END;

63 /* MUST associate begin and end with a date, time or datetime format */
64 /* as appropriate, otherwise user-defined interval will not work */
65     FORMAT begin WEEKDATE.;
66     RUN;

```

APPENDIX 3: RETAIL INDUSTRY INTERVALS

| | |
|---------|---|
| YEARV | specifies ISO 8601 yearly intervals. The ISO 8601 year begins on the Monday on or immediately preceding January 4. Note that it is possible for the ISO 8601 year to begin in December of the preceding year. Also, some ISO 8601 years contain a leap week. The beginning subperiod s is written in ISO 8601 weeks (WEEKV). |
| R445YR | is the same as YEARV except that in the retail industry the beginning subperiod s is 4-4-5 months (R445MON). |
| R454YR | is the same as YEARV except that in the retail industry the beginning subperiod s is 4-5-4 months (R454MON). |
| R544YR | is the same as YEARV except that in the retail industry the beginning subperiod s is 5-4-4 months (R544MON). |
| R445QTR | specifies retail 4-4-5 quarterly intervals (every 13 ISO 8601 weeks). Some fourth quarters contain a leap week. The beginning subperiod s is 4-4-5 months (R445MON). |
| R454QTR | specifies retail 4-5-4 quarterly intervals (every 13 ISO 8601 weeks). Some fourth quarters contain a leap week. The beginning subperiod s is 4-5-4 months (R454MON). |
| R544QTR | specifies retail 5-4-4 quarterly intervals (every 13 ISO 8601 weeks). Some fourth quarters contain a leap week. The beginning subperiod s is 5-4-4 months (R544MON). |
| R445MON | specifies retail 4-4-5 monthly intervals. The 3rd, 6th, 9th, and 12th months are five ISO 8601 weeks long with the exception that some 12th months contain leap weeks. All other months are four ISO 8601 weeks long. R445MON intervals begin with the 1st, 5th, 9th, 14th, 18th, 22nd, 27th, 31st, 35th, 40th, 44th, and 48th weeks of the ISO year. The beginning subperiod s is 4-4-5 months (R445MON). |
| R454MON | specifies retail 4-5-4 monthly intervals. The 2nd, 5th, 8th, and 11th months are five ISO 8601 weeks long with the exception that some 12th months contain leap weeks. R454MON intervals begin with the 1st, 5th, 10th, 14th, 18th, 23rd, 27th, 31st, 36th, 40th, 44th, and 49th weeks of the ISO year. The beginning subperiod s is 4-5-4 months (R454MON). |
| R544MON | specifies retail 5-4-4 monthly intervals. The 1st, 4th, 7th, and 10th months are five ISO 8601 weeks long. All other months are four ISO 8601 weeks long with the exception that some 12th months contain leap weeks. R544MON intervals begin with the 1st, 6th, 10th, 14th, 19th, 23rd, 27th, 32nd, 36th, 40th, 45th, and 49th weeks of the ISO year. The beginning subperiod s is 5-4-4 months (R544MON). |
| WEEKV | specifies ISO 8601 weekly intervals of seven days. Each week begins on Monday. The beginning subperiod s is calculated in days (DAY). Note that WEEKV differs from WEEK in that WEEKV.1 begins on Monday, WEEKV.2 begins on Tuesday, and so on. |