Paper 129-2023

# Demystifying Intervals

Derek Morgan, Bristol Myers Squibb

## 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.

## INTRODUCTION

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's 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 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. If you're asked to create quarterly earnings reports, you aren't being asked to report earnings for every 90 days. A quarter is only 90 days long when it's the first quarter of a leap year. I have a hunch that making that every-90-day report won't go over well.

Manipulating data with intervals differs from the simple math you use with date data; instead of mathematical operations, you manipulate date and time data with a series of SAS functions that operate on dates, times, and datetimes using the intervals you specify. The SAS System has several intervals already defined to correspond to standard definitions. 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 creates a "century" (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 July 1 (the seventh month), 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. This paper also refers to capabilities available in SAS version 9. Some of the additional functionality described here does not exist prior to version 9.

The mainstays of the SAS interval facility are 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 use this function to determine how many intervals have elapsed between two dates. The INTNX() function will take a date, time, or datetime 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 point.

## A WORD OF CAUTION

Intervals are just like every other part of the SAS date, time, and datetime handling. The results are just numbers to SAS; you are responsible for providing the context to interpret those numbers correctly. That

means you can only use date intervals with SAS dates, time intervals with SAS time values, and datetime intervals with SAS datetime values. If you mix this up, your results will be unpredictable, and unless you're extremely lucky, they'll be wrong.

## WHEN IS A YEAR NOT A YEAR?

Let's start with the INTCK() function. 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 calculates the number of ***intervals*** between ***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 also 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. It counts interval boundaries. On the other hand, the CONTINUOUS method counts actual elapsed periods of time. A simple way to understand the CONTINUOUS method is that it counts intervals the same 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 changing it to the more intuitive method.

To illustrate the potential for problems with SAS intervals, consider Example 1. Note the result in bold:

**Example 1: Why Intervals Can Be Confusing**

```
DATA _NULL_;
   v1 = INTCK('YEAR','01jan2022'd,'01jan2023'd);
   v2 = INTCK('YEAR','31dec2022'd,'01jan2023'd);
   PUT v1= +3 v2=;
RUN;
v1=1    v2=1
```

Now wait a minute. We know that a year from December 31, 2022, is not January 1, 2023. What happened? SAS intervals are not a shortcut for doing math. When the DISCRETE method is used (either explicitly or by default,) the INTCK() function counts the number of times that the 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. The YEAR interval always starts on January 1. Therefore, given any starting date in 2022, INTCK() will only count a single YEAR interval for any given date in 2023 because it only passes January 1, 2023 once. There is an enormous potential for bad results if you misunderstand how INTCK() calculates using its default (DISCRETE) method. This is the most confusing part of intervals, and once you get it, you'll find working with intervals becomes much easier. The other important thing to remember about intervals is that intervals are based on the starting point of the interval you're using, and not the start-of-period date you give to INTCK().

Now, let's try the same two dates with the CONTINUOUS method in Example 2:

**Example 2: INTCK() Calculation Using the CONTINUOUS Method**

```
DATA _NULL_;
   v1 = INTCK('YEAR','01jan2022'd,'01jan2023'd,'C');
   v2 = INTCK('YEAR','31dec2022'd,'01jan2023'd,'C');
   PUT v1= +3 v2=;
RUN;

v1=1    v2=0
```

This looks like what you'd expect. One year elapsed from January 1, 2022 to January 1, 2023, but not between December 31, 2022 and January 1, 2023. 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 isn't considered a complete month when it's a leap year, or ***start-of-period*** is in any month other than February. When you use the INTCK() function with the CONTINUOUS method, it won't count a month for the 30 days between January 1 and January 31, but it will for the 30 days between April 1 and May 1.

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 should be cautious in choosing which of the two methods you select. Again, don't change the method for any legacy code that uses the INTCK() function just because the CONTINUOUS method makes more sense to you. That legacy code was written to take advantage of the DISCRETE method, and as shown by Example 1 and Example 2, the two methods are not equivalent.

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

**Table 1: Sample INTCK() Function Calls Using DISCRETE Method**

| Function Call | Result |
|---|---|
| INTCK('DAY', '31dec2020'd, '06jan2021'd, 'D') | 6 |
| INTCK('WEEK', '31dec2020'd, '06jan2021'd, 'D') | 1 |
| INTCK('MONTH', '31dec2020'd, '06jan2021'd, 'D') | 1 |
| INTCK('YEAR', '31dec2020'd, '06jan2021'd, 'D') | 1 |

Although 7 days have elapsed, only six DAY interval boundaries have been crossed since December 31, 2020 (01/01/2021, 01/02, 01/03, 01/04, 01/05, and 01/06.) The start of the WEEK interval for January 6, 2021 is Sunday, January 3 (week intervals start on Sunday by default, although you can change this with a shift index.) The MONTH and YEAR intervals for January 6, 2021 also start on January 1, 2021, 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. The INTCK() function cannot 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. For example, you won't get 1.5 when you count the number of years between July 1, 2022 and January 1, 2024, However, you could use a MONTH6 interval and get 3, as shown in Example 3:

**Example 3: No Such Thing as Partial Intervals**

| Row | Function Call | Result | Days |
|---|---|---|---|
| 1 | INTCK('YEAR','01jul2022'd,'01jan2024'd,'D'); | 2 | 549 |
| 2 | INTCK('MONTH6','01jul2022'd,'01jan2024'd,'D'); | 3 | 549 |
| 3 | INTCK('MONTH','01jul2022'd,'01jan2024'd,'D'); | 18 | 549 |

You might wonder why the result in the first row is 2. The year boundaries are January 1, 2023 and January 1, 2024. Again, this is how intervals have worked in SAS since their inception. What happens if you use the CONTINUOUS method?

**Example 4: Still No Such Thing as Partial Intervals**

| Row | Function Call | Result | Days |
|---|---|---|---|
| 1 | INTCK('YEAR','01jul2022'd,'01jan2024'd,'C'); | 1 | 549 |
| 2 | INTCK('MONTH6','01jul2022'd,'01jan2024'd,'C'); | 3 | 549 |
| 3 | INTCK('MONTH','01jul2022'd,'01jan2024'd,'C’); | 18 | 549 |

This time, row 1 gives you the “expected” result of 1. SAS tests if a period of 365 days (366 if a leap year) has elapsed using the CONTINUOUS method. It would be 2 if 730 days had elapsed. However, it did not calculate 730 divided by 549 (you can do this yourself with math if you need to.)

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 can be thought of as the way most people think of elapsed time, and it’s good for counting anniversaries and milestones. The methods are NOT interchangeable, and it is a very bad idea to change existing code to use the new method.

## 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*);**

***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 the function will only use the integer portion of the parameter if it isn’t an integer, 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*** parameter. If you need to calculate the last day of a month, this would be the function you’d use to do it. To illustrate how alignment works with the INTNX() function, Example 5 provides a sample program that adds six months to March 20, 2023 with varying alignment values. The result is in bold.

**Example 5: INTNX() Function with Different Alignments**

```
DATA _NULL_;
a = INTNX('MONTH','20MAR2023'd,6);
b = INTNX('MONTH','20MAR2023'd,6,'B');
c = INTNX('MONTH','20MAR2023'd,6,'M');
d = INTNX('MONTH','20MAR/2023'd,6,'E');
e = INTNX('MONTH','20MAR/2023'd,6,'S');
PUT "A) 6 months from 3/20/2023 with default alignment = " a mmddyy10.;
PUT "B) 6 months from 3/20/2023 aligned with beginning of MONTH interval= " b
   mmddyy10.;
PUT "C) 6 months from 3/20/2023 aligned with middle of MONTH interval= " c
   mmddyy10.;
PUT "D) 6 months from 3/20/2023 aligned with end of MONTH interval= " d mmddyy10.;
PUT "E) 6 months from 3/20/2023 aligned with same day in MONTH interval= " e
   mmddyy10.;
RUN;
```

**A) 6 months from 3/20/2023 with default alignment = 09/01/2023**
**B) 6 months from 3/20/2023 aligned with beginning of MONTH interval= 09/01/2023**
**C) 6 months from 3/20/2023 aligned with middle of MONTH interval= 09/15/2023**
**D) 6 months from 3/20/2023 aligned with end of MONTH interval= 09/30/2023**
**E) 6 months from 3/20/2023 aligned with same day in MONTH interval= 09/20/2023**

A) and B) set the result to the first day of the month, because it's the beginning of the MONTH interval. The INTNX() function performs this calculation every time it is called. The alignment operators always kick in **after** this calculation. C) moves the date from 09/01/2023 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 look as if the function just calculated those dates directly by using partial intervals, it is important to remember that the INTNX() function always calculates the date from the number of interval boundaries parameter first, 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 would seem to have a blind spot when it comes to figuring out intervals that do not coincide with that. For example, what if you want to know the number of YEAR intervals between two dates, but instead of calculating calendar years, you wanted to calculate your company's fiscal year, which starts on April 1? You can easily move the starting point of any given interval by telling SAS how many shift units to move. Each interval has a shift unit. For years, the shift unit is months, so you 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 shift unit corresponding to the starting point you want. As an example, to move the start of the YEAR interval to February 1, you would use the interval "**YEAR.2**".

Why not use "**YEAR.1**"? The interval "**YEAR.1**" will move the starting point of the year to the first shift unit for YEAR intervals… which is January. You aren't adding one month to the start of the year interval to move it to February. You're moving the start of the year to the second month. Another handy way to remember this is that the starting point of any ***interval.1*** is the same as the starting point of ***interval***. By the way, there is no shift index equal to zero. SAS will give you an error if you try something like "YEAR.0".

You can only shift an interval by the maximum number of sub-intervals it contains. For example, you cannot shift a DAY interval, because the shift point is also a DAY. However, you can shift a WEEK because the shift point is a DAY. There are seven days in a week, so 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?

You can define non-standard intervals in SAS 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. As you have seen earlier, 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 each interval, and you can move the starting point of the interval using a shift index that is based on the shift point of the original SAS interval.

Let's use a trimester interval (MONTH4) interval as an example. Instead of having the interval boundary occur on January 1 of each year, let's set the interval so that the annual boundary falls on July 1 of each year. First, you need to figure out where the default boundaries for the MONTH4 interval are. 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. Table 2 shows the interval boundaries for 4 trimesters relative to January 1, 2022.

**Table 2: Interval Boundaries for Custom Intervals**

| Function Call | Date |
|---|---|
| INTNX('MONTH4','01jan2022'd,0,'B') | Saturday, January 1, 2022 |
| INTNX('MONTH4','01jan2022'd,1,'B') | Sunday, May 1, 2022 |
| INTNX('MONTH4','01jan2022'd,2,'B') | Thursday, September 1, 2022 |
| INTNX('MONTH4','01jan2022'd,3,'B') | Sunday, January 1, 2023 |

It looks like 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, you won't see any difference. So how can you move this trimester interval so that one of the trimesters will always start on July 1? Unlike the YEAR interval, you 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 is the same as the starting point of an interval; therefore, you need to shift the start of the MONTH4 interval by 3 months (May to June to July). That results in the interval being MONTH4.3. as shown in Table 3:

**Table 3: Using the SHIFT Index to Customize Interval Boundaries**

| Function Call | Date |
|---|---|
| INTNX('MONTH4.3','01jan2022'd,0,'B') | Monday, November 1, 2021 |
| INTNX('MONTH4.3','01jan2022'd,1,'B') | Tuesday, March 1, 2022 |
| INTNX('MONTH4.3','01jan2022'd,2,'B') | Friday, July 1, 2022 |
| INTNX('MONTH4.3','01jan2022'd,3,'B') | Tuesday, November 1, 2022 |

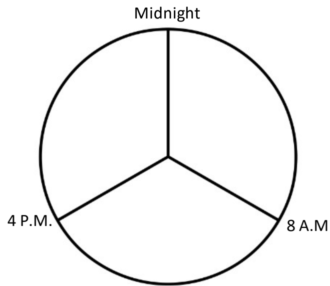
Using MONTH4.3 ensures July 1 will always be an interval boundary for our trimester interval. If you're wondering about the starting date of this interval (first row), bear in mind that SAS calculates intervals from the first interval boundary *containing* January 1, 1960, not the first interval boundary *after* January 1, 1960. The MONTH4.3 interval *containing* January 1, 1960 started on November 1, 1959, so the MONTH4.3 interval always starts on November 1 (4 months before March 1, which is the starting point of the first complete interval after January 1.). Therefore, trimesters based on a MONTH4.3 interval will always start in the months of November, March, and July.

In speaking about shifting and multiplying intervals, one of the most frequent questions I get is, "why can't you shift by more than the number of shift points in an interval?" The simple answer is that intervals

repeat. Essentially, every year is its own year interval. If you were to try to use an interval such as YEAR.14, you'd get an error because you can only shift the start point of a year by a maximum of 12 months. What if you wanted your year interval to start in the second month of the next year? Intervals don't measure specific dates, they measure periods of time, so once you hit the end of that period, the next period starts. A year interval is 12 months long, and at the end of those 12 months, another year interval starts. This can be even more confusing when it comes to multiplied intervals.

To demonstrate, let's use an HOUR8 interval, which breaks 24 hours into three 8-hour periods as shown in Figure 1:

**Figure 1: Day Broken into Three 8-Hour Periods**



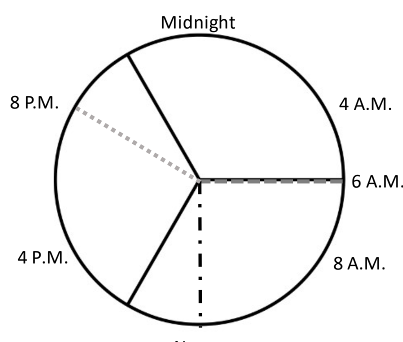

By default, the HOUR8 intervals start at midnight, eight in the morning, and four in the afternoon. What if you want your intervals to start at a different time? You'd use a shift index as discussed above. Since there are three eight-hour periods in a day, there will always be three starting points for the HOUR8 interval. Therefore, if you want your HOUR8 interval to start at six a.m., you'll use the HOUR8.7 interval (again, you're moving to the start of the seventh hour) as shown in Table 4:

**Table 4: Shifting the Start Points of an HOUR8 Interval**

| Interval Name | First Start Point of Interval within a 24-hour Day | Second Start Point of Interval within a 24-hour Day | Third Start Point of Interval within a 24-hour Day |
|---|---|---|---|
| **HOUR8.1** | 12:00:00 AM | 8:00:00 AM | 4:00:00 PM |
| **HOUR8.2** | 1:00:00 AM | 9:00:00 AM | 5:00:00 PM |
| **HOUR8.3** | 2:00:00 AM | 10:00:00 AM | 6:00:00 PM |
| **HOUR8.4** | 3:00:00 AM | 11:00:00 AM | 7:00:00 PM |
| **HOUR8.5** | 4:00:00 AM | 12:00:00 PM | 8:00:00 PM |
| **HOUR8.6** | 5:00:00 AM | 1:00:00 PM | 9:00:00 PM |
| ***HOUR8.7*** | *6:00:00 AM* | *2:00:00 PM* | *10:00:00 PM* |
| **HOUR8.8** | 7:00:00 AM | 3:00:00 PM | 11:00:00 PM |

So even when you choose HOUR8.7, you'll still have two more HOUR8.7 intervals within a 24-hour day. One starts at 2 in the afternoon and the other at ten at night. If you want your interval to start at 10 p.m., it's going to be the same HOUR8.7, because an eight-hour period doesn't take days into account. As shown in Figure 2, you're just rotating the circle.

**Figure 2: What Happens When You Shift an Interval (Rotating the Circle)**




I hope this clarifies how multipliers and shift indexes work. They can give you quite a bit of power when you work with SAS intervals. Multipliers and shift indexes have allowed SAS programmers to create contiguous periods of time that aren't already defined by SAS. Historically, this has been the only way to modify interval definitions, as shown by the examples in Figure 1, Table 4, and Figure 2. An HOUR8 interval is an easy way to handle the case of three working shifts in a 24-hour day. As you've seen, you would use HOUR8.7 to create intervals with start times of 6 a.m., 2 p.m., and 10 p.m.

Now, with version 9, you can create your own customized intervals using a SAS dataset. These user-defined intervals allow you to create intervals that don't correspond to shifted and/or multiplied standard SAS intervals. You can build intervals with irregular boundaries, or that don't encompass an entire pre-defined SAS interval, or intervals specific to your industry or company.

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, you can't easily do this with standard SAS intervals using only a multiplier 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 run from 6 a.m. to 6 p.m. instead of midnight to midnight, it isn't easy to do. You can't use HOUR.7 because that will only work with time values, and you'd lose track of the changing dates. You would need to use a DTHOUR24.7 interval, because that works with datetime values and would keep track of the dates. However, you'd first have to convert all your date data to datetime data.

## THE MECHANICS OF USER-DEFINED INTERVALS

Creating a custom, user-defined 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 the begin value of the next record in your dataset minus one. 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 and retail 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 an interval as follows:

**OPTIONS INTERVALDS(*name-of-custom-interval, dataset-name-describing-interval*);**

Both arguments must be present. I strongly recommend naming your interval definition dataset the same as the interval it defines. If you need more than one user-defined interval, just add more paired arguments to the INTERVALDS option statement. If you needed to create two custom intervals, one for semester and one for the number of production days available for the factory, it would look like Sample Code 1:

**Sample Code 1: Setting up Two Custom Intervals in SAS**

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

This tells SAS that a custom interval named SEMESTER is created from the dataset CUSTOM.SEMESTER, and the custom interval PRODDAYS is created from the dataset CUSTOM.ANYNAME. The rules for naming your intervals are the usual ones: 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.

To demonstrate a use for custom intervals, 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, New Year's Eve, the day after Thanksgiving, the Friday before Easter, and the company's anniversary, which is August 6 of each year. When this date (or Christmas Eve, or New Year's Eve) 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. They give their clients a 25% discount if they don't meet their stated delivery date. Therefore, it is crucial the company knows exactly how long production takes down to the specific day. Appendix 2 holds the code to build the dataset used to create the PRODDAYS interval. Feel free to copy and run it to try this out.

First, let's look at the actual days, number of weekdays, and the number of production days for the calendar year 2022 in Table 5:

**Table 5:Actual Days vs. Weekdays vs. Custom Production Days in 2022**

| start | stop | Actual Days | Weekdays | Production Days |
|---|---|---|---|---|
| 01JAN2022 | 01JAN2023 | 365 | 260 | 242 |

You can see the slight difference in the number of weekdays vs. the number of days that the production lines are working. Each of the company's products requires a different number of production days from order to delivery. A new customer comes in with an order for six of these products on Monday, June 6, 2022, 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. Table 6 shows the time required for each item in the customer's order, the delivery date when using the WEEKDAY interval, and the exact one using the PRODDAYS interval.

**Table 6: Delivery Dates using User-Defined Intervals**

| Product ID | Days to Deliver | Delivery Date Using WEEKDAY Interval | Delivery Date Using PRODDAYS Interval |
|---|---|---|---|
| **001** | 35 | Monday, July 25, 2022 | Tuesday, August 2, 2022 |
| **003** | 65 | Monday, September 5, 2022 | Thursday, September 15, 2022 |
| **066** | 33 | Thursday, July 21, 2022 | Friday, July 29, 2022 |
| **069** | 32 | Wednesday, July 20, 2022 | Thursday, July 28, 2022 |
| **086** | 42 | Wednesday, August 3, 2022 | Friday, August 12, 2022 |
| **097** | 23 | Thursday, July 7, 2022 | Friday, July 15, 2022 |

Product 097 only takes 23 days, but they're not working the weekdays from June 27-June 30 (inventory), or on the 4$^{th}$ of July. The biggest difference is in product 003, because of inventory and the company holidays of Independence Day, Founder's Day, and Labor Day.

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 2022, but not beyond? Here is an example of using the INTCK() function to count the number of PRODDAYS from December 22, 2022:

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

| Obs | startDate | endDate | result |
|---|---|---|---|
| 1 | Friday, December 23, 2022 | Saturday, December 24, 2022 | 0 |
| 2 | Friday, December 23, 2022 | Sunday, December 25, 2022 | 0 |
| 3 | Friday, December 23, 2022 | Monday, December 26, 2022 | 0 |
| 4 | Friday, December 23, 2022 | Tuesday, December 27, 2022 | 1 |
| 5 | Friday, December 23, 2022 | Wednesday, December 28, 2022 | 2 |
| 6 | Friday, December 23, 2022 | Thursday, December 29, 2022 | 3 |
| 7 | Friday, December 23, 2022 | Friday, December 30, 2022 | 3 |
| 8 | Friday, December 23, 2022 | Saturday, December 31, 2022 | 3 |
| 9 | Friday, December 23, 2022 | Sunday, January 1, 2023 | 3 |
| 10 | Friday, December 23, 2022 | Monday, January 2, 2023 | 3 |
| 11 | Friday, December 23, 2022 | Tuesday, January 3, 2023 | 3 |
| 12 | Friday, December 23, 2022 | Wednesday, January 4, 2023 | 3 |
| 13 | Friday, December 23, 2022 | Thursday, January 5, 2023 | 3 |
| 14 | Friday, December 23, 2022 | Friday, January 6, 2023 | 3 |
| 15 | Friday, December 23, 2022 | Saturday, January 7, 2023 | 3 |

Example 6 shows that the first three days after December 22 are not production days. This makes sense; the company is closed on Christmas Eve (holiday moved to Friday from Saturday), over the weekend, and on Christmas Day (Monday after Christmas off is a US holiday because Christmas fell on a Sunday.) The remainder of the following week counts as you would expect until Friday, because the company is closed on New Year's Eve (given Friday off because it falls on a Saturday), followed by January 2, 2023, another federal holiday. However, none of the following days are counted as production days. Shouldn't they be? Yes, but the dataset we used to create the interval only provides dates through December 31, 2022. Since those dates in 2023 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 left them out of the interval dataset, so they don't count in the PRODDAYS interval.

There are limitations with user-defined intervals: you cannot use a multiplier and/or 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 combination of SAS intervals, multipliers, and shift indexes can't get you what you need.

## WHAT ELSE CAN I DO WITH INTERVALS?

One of the easiest and most common uses for intervals is graphing. Intervals let you define your axes without having to use exact dates. If you need to recreate a graph later using updated data, you won't have to change the points on your axis, which makes your graph code reusable. You have most of the power of intervals for this at your disposal, including shift indexes and multipliers. The only part you don't have are custom intervals you've defined with datasets.

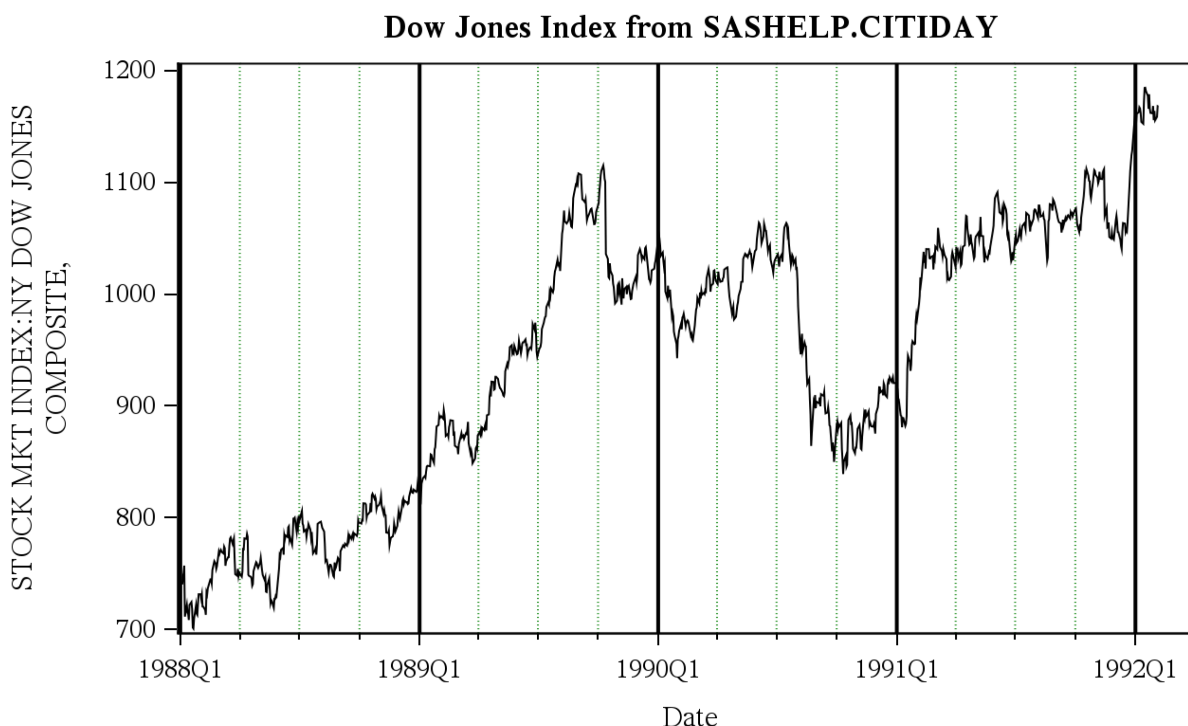
I’m going to use the SASHELP.CITIDAY dataset for this example so you can copy the code and play with it yourself. I’ll plot the Dow Jones composite stock market index using Code Fragment 1:

**Code Fragment 1: Plotting Dow Jones Composite and Foreign Exchange Rate over Time**

```
Title "Dow Jones Index from SASHELP.CITIDAY";
PROC SGPLOT DATA=sashelp.citiday;
SERIES X=date Y=snydjcm;
XAXIS TYPE=time VALUESFORMAT=yyq7. INTERVAL=year
GRID GRIDATTRS=(COLOR=black PATTERN=solid THICKNESS=2)
      MINOR MINORINTERVAL=quarter
      MINORGRID MINORGRIDATTRS=(COLOR=green PATTERN=dot THICKNESS=1);
RUN;
```

Note that the XASIS statement defines the axis as type TIME. For this to display well, you need to format your axis values using a date/time/datetime format. The INTERVAL option sets up your major tick marks. In this example, we’re using years. We’re also subdividing the years into quarters with the MINORINTERVAL statement. You don’t have to change the text for the points on the axis when you do it this way. The code in Code Fragment 1 produces Figure 3:

**Figure 3: Defining a Graph Axis Using Intervals**




SAS has created the labels and minor tick marks for you. If you had to run this code on data from 2022 to 2023, you could do it without modifying the code.

SAS also provides specific intervals geared towards the retail industry. Unfortunately, I don’t have any practical experience with that, so I can’t provide any examples. Nonetheless, I’ve listed these intervals and their definitions in Appendix 3.

## WHAT SHOULDN'T I USE INTERVALS FOR?

Age calculations or simple time and date durations. How many days are between October 31, 2023 and January 18, 2024? You could use the INTCK() function, but it's easier (and more efficient) to use simple subtraction as shown in Code Fragment 2:

**Code Fragment 2: How Many Days Between…?**

```
data a;
a = '18Jan2024'd - '31Oct2023'd;
put a= 'days to vacation';
run;

a=79 days to vacation
```

As for calculating ages, as long as you're calculating years, I'd strongly recommend using the YRDIF() function with the "AGE" argument. It's the most accurate method in SAS for calculating ages because it accounts for leap years between the two dates. While the difference may only show up in decimal places, you can also get an unexpected answer with the INTCK() function, as shown in Code Fragment 3:

**Code Fragment 3: Use YRDIF() Instead of INTCK()**

```
data b;
age1 = YRDIF('22dec1975'd, '31oct2023'd, 'AGE');
age2 = INTCK('YEARS', '22dec1975'd, '31oct2023'd, 'D');
age3 = INTCK('YEARS', '22dec1975'd, '31oct2023'd, 'C');
put age1= / age2= /age3 =;run;

age1=47.857534247
age2=48
age3=47
```

Once again, the difference between the DISCRETE (age2) and CONTINUOUS (age3) methods comes into play with INTCK(). The decimal portion of the YRDIF() result can be important to your analysis, such as in the case of survival analyses. If not, you can use the FLOOR() or CEIL() functions as appropriate to truncate the decimals. Regardless, there's no confusion over which INTCK() method you used for the calculation.

## SUMMARY

SAS intervals aren't as weird as they might seem at first glance. They are a powerful addition to your date and time toolbox. The "year in a day" concept can be confusing, but once you understand SAS is counting interval boundaries, it becomes much easier to use these tools appropriately. You can move the starting point of a SAS interval with shift points, and you can create longer intervals with multipliers. If you need to shift a multiplied interval, you can do that too.

The INTCK() and INTNX() functions continue to be the heart of interval usage. 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 method still has its uses, and you shouldn't adopt the new method for all situations. The SAMEDAY adjustment has been added to the INTNX() function and 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 in SAS 9 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.

SAS intervals can simplify defining the tick marks on axes within your graphs. You don't have to list actual values. Just specify an interval, an appropriate format, and SAS will do the rest.

SAS intervals don't have to be scary, they aren't mysterious, and they aren't that tricky to use. I've laid out the basics here and given you some examples to increase your understanding of how they work.

## REFERENCES

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

## RECOMMENDED RESOURCES

"Intro to SAS Intervals", https://www.youtube.com/watch?v=zRM70C0JJ4A, Dec 2020, SAS YouTube Channel

## ACKNOWLEDGMENTS

Thanks to SAS Technical Support in general for their assistance over the past 35 (!) years.

Thanks to the Florida Center for Instructional Technology Clipart ETC (Tampa, FL: University of South Florida, 2009) for the circle illustration.

## 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*days*W | Daily intervals with Friday-Saturday-Sunday counted as the same day (five-day work week with a Saturday-Sunday weekend). *days* identifies the individual numbers of the weekend day(s) by number (1=Sunday ... 7=Saturday). By default, *days*="17", so the default interval is WEEKDAY17W. | Each day |
| TENDAY | Ten-day intervals (a U.S. automobile industry convention) | 1st, 11th, and 21st of each month |
| SEMIMONTH | Half-month intervals | 1st and 16th of each month |
| MONTH | Monthly intervals | 1st of each month |
| QTR | Quarterly (three-month) intervals | 1-Jan<br>1-Apr<br>1-Jul<br>1-Oct |
| SEMIYEAR | Semi-annual (six-month) intervals | 1-Jan<br>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*days*W | Daily intervals with Friday-Saturday-Sunday counted as the same day (five-day work week with a Saturday-Sunday weekend.) *days* identifies the individual weekend days by number (1=Sunday ... 7=Saturday.) By default, *days*="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 | 1st, 11th, and 21st of each month |
| DTSEMIMONTH | Half-month intervals used with datetime values | 1st and 16th of each month |
| DTMONTH | Monthly intervals used with datetime values | 1st of each month |
| DTQTR | Quarterly (three-month) intervals used with datetime values | 1-Jan<br>1-Apr<br>1-Jul<br>1-Oct |
| DTSEMIYEAR | Semiannual (six-month) intervals used with datetime values | 1-Jan<br>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

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

/* Only need BEGIN variable, assume end-of-day as same day */
start = '01JAN2022'D;
stop  = '31DEC2022'D;
nweekdays = INTCK('WEEKDAY',start,stop);
DO i = 0 TO nweekdays;
   begin = INTNX('WEEKDAY',start,i); /* weekdays between start/stop */
   year = YEAR(begin);

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

/* Calculate days of company-specific holidays */
/* Use HOLIDAY() function to get dates for standard US holidays */

/* New Year’s Eve is the day before the first day of the FOLLOWING */
/* year – therefore, must add 1 to year value                       */
   nye = HOLIDAY('NEWYEAR',year+1) - 1;
   blkfri = HOLIDAY("THANKSGIVING",year) + 1; /* Thanksgiving Friday*/
   xmaseve = HOLIDAY('CHRISTMAS',year) - 1;  /* Christmas Eve */
   sprng = HOLIDAY("EASTER",year) - 2; /* Friday before Easter */

/* Founders Day - If on weekend, move forward/back as appropriate */
   founders = MDY(8,6,year);
   SELECT(WEEKDAY(founders)); /* Get day of week for Founders Day */
      WHEN(7) founders = founders - 1; /* Saturday */
      WHEN(1) founders = founders + 1; /* Sunday */
      OTHERWISE founders = founders;
   END;

/* Also need to adjust New Year’s Eve, and Christmas Eve if */
/* they fall on weekend days */
   SELECT(WEEKDAY(xmaseve)); /* Move Christmas Eve to preceding Fri */
      WHEN(7) xmaseve = xmaseve - 1;
      WHEN(1) xmaseve = xmaseve - 2;
      OTHERWISE xmaseve = xmaseve;
   END;

   SELECT(WEEKDAY(nye)); /* Move to preceding Fri */
      WHEN(7) nye = nye - 1;
      WHEN(1) nye = nye - 2;
      OTHERWISE nye = nye;
   END;
```

```
/* Exclude "normal" holidays, company-added holidays and inventory time */
/* from calendar data set */

   IF begin NE nye AND
      begin NE HOLIDAY('NEWYEAR',year) AND
      begin NE HOLIDAY("MLK",year) AND
      begin NE HOLIDAY("USPRESIDENTS",year) AND
      begin NE HOLIDAY("MEMORIAL",year) AND
      begin NE HOLIDAY("USINDEPENDENCE",year) AND
      begin NE HOLIDAY("LABOR",year) AND
      begin NE HOLIDAY("VETERANS",year) AND
      begin NE HOLIDAY("THANKSGIVING",year) AND
      begin NE HOLIDAY("CHRISTMAS",year) AND
      begin NE xmaseve AND
      begin NE blkfri AND
      begin NE sprng AND
      begin NE founders AND
      (begin < invstrt or begin > invend) THEN  /* Inventory time */
   OUTPUT;
END;

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