Paper BON-01

ISO 101: A SAS® Guide to International Dating

Xiaojin Qin, Covance Pharmaceutical R&D Co., Ltd., Beijing, China

Peter Eberhardt, Fernwood Consulting Group Inc, Toronto, Canada

ABSTRACT

For most new SAS programmers SAS dates can be confusing. Once some of this confusion is cleared the programmer may then come across the ISO date formats in SAS, and another level of confusion sets in. This paper will review SAS date, SAS datetime, and SAS time variables and some of the ways they can be managed. It will then turn to the SAS ISO date formats and show how to make your dates international.

INTRODUCTION

Landmarks. Dates are important landmarks in our lives – birthdays, anniversaries, Halloween. Dates and the measured passage of time have preoccupied people, probably since the dawn of civilizations; in ancient times those who understood the measured passage of time were often revered. Today we have $1.00 digital watches that provide milli-second accuracy yet people are still late for meetings; the reverence for the keepers of time may have diminished, but the need to keep our landmark dates accurate has not.

In this paper we will review the basic concepts of date, datetime, and time variables in SAS. In particular we will look at the basic difference between the data representation and the visual representation of these variables. From there we will look at SAS functions that help us manipulate dates; some functions are used to create dates, some to extract parts of dates, and some to do date arithmetic. After this review we will look at two features that allow for the internationalization of dates: LOCALE and ISO 8601 dates.

Throughout the paper we will usually refer to SAS date variables; in our experience date variables are far more common and the need to manipulate them is subsequently more common. When the discussion turns to ISO dates we will look more into times and durations.

19,477

19,477 is a landmark; it represents 29Apr2013. It also represents 2013-04-29 and 04/29/13. It is probably the birthday of someone reading this paper and also someone's wedding anniversary. With certainty we can say it is not Halloween. 19,477 could also be the price of a car.

If we are to accept that 19,477 represents 29Apr2013, we need to know something about it. From a lower level perspective it is an integer representing the number of days from a reference point; in SAS the reference point is 01Jan1960. The value 19477 tells us that 29Apr2013 is 19,477 days from 01Jan1960; in SAS, an integer with the number of days since 01Jan1960 is the ***data representation*** of a date. Having dates represent the number of days since a reference point has two immediate advantages. The initial advantage is that it enables SAS to store the value in a small and easily manipulated data type – an integer. Although in today's computers where storage is plentiful and inexpensive this may not seem valuable, in the initial days of SAS implementations this was not the case. The second advantage can be seen from a programmer's perspective - it makes date arithmetic simple. To get tomorrow's date we simply add 1 to today's date; to find the number of days between two dates we simply subtract one from the other.

```
10   data dates;
11        today = "29apr2013"d;        /* assign a date constant */
12        tomorrow = today + 1;        /* add 1 to the date      */
13        nextWeek = "06may2013"d;     /* assign a date constant */
14        elapsed = nextWeek - today;  /* take the difference    */
15        put today= tomorrow= nextweek= elapsed=;
16   run;

today=19477 tomorrow=19478 nextWeek=19484 elapsed=7
```

We probably agree that a date representing the number of days from a reference date makes sense. We also probably agree that numbers like 19477 and 19478 are essentially of little usable value in our daily lives, even to many of us hard core SAS geeks. This leads us to the part that often confuses new SAS programmers – the ***visual representation*** of a date.

Let's look at the previous SAS code, but this time we include some visual representations:

```
17   data dates;
18        today = "29apr2013"d;
19        tomorrow = today + 1;
20        nextWeek = "06may2013"d;
21        elapsed = nextWeek - today;
22        put today=      today=    yymmdd10. today=    worddate23. /
23            tomorrow=   tomorrow= yymmdd10. tomorrow= worddate23. /
24            nextweek=   nextweek= yymmdd10. nextweek= worddate23. /
25            elapsed=;
26   run;

today=19477 today=2013-04-29 today=April 29, 2013
tomorrow=19478 tomorrow=2013-04-30 tomorrow=April 30, 2013
nextWeek=19484 nextWeek=2013-05-06 nextWeek=May 6, 2013
elapsed=7
```

From this example we can see the data representation of ***today*** is 19477, however we can have multiple visual representations; in this case we show two – ***2013-04-29***, and ***April 29, 2013***. Put another way, no matter how we display the date, the underlying value does not change. More importantly we do not need to have a separate variable to display the value differently. Applying a SAS ***format*** to the variable allows us to easily change the visual representation. In the example above we used the formats ***yymmdd10.*** (to display 2013-04-29) and ***worddate23.*** (to display April 29, 2013) to display each variable with more than one visual representation. When we associate a date format with the number we now can say that 19,477 is not the price of a car.

2013/04/29

We have seen how we can start with a date value and display it is different ways; that is, we take the data representation and convert it to visual representations through the use of formats. What if we want to be introduced to a good looking date? That is, how do we convert visual representations into data representations?

To convert from visual representations of dates, the representation we usually get when we read a raw input file, we use SAS ***informats***. Whereas a SAS format converts an underlying data representation to a visual representation, an informat converts a visual representation into an underlying data representation. The following example reads two variables with different visual representations and converts them to SAS date values; we can see that SAS converts both 30apr2013 and 2013-04-30 to 19478:

```
data _null_;
     input toDay          yymmdd10.  +1
           tomorrow          date9.  +1;
     put toDay= tomorrow=;
datalines;
2013-04-29 30apr2013
2013-04-30 01may2013
;;
run;
toDay=19477 tomorrow=19478
toDay=19478 tomorrow=19479
```

As we now know, we can display the date values any way that is appropriate for our application; we are not restricted to displaying the dates in the format in which we read the dates.

Converting the input representation to a date value makes sense. "***BUT*** the person who created my dataset read the dates as character strings, not as dates – and I need to do some manipulations on the dates". This is a common problem. Sometimes the data were read this way because the person only wanted to display the variable and never

expected to use it in any calculations. No matter the reason why the data were created this way, we can remedy the situation without having to go back to the raw data.

In the previous example we saw the use of the ***INPUT*** statement coupled with informats to create date variables. We can perform a similar action using the ***input()*** function, as demonstrated in the next example.

```
37   data baddates;
38        charDate = "2013-04-29";
39        put charDate=;
40        numDate = input(charDate, yymmdd10.);
41        put numDate= numDate= yymmdd10. numDate= date9.;
42   run;

charDate=2013-04-29
numDate=19477 numDate=2013-04-29 numDate=29APR2013
```

The ***input()*** function takes a character variable, in our example **charDate**, and converts it to a number using a specified format, in our case ***yymmdd10.***. Of course the format you use must match the format of the character variable. You will note that we now have two variables – one a character representation of the date, and the other a numeric with the underlying SAS representation of the date. If you need to keep the same variable name you will need to do some renaming of the variables. One way to do this is:

```
49   data fixedDates;
50        set baddates (rename=(dateVar=c_dateVar));
51        drop c_dateVar;
52        dateVar = input(c_dateVar, yymmdd10.);
53   run;
```

1,682,856,000

The value ***1,682,856,000*** is the ***data representation*** of 29Apr2013 at 12:00 pm (noon). Specifically it is an integer representing the number of seconds from a reference point; in SAS the reference point is 01Jan1960 at midnight. As we saw above with dates, SAS has a way to assign values that contain both a date and time component:

```
53   data datetimes;
54        toDayAtNoon = "29apr2013:12:00:00"dt; /* assign a datetime constant */
55        put toDayAtNoon=;
56   run;

toDayAtNoon=1682856000
```

As with the date constant we originally saw, there is a specific formula for a ***datetime*** constant. In this case we extend the day part to include the hour, minute, and second (separated by the colon); instead of the ***d*** modifier we have the ***dt*** (for datetime) modifier.

Where date values represent the number of days since 01Jan1960, a datetime variable represents the number of seconds since midnight of 01Jan1960 (or “01jan1960:00:00:00”dt). We could convert the datetime variable to get the SAS date by dividing by 86400 (24 hrs * 60 min * 60 sec) and dropping the fraction – or we could be smart and use the SAS function ***datepart().*** The ***datepart()*** function takes a SAS datetime variable and returns the SAS date. This function is useful when you are reading dates from a relational database that stores its dates in the equivalent of SAS datetime variables. Code to do this could look like:

```
PROC SQL;
connect to ODBC dsn=sqlSrv;
select ..., datepart(datevar) as datevar, ...
from connection to odbc
(
  Select ..., datevar, ...
).
```

As with SAS date variables you can use formats to change the visual representation to something more meaningful; unfortunately there is not the richness of built-in formats for displaying datetime variables

```
57   data datetimes;
58        toDayAtNoon = "29apr2013:12:00:00"dt;
59        put toDayAtNoon= toDayAtNoon= datetime23.;
60   run;

toDayAtNoon=1682856000 toDayAtNoon=29APR2013:12:00:00
```

We have looked at SAS date and SAS datetime variables. Now let's look at SAS time variables

45,000

The value ***45,000*** is the ***data representation*** of 12:00 pm (noon).

```
61   data times;
62        twelveThirty = "12:30:00"t;
63        put twelveThirty=;
64        twelveThirtyPlus = "12:30:00.5"t;
65        put twelveThirtyPlus=;
66   run;

twelveThirty=45000
twelveThirtyPlus=45000.5
```

Once again we see a formula for a time constant; this time it is a string that looks like "HH:MM:SS" with a ***t*** modifier to specify this is a time value. Note also that we can represent fractions of a second; although it was not demonstrated, datetime variables can also have fractions of seconds, the underlying data representation of a time variable (for example 45000) is the number of seconds since midnight.

Of course we can also change the visual representation to something more meaningful:

```
67   data times;
68        twelveThirty = "12:30:00"t;
69        put twelveThirty= twelveThirty= time. twelveThirty= timeampm.;
70        twelveThirtyPlus = "12:30:00.5"t;
71        put twelveThirtyPlus=
              twelveThirtyPlus= time.   twelveThirtyPlus= timeampm14.1;
72   run;

twelveThirty=45000 twelveThirty=12:30:00 twelveThirty=12:30:00 PM
twelveThirtyPlus=45000.5 twelveThirtyPlus=12:30:01 twelveThirtyPlus=12:30:00.5 PM
```

The examples above focused on SAS date, datetime, and time variables and how they are represented. We saw that each represent a number of intervals from some reference point.

- For SAS dates the interval was a day and the reference point was 01Jan1960
- For SAS datetimes the interval was a second and the reference point was 01Jan1960 at midnight
- For SAS times the interval was a second and the reference point was midnight of the current day

From here we will look at some useful SAS functions that work with SAS date, datetime, and time variables.

FUNCTIONS

SAS has a number of date, datetime, and time functions. We are going split them into two arbitrary groups – one we will call informational and one we will call computational. We will look at these groups next.

INFORMATIONAL FUNCTIONS

The functions we call informational are of two types: functions that return the current date or time, and functions that return information on date, datetime, and time variables. Let's look at the former first.

A common need is to capture the current date and/or time as part of your processing – for example you may need to determine how many days overdue an account is, or you may need to put the current date or time in a title of footer. If you need the current date SAS provides two functions: ***today()*** and ***date()***. The following example shows the use of the ***today()*** function in a DATA step to determine how many days old an account is; note that the ***date()*** function could have been used instead.

```
data getAccountAge;
     set currentLedger;
     if _n_ = 1 then billingDay = today(); /* could have used date() */
     * get the number of days since invoiced ;
     daysOld = billingDay - invoiceDate;
     * create a category based on how old the account is;
     select;
        when (daysOld < 31)  category = 'Current';
        when (daysOld < 61)  category = 'Over 30';
        when (daysOld < 91)  category = 'Over 60';
        otherwise            category = 'Over 90';
     end;
run;
```

Using a function to get the current date as part of DATA step processing makes sense, however for use in titles and footnote we do not need the DATA step. Let's see how we can use functions to add the current date and current time to a title in a report.

Since we want to use the date and the time as part of a title we need to get both the date and the time and convert them to macro variables. SAS not only provides the functions to return the current value of the date (date()) and time (time()), but also a means to easily create a formatted macro variable from the function. This example will create two macro variables, runDate with the current date and runtime with the current time; both variable will be formatted with an appropriate visual representation (e.g. 2013/04/29 and 9:30AM). For the time variable we used a format width that was too narrow to display the seconds as part of the time; as a report title time with minute accuracy was adequate for our needs.

```
%let runDate = %sysfunc(today(),yymmdd10.);
%let runTime = %sysfunc(time(),timeAMPM8.);
title " Xiao Jin's Report ... Processed on &rundate Starting at &runTime";
```

In this example we used the macro function ***%SYSFUNC()*** to call the DATA step functions. Not only can SYSFUNC call a DATA step function, but also it can apply a format to the result. In one call we both captured the current date and we formatted the current date. There is also an automatic SAS macro variable ***&sysdate*** that has the date; this macro variable has the date the current SAS session started, not the current date.

Although ***date()*** and ***time()*** are valuable functions, they are not the only informational functions. Unlike ***date()*** and ***time()***, the other functions act on other variables to give us information about dates or times; some of these functions take dates apart to give a component part while others take components and make dates.

One common problem we run into is dealing with dates from a relational database, for example SQL Server. Many relational databases store their dates in the equivalent of SAS datetime variables; as we have seen SAS dates and SAS datetimes have very different underlying representations. Many times we have extracted data from SQL Server and then tried matching a SQL date variable on a SAS date variable. Many times have we subsequently wondered why we had no matches after the merge – at least until we remember the two variables cannot be compared directly. This is where the ***datepart()*** function comes in; the ***datepart()*** function returns the date part of a datetime variable. As you will probably guess there is a ***timepart()*** function to return the time part of a datetime variable. Here are these functions in action:

```
data fromSQLServer;
  set sql.dataTable;
    sasDate = datepart(sqlDate); /* get the date */
```

```
    sasTime = timepart(sqlDate); /* get the time */
    drop sqlDate;
  format sasDate yymmmdd10.;
  format sasTime time8.;
run;
```

In this example we see the SQL date variable split into two SAS variables – one with the date and one with the time. Of course if your SAS datasets use datetime variables you may not have to split the SQL date into two.

Some other functions that provide the component parts of a date variable are:

- day(dateVar)
  Returns the day (1 – 31) of a SAS date variable
- month(dateVar)
  Returns the month (1 – 12) of a SAS date variable
- year(dateVar)
  Returns the year of a SAS date variable
- qtr(dateVar)
  Returns the quarter (1 – 4) of a SAS date variable

If you can take a date apart you should be able to put it back again. SAS provides functions to do this:

- mdy(mm, dd, yy)
  Returns a SAS date from the three integer variables for month (mm), day (dd), and year (yy)
- hms(hh, mm, ss)
  Returns a SAS time variable for the three variables hour (hh), minute (mm), and second (ss). The seconds value can be floating point
- dhms(date, hh, mm, ss)
  Returns a SAS date time variable from the four variables for date, hour (hh), minute (mm), and second (ss)

This has been a quick overview of some of the informational functions. We saw the two functions that tell us the date and time – aptly called ***date()*** and ***time()***. We also saw functions that gave us the component parts of dates. Finally we saw some functions that start with component parts and give us date variables. We will now turn to two powerful computational functions.

COMPUTATIONAL FUNCTIONS

In this section we are going to focus exclusively on SAS dates; late we will return to datetime variables.

Boundaries and Intervals

When dealing with SAS dates we commonly need to know two things:

1. How many intervals are there between two dates,
2. What is the date x intervals away.

Before we can proceed we need to have a common understanding of intervals, and of its close companion boundaries.

We have already come across intervals when we introduced date variables – we said a SAS date variable represented the number of intervals, where the interval was a day, from the reference point of 01Jan1960. Since the interval is a day, we naturally think of the boundary between days as mid-night – at mid-night we cross the boundary of one day (say Tuesday) to the next (Wednesday). Of course there are many other intervals with which we need to deal.

If all we needed to worry about when looking at two dates was the number of days between them then we would have no need for any computational functions – to find the interval between two dates we need only subtract one from the

other. To find a date some interval (say 10 days) away from today we need only add 10 to the date. Luckily the world is not so simple so we have gainful employment getting new dates.

Some of the other common boundaries with which we have to work are:

- Week
- Month
- Quarter
- Year

That is we need to know how many weeks are between two dates, or you may be asked to compute a date 3 months from now. In order to answer these questions we need to understand what the boundary, or starting point, for each of these is:

Interval	Boundary (starting point)
Week	Sunday
Month	first day of month
Quarter	Jan 1, Apr 1, Jul 1, Oct 1
Year	Jan 1

This means

- Every time we cross into a Sunday we start a new week
- Every time we cross into the first day of a month we start a new month
- Every time we cross Jan 1/ Apr 1/ Jul 1/ Oct 1 we start a new quarter
- Every time we cross Jan 1 we start a new year

With this understanding of intervals and boundaries (or starting points) in mind we will look at two functions for manipulating SAS dates – ***intnx()*** and ***intck()***

- ***intck('interval', start, end)***
  Returns the count of the number of interval boundaries between two dates, two times, or two datetime values
  Start can be a date, time, or datetime variable
  End must be the same type as start
- ***intnx('interval', start, increment <, 'align'>)***
  Increments a date, time, or datetime value by a given time interval, and returns a date, time, or datetime value.
  Start can be a date, time, or datetime variable
  Increment: is number of intervals to add; can be negative
  'align': controls position of the resultant date within the interval. Optional.
  - 'beginning' ('b') This is the default
  - 'middle' ('m')
  - 'end' ('e')
  - 'same' ('s')

When you need to know how many intervals are between two dates you use ***intck()***. When you need to create a date that is some number of intervals away from another date you use ***intnx()***. Some examples will show how these functions work, and how you need to understand boundaries (or starting points) to be sure you get the result you expect – or understand the result you got.

```
73   data _null_;
74       start = '01jan2013'd;
75       end = '01jul2013'd;
76       days1 = end - start;
```

```
77       days2 = intck('day', start, end);
78       weeks = intck('WEEK', start, end);
79       months = intck('month', start, end);
80       year = intck('year', start, end);
81       put days1= days2= weeks= months= year=;
82   run;

days1=181 days2=181 weeks=26 months=6 year=0
```

In this example we are looking at two dates – 01Jan2013 and 01Jul2013. Since we are using ***intck()*** we will be looking at a count of intervals between the two; in particular we will look at days, weeks, months, and years. When we look at the log we see that taking the difference between the two dates (line 76) and using the ***intck()*** function use a day interval, we end up with the same result: there are 181 days between the two dates. When we look at the interval count for weeks (26), months (6) , and years (0) we get results we would expect. After we make a small change, set the end date to 30Jun2013 instead of 1Jul2013 (one day earlier) we get the following:

```
83   data _null_;
84       start = '01jan2013'd;
85       end = '30jun2013'd;
86       days1 = end - start;
87       days2 = intck('day', start, end);
88       weeks = intck('WEEK', start, end);
89       months = intck('month', start, end);
90       year = intck('year', start, end);
91       put days1= days2= weeks= months= year=;
92   run;

days1=180 days2=180 weeks=26 months=5 year=0
```

Here we see the number of days difference is one less (180), which makes sense, The dates are still 26 weeks apart, but they are now five months apart even though we changed the end date by only one day. The one day change meant the end date did not cross the 'first of month' boundary hence the dates are one month less apart.

When we change the dates from 1Jan2013 to 31Dec2013 we get:

```
93   data _null_;
94        start = '01jan2013'd;
95        end = '31dec2013'd;
96        days1 = end - start;
97        days2 = intck('day', start, end);
98        weeks = intck('WEEK', start, end);
99        months = intck('month', start, end);
100       year = intck('year', start, end);
101       put days1= days2= weeks= months= year=;
102  run;

days1=364 days2=364 weeks=52 months=11 year=0
```

Although we would think a year has gone by, and ***intck()*** sees 52 weeks, which we think of as one year, only 11 months (11 times we crossed into a new month) and still zero years (we have yet to pass a Jan 1 date) have gone by.

One more example

```
103  data _null_;
104       start = '31dec2013'd;
105       end = '01jan2014'd;
106       days1 = end - start;
107       days2 = intck('day', start, end);
108       weeks = intck('WEEK', start, end);
109       months = intck('month', start, end);
110       year = intck('year', start, end);
```

```
111       put days1= days2= weeks= months= year=;
112  run;

days1=1 days2=1 weeks=0 months=1 year=1
```

Now we go from 31Dec2013 to 2014-01-01, a change of one day as seen by the two day interval results. We also have a difference of one month (we crossed from one month into the next), and a difference of one year (we crossed from one year into the next).

From these simple examples it is easy to see how ***intck()*** calculates intervals; it is also easy to see how you have to be careful that you understand how intervals and boundaries work or your results may not be what you expect.

With a better understanding of intervals and boundaries let's look at the ***intnx()*** function and how we can get a date that is some number of intervals away.

```
113  data _null_;
114       start = '01jan2013'd;
115       week  = intnx('week', start, 1);
116       weeksE = intnx('week', start, 1, 'e');
117       weeksS = intnx('week', start, 1, 's');
118       weeksM = intnx('week', start, 1, 'M');
119       put week= yymmdd10. weeksE= yymmdd10. weeksS= yymmdd10. weeksM=
yymmdd10.;
120  run;

week=2013-01-06 weeksE=2013-01-12 weeksS=2013-01-08 weeksM=2013-01-09
```

In this example we are starting with 01Jan2013 and adding one week to it – four different ways. The first way (line 115) uses the default alignment of beginning; this means we will get a date at the beginning of the next week. Since the Sunday is the first day (beginning) of a week the call on line 115 should return Sunday Jan 3, 2010 – which it does.

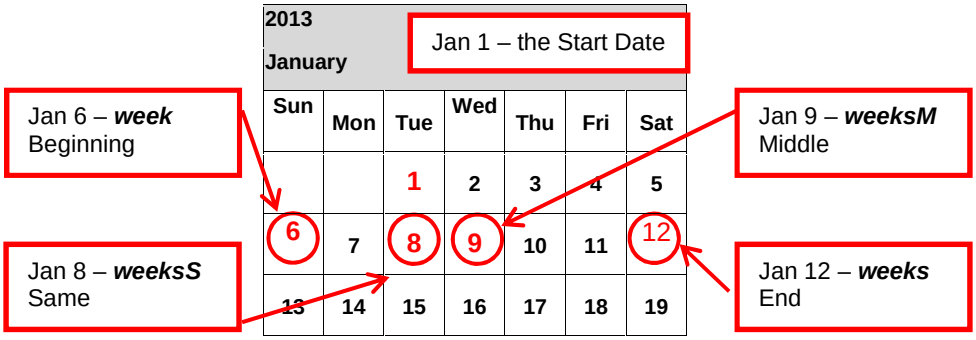


Figure 1: Add One Week

The 'end of week' alignment (weeksE) results in Sat Jan 12, the ' same day of week' alignment (weeksS) results in Tue Jan 8, and the 'middle of week' alignment (weeksM) results in Wed Jan 9. This example shows us we can use ***intnx()*** to get a variety of different dates that are all "1 week" away. There are two common dates we need that can be easily calculated with ***intnx()***.

Two dates we commonly need to determine are the first and the last day of a given month. After looking at the previous example we can guess that these are easily computed. To get a date within the current month the increment should be 0 (zero) – we are adding zero months to the current month. To get the first day of a month, the alignment is 'beginning' and to get the last day of the month the alignment is 'end'

```
121  data _null_;
122       start = '11jan2013'd;
123       monthStart = intnx('month', start, 0, 'b');
124       monthEnd   = intnx('month', start, 0, 'e');
125       put monthStart= yymmdd10. monthEnd= yymmdd10.;
126  run;

monthStart=2013-01-01 monthEnd=2013-01-31
```

As you look at these examples you may be asking yourself "If today is Jan 30 and I want the **same day** one month later, what date do I get?". As many school text books would say, this is left for an exercise.

The interval parameter of the ***intnx()*** function can get interesting; in this paper we will examine some options, that will demonstrate how useful it can be. Earlier we noted the call to intnx() looked like:

- ***intnx('interval', start, increment <, 'align'>)***

The interval argument can be more complex than this:

- ***intnx('interval'<multiple><.shift>, start, increment <, 'align'>)***

  multiple indicates the optional multiplier for the interval. For example YEAR2, YEAR is the interval and 2 the multiplier; this means each interval is 2 years.

  Shift the optional shift of the starting boundary. For example YEAR.10, YEAR is the interval and 10 (October 1) is the start of the year. YEAR4.11 has a 4 year interval and starts November 1, much like the U.S. election cycle

Fiscal years rarely align with calendar years; as SAS programmers we regularly have to align calendar dates with the fiscal year – ***intnx()*** can help.

The alignment parameter can take on compound values so that the boundary (or start period) and what one interval is can be changed. If we consider the interval ***year***

- Boundary (or start period) is Jan 1
- One interval is one year

If our fiscal year starts on Oct 1 then we could look at years that have a boundary (or start period) of Oct 1 rather than Jan 1; that is, every time we cross Oct 1 we cross into a new year. If we want to tell ***intnx()*** that a year begins in October we have to alter the interval as

- 'year.10'

If we look back at the example of finding the first day of the current month we saw we had to specify an increment of zero (add zero months to the current month), an alignment of 'b' (beginning), and an interval of 'month'. By extension we can see that to get the beginning of the current year we only have to change the interval from 'month' to 'year'. If a year begins in October rather than January, then we know out interval is 'year.10'. Putting this together we can see from the following example we can take any date and get the start of the fiscal year:

```
127  data _null_;
128       start = '11JUN2013'd;
129       yearStart = intnx('year', start, 0, 'b');
130       fiscalStartApr= intnx('year.4',  start, 1, 'b');
131       fiscalStartOct= intnx('year.10', start, 1, 'b');
132       fiscalEndApr   = intnx('year.4',  start, 1, 'e');
133       fiscalEndOct   = intnx('year.10', start, 1, 'e');
134       put yearStart= yymmdd10. /
135       fiscalStartApr= yymmdd10. fiscalEndApr= yymmdd10. /
136       fiscalStartOct= yymmdd10. fiscalEndOct= yymmdd10.;
137  run;

yearStart=2013-01-01
fiscalStartApr=2014-04-01 fiscalEndApr=2015-03-31
fiscalStartOct=2013-10-01 fiscalEndOct=2014-09-30
```

From this example we see that starting from a date of June 11 we can easily determine the beginning of the year.

- The beginning can be the calendar start (Jan 1) as we see in the variable ***yearStart***.
- The beginning of the year can be Apr 1 as can be seen in the variable ***fiscalStartApr***
- The beginning of the year can be Oct 1 as can be seen in the variable ***fiscalStartOct***

Needless to say there is far more you can do with the interval; we have only attempted to show that with a little scratching the surface can be greatly enlarged. For more examples of manipulating SAS dates see Tabachneck et al, 2012

LOCALE FOR NATIONAL LANGUAGE SUPPORT

Peter works in Toronto; he generates his data and reports and then passes them on to Xiao Jin in Beijing. There are numerous dates in Peter's work that he would like to convert to make it easier for Xiao Jin to use and pass on to her clients. Thinking himself very clever, Peter undertakes an project to create several macros to convert the dates to a format Xiao Jin's clients will like. After several unsuccessful attempts, Xiao Jin tells Peter his efforts are not necessary, she has the solution: use LOCALE. Being able to switch the language of reports is ideal to international data reporters. Let's see how this can be achieved by specifying the value of LOCALE=.

A locale reflects the language, local conventions such as data formatting, and culture for a geographical region. Local conventions might include specific formatting rules for dates, times, and numbers and a currency symbol for the country or region.

Dates have many representations, depending on the conventions that are accepted in a culture. The month might be represented as a number or as a name. The name might be fully spelled or abbreviated. The order of the month, day, and year might differ according to locale.

We can use the LOCALE= system option to specify the locale of the SAS session at SAS invocation.

Sample scenarios

Below are some sample scenarios to illustrate the process.

Scenario 1: Create a report using the SAS Default Settings for LOCALE =.

```
options locale=English_UnitedStates nonumber nodate;
data nldt;
    dt="29APR2013:16:30"dt;
    dt_nl=put(dt,nldatm.);
    format dt datetime20.;
run;

proc print data=nldt;
title "Date representation using the SAS Default Settings for LOCALE=";
run;
```

This is what the result looks like:

Scenario 2: Create a report in Chinese format by specifying LOCALE = Chinese_China.

```
options locale=Chinese_China nonumber nodate;
data nldt;
    dt="29APR2013:16:30"dt;
    dt_nl=put(dt,nldatm.);
    format dt datetime20.;
run;

proc print data=nldt;
title "更改 LOCALE=Chinese_China 设置将时间显示为中文格式";
run;
```

This is what the result looks like:

We have shown some simple examples of LOCALE for dates, but it can also be used for other formatting.

ISO 8601

Now that we have come to embrace SAS dates, datetimes, and times as numbers it is time to rethink some of our dating habits.

WHAT IS ISO 8601

The International Organization for Standardization (ISO) has released a standard ISO 8601:2004(E) that defines an internationally accepted way to represent dates and times in an unambiguous manner. For an organization that deals

only with internal data and only within one time zone, this may not be important (although we will see examples where it can be important even in this limited scope); however, for organizations that exchange data with other organizations in different countries and different time zones the value of the standard becomes apparent. Let's look at a simple illustration.

In Toronto on Saturday April 27 I set my watch to a published research time signal at 6:00 pm; as soon as I set the watch my plane takes off for San Francisco and arrives on time after the scheduled five hour flight. Once I disembark and collect my luggage another hour has passed; it is now 9:00 pm. I stop at the ATM to get some cash from my bank account. What day will be on the transaction recorded by my bank? Will it report the date from the ATM (Saturday) or the date in the processing centre in Toronto (Sunday)? If the transaction is sent as plain text in the accepted US format 04292013, will it be rejected by a system where the accepted date format would be 29/04/2013? Is my watch, which now reads 12:00 AM Monday, showing the wrong time?

Obviously this is a contrived scenario, but it does highlight some important issues that become more and more critical as our information exchanges cross time zones and cultural zones.

YYYY-MM-DDTHH:MM:SS±HH:MM

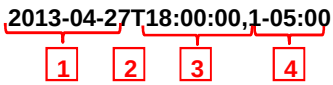
This is the ISO representation for a complete and unambiguous point in time. There are three components of this representation

The date: YYYY-MM-DD

The time: T followed by hh:mm:ss

The time zone: ± (a plus or minus) followed by the hour/minutes difference of local time from Coordinated Universal Time (UTC).

If I had an ISO 8601 compliant watch, when I set it in Toronto it would have read 2013-04-27T18:00:00-05:00, although the correct reading would be closer to 2013-04-27T18:00:00,1-05:00 since I would have a short reaction delay between hearing the time signal and resetting my watch.



Let's look at the components:

1. 2013-04-27: the 'date'.
    - A four digit year
    - Leading 0 in month (and day if less than 10)
    - Hyphen separator.
2. T: indicates a time will follow
3. 18:00:00,1
    - A 24 hour clock is used
    - Minutes and seconds both expanded with 0 (zero) to fill the positions allocates
    - One tenth of a second; the comma is the delimiter
4. -05:00: time zone offset
    - Toronto is five hours BEHIND UTC (-05:00)
    - Hours and minutes are zero-filled.

In this complete representation there is one simplification that can be made; instead of the ±hh:mm indicating the offset from UTC, the time can be normalized to UTC and a ***Z*** placed at the end. The following two examples represent the same point in time:

- **2013-04-27T18:00:00-05:00 (6:00 pm EDT)**
- **2013-04-27T23:00:00Z (11:00 pm UTC)**

The examples above are known as the ***Extended Format***; the extended format includes the separators. There is also a ***Basic Format*** that omits the separators. In basic format my watch would look like:

- 20130427T180000,1-0600
- 20130427T180000,1Z

When you see the basic format you can see the need for zero filling. For ease of reading, all the examples will use the extended format; an equivalent basic format is available. When we use the extended format the separators identified in the standard are:

- **-** hyphen, to separate the date elements (year, month, day, week)
- : colon, to separate the time elements (hour, minute, second)

To indicate a decimal fraction, a comma (,) is the preferred separator, although the period (.) can be used as long as all parties involved in the data exchange agree.

If our input data has a complete ISO 8601 representation, how do we read it? We could read in the three main components (date, time, time zone offset) separately and create a SAS datetime variable, or we could use the built in ISO informats as in this code:

```
data iso;
     input @1 isodate e8601dz27.1 ;
     put isodate= isodate= datetime27.1 isodate=e8601dz27.1;
     datalines;
2013-04-27T18:00:00,1-05:00
2013-04-27T23:00:00,1Z
2013-04-27T00:00:00.1Z
;;
run;
```

The SAS log:

```
isodate=1682722800.1 isodate=27APR2013:23:00:00.1
    isodate=2013-04-27T23:00:00.1+00:00
isodate=1682722800.1 isodate=27APR2013:23:00:00.1
    isodate=2013-04-27T23:00:00.1+00:00
isodate=1682640000.1 isodate=27APR2013:00:00:00.1
    isodate=2013-04-27T00:00:00.1+00:00
```

From this example we see we note:

- SAS provides a built-in INFORMAT (e8601dz27.1) to read ISO compliant dates. As we will see later, there are several INFORMATS available.
- Once read in, the variable is converted to a SAS datetime variable. This means it can be managed as any SAS datetime variable is managed. In this example we only display the variable,
    1. In its data representation
    2. Using a built-in SAS datetime format (datetime27.1)
    3. Using a built-in SAS ISO 8601 format (e8601dz17.1)
- Converting to a SAS datetime format causes some information to be lost. Our input data had a time (18:00:00,1) and a UTC offset (-05:00). On display, we can see we lost the UTC offset. The value saved (2013-04-27T23:00:00.1+00:00) represents the same point in time as the value we read in (2013-04-27T18:00:00,1-05:00), but we no longer know the original UTC offset. If this is important (the UTC offset) to your analysis, you will have to explicitly read and save it.

ISO 8601 OVERVIEW

Before we look more into some of the SAS FORMATS/INFORMATS and SAS call routine IS8601_CONVERT(), let's look briefly at the ISO 8601 standard, formally known as "*ISO 8601, Data elements and interchange formats — Information interchange — Representation of dates and times*". This standard is in its third edition (2004-12-01) and will often be referred to as ISO8601:2004. The standards document [ISO 2004] is 33 pages; this section will be a brief summary. The following excerpt [1] from the document outlines its scope:

> ***1 Scope***
>
> *This International Standard is applicable whenever representation of dates in the Gregorian calendar, times in the 24-hour timekeeping system, time intervals and recurring time intervals or of the formats of these representations are included in information interchange. It includes*
>
> - *calendar dates expressed in terms of calendar year, calendar month and calendar day of the month;*
> - *ordinal dates expressed in terms of calendar year and calendar day of the year;*
> - *week dates expressed in terms of calendar year, calendar week number and calendar day of the week;*
> - *local time based upon the 24-hour timekeeping system;*
> - *Coordinated Universal Time of day;*
> - *local time and the difference from Coordinated Universal Time;*
> - *combination of date and time of day;*
> - *time intervals;*
> - *recurring time intervals.*
>
> *This International Standard does not cover dates and times where words are used in the representation and dates and times where characters are not used in the representation.*

Order

We saw an example above of an ISO representation for a complete and unambiguous point in time: 2013-04-27T18:00:00-05:00. From this we see a general rule that is followed throughout the standard:

- Date and time values are organized in a "biggest to smallest" order: year, month, day, hour, minute, second, and fraction of second.

Listed below are elements of dates and times in the standard. In these examples we show the ***extended*** format; keep in mind that there is an equivalent ***basic*** format. When dealing with the SAS informats/formats we also only show the extended informat/format. To use the basic SAS informat/format replace the leading ***E*** with a ***B***; for example ***E***8601DA becomes ***B***8601DA. All of these examples are use 2013-04-29.

Calendar date

YYYY-MM-DD

where YYYY is the year, MM is the month of the year between 01 (January) and 12 (December), and DD is the day of the month between 01 and 31.

Example: 2013-04-29 represents the twenty-ninth day of April in 2003.

The SAS informat/format ***E8601DAw.*** is used to read/write ISO 8601 calendar dates and return a SAS ***date*** variable. This is equivalent to the YYMMDD10. informat/format.

The SAS informat/format ***E8601DNw.*** is used to read/write ISO 8601 calendar dates and return a SAS ***datetime*** variable where the time would be set to 00:00:00.

```
sasDate= input('2013-04-29', E8601DA.);
put sasDate E8601DA.;   /* log: 2013-04-29 */
```

Week date

YYYY-Www-D

where YYYY is the year, W is the indicator that this is a “week” representation, ww is the week of the year between 01 (the first week) and 52 or 53 (the last week), and D is the day in the week between 1 (Monday) and 7 (Sunday).

Example: 2013-W17-6 represents the sixth day of the seventeenth week of 2013.

The SAS informat/format ***WEEKw.*** is used to read/write ISO 8601 week dates. While ISO 8601 states that a calendar week variable would have 10 characters, the SAS format WEEKV can have a width as small as 3 making it not ISO 8601 compliant.

```
sasDate= input('2013-04-29', E8601DA.);
put sasDate E8601DA. ' ' sasDate WEEKV.;   /* log: 2013-04-29  2013-W18-01*/
```

In ISO 8601 , weeks start on Monday. The first week of the year is the week that contains the first Thursday of the year. Put another way, if January 1 falls on Monday, Tuesday, Wednesday or Thursday it is in week 01. However, if January 1 falls on a Friday, Saturday or Sunday it falls into the last week of the previous year. In 2013, January 1 fell on Tuesday so it was week 01 day 2 of 2013. In 2012, January 1 fell on Sunday, so it was week 53 day 7 of 2011.

Time of the day is the time representation, using the 24-hour timekeeping system. :

hh:mm:ss

where hh is the number of complete hours that have passed since midnight, mm is the number of complete minutes since the start of the hour, and ss is the number of complete seconds since the start of the minute.

Example: 23:59:59 represents the time one second before midnight.

The SAS informat/format ***E8601TMw.d*** is used to read write time of day variables.

```
sasTime= input('12:30:15', E8601TM.);
put sasTime E8601TM.;  /* log: 12:20:15 */
```

Midnight is a special case and can be referred to as both 00:00:00 and 24:00:00. The notation 00:00:00 is normally used to represent the beginning of a calendar day and 24:00:00 is normally used to represent the end of a calendar day. The SAS E8601TM. informat will correctly read the time 00:00:00, but will set a time of 24:00:00 to missing.

```
sasTime= input('00:00:00', E8601TM.);
put sasTime E8601TM.;  /* log: 00:00:00 */
sasTime= input('24:00:00', E8601TM.);
put sasTime E8601TM.;  /* log: .        */
```

Date and time

YYYY-MM-DDThh:mm:ss

where the T is used to separate the date and time components.

Example: 2013-04-29T16:30:01 represents thirty minutes and one second after four o’clock in the afternoon of 2013-04-29.

The SAS informat/format ***E8601DTw.*** is used to read and write date and time variables.

Note that 2013-04-29T24:00:00 and 2013-04-30T00:00:00 represent the same point in time, however as noted above, the SAS informats do not recognize 24:00:00 as a valid time; reading a date and time value that has a time value of 24:00:00 will result in a missing value.

```
1  data iso;
2       input @1 isoTime e8601dt. @1 charDate :$27.;
3       put chardate $27. ' ' isotime=e8601dt.  ;
4       datalines;

2013-04-29T16:30:15         isoTime=2013-04-29T16:30:15
2013-04-29T24:00:00         isoTime=.
2013-04-29T00:00:00         isoTime=2013-04-29T00:00:00
```

UTC

Unless otherwise specified, times (either in a time of day or date and time) represent the “local time”. If all transactions are based in the same location and time stamped from the same source there should not be any ambiguity. However, once multiple time sources are involved then we need to deal with ambiguity; the use of UTC will remove the ambiguity.

Coordinated Universal Time (UTC) is the world standard for setting and regulating time zones around the world; UTC has superseded Greenwich Mean Time (GMT), however for most purposes these two are interchangeable. Time zones around the world are designated by their offset from UTC. For example, Peter in Toronto is five hours behind (-05:00) UTC and Xiao Jin in Beijing is eight hours ahead (+08:00) of UTC; every day at 19:00:00 Peter asks Xiao Jin for a dinner date and every day she says 08:00:00 is too early for dinner. When Toronto moves daylight savings (UTC -04:00) in March Peter moves one hour closer to Xiao Jin (China does not adjust for daylight savings) but not close enough for dinner.

By adding the UTC offset to a time or day and time we make the point in time unambiguous. Each of these dates represents the same point in time:

- 2012-04-17T17:00:00Z (UTC)
- 2012-04-17T13:00:00-04:00 (Peter in Toronto during daylight savings)
- 2012-04-18T01:00:00+08:00 (Xiao Jin in Beijing)

The standard has provisions for the omission of components representing smaller units (seconds, minutes), where such precision is not needed. The SAS informat E8601TM. will properly read truncated times as this log shows:

```
1  data iso;
2       input @1 isoTime e8601tm10.2 @1 charDate :$27.;
3       put chardate $27. ' ' isotime=e8601tm12.2  ;
4       datalines;

08:30:15,2                  isoTime=08:30:15.20
08:30:15                    isoTime=08:30:15.00
08:30                       isoTime=08:30:00.00
08                          isoTime=08:00:00.00
```

The SAS ISO informats will not properly read truncated dates:

```
1  data iso;
2       input @1 isoDate e8601da. @1 charDate :$27.;
3       put chardate $27. ' ' isoDate=e8601da.  ;
4       datalines;

2013-04-29                  isoDate=2013-04-29
NOTE: Invalid data for isoDate in line 6 1-10.
2013-04                     isoDate=.
RULE:      ----+----1----+----2----+----3----+----4----+----5----+----6----+----7--
--+----8----+--
6       2013-04
isoDate=. charDate=2013-04 _ERROR_=1 _N_=2
NOTE: Invalid data for isoDate in line 7 1-10.
2013                        isoDate=.
7       2013
isoDate=. charDate=2013 _ERROR_=1 _N_=3
```

```
1 data a;
2  dtid=dhms('29Apr03'd,16,30,01);
3  put dtid;
4  put dtid datetime.;
5  put dtid E8601DZ.;
6  put dtid E8601Dt.;
7 run;
```

Durations and Intervals

ISO 8601 also defines intervals and durations. ***Durations*** represent the "amount of time" between points in time, that is this difference in time; they are meant to be used as part of time intervals. Durations are expressed as positive amounts. An ***interval*** also represents the "amount of time" between two points. While the duration is always the measure of the amount of time, the interval is represented one of three ways:

- a start point and an endpoint
- a duration and an endpoint
- a start point and a duration

Complete Durations

The ISO 8601specification of duration is:

Pn**Y**n**M**n**DT**n**H**n**M**n**S**

n: the value for the designator the follows; that is, nY is n years, nM is n months etc.

P: is the duration designator placed at the start of the duration representation.

Y: is the year designator that follows the value for the number of years.

M: is the month designator that follows the value for the number of months.

W: is the week designator that follows the value for the number of weeks.

D: is the day designator that follows the value for the number of days.

T: is the time designator that precedes the time components of the representation.

H: is the hour designator that follows the value for the number of hours.

M: is the minute designator that follows the value for the number of minutes.

S: is the second designator that follows the value for the number of seconds.

For example, P2Y8M16DT8H15M1S represents a duration of 2 years, 8 months, 16 days, 12 hours, 30 minutes, and 5 seconds. When exchanging durations, the parties must agree on the maximum number of digits each component (i.e. year, month etc.) will have. Leading zeros are not required.

Durations can also be expressed as PYYYY-MM-DDTHH:MM:SS. Once again the leading P indicates an interval will follow, YYYY would be the number of years, MM the number of months etc.. In this representation the interval P2Y8M16DT8H15M1S shown above would look like P0003-08-16T12:30:05; note that all the components the leading zeros are required in this format.

Complete Intervals

As noted above, an interval can be represented in one of three ways. In each case, parts of the interval are separated by a solidus (/)

1. a start point and an endpoint. Here we have two date time variables separated by the solidus (/).
   YYYY-MM-DDTHH:MM:SS/YYYY-MM-DDTHH:MM:SS
   e.g. 2012-12-13T08:30:00/2013-04-29T16:30:00

2. a duration and an endpoint. Here we have an interval (as outlined above) followed by a date time variable.
   PnYmMnDTnHnMnS/YYYY-MM-DDTHH:MM:SS
   e.g. P5D/2013-04-29T16:30:00 – 5 days ending at 2013-04-29 at 16:30
   PYYYY-MM-DDTHH:MM:SS/YYYY-MM-DDTHH:MM:SS
   e.g. P0000-00-05T00:00:00/2013-04-29T16:30:00 – 5 days ending at 2013-04-29 at 16:30

3. a start point and a duration. Similar to point 2.
   YYYY-MM-DDTHH:MM:SS/PnYnMnDYnHnMnS
   e.g. 2013-04-29T16:30:00/P0Y0M5DT0H0M0S
   YYYY-MM-DDTHH:MM:SS/PYYYY-MM-DDTHH:MM:SS
   e.g. 2013-04-29T16:30:00/P0000-00-05T00:00:00

Reading/Writing Intervals Durations.

SAS provides two informats/formats read/write intervals and durations: $N6801E. and $N6801B.; $N6801B. can read both extended and basic ISO 8601 notations whereas $N6801E. will only read the extended notation. If you need to ensure compliance with the extended notation you should use $N6801E. Unlike the informats B8601*, the $N8601* informats do not convert the values to SAS date, time, or datetime variable. These informats convert to SAS defined hex value; this means the values can be decoded and displayed by one of the SAS $N6801* duration formats. For a complete list of the SAS $N8601* see the SAS help.

```
data iso;
     infile datalines pad;
     input @1 charDate :$60.
           @1 isoDate $N8601E50. ;
     put chardate  / isoDate= / isoDate=$N8601E.  //;
     datalines;
2012-12-13T08:30:00/2013-04-29T16:30:00
P5D/2013-04-29T16:30:00
P0000-00-05T00:00:00/2013-04-29T16:30:00
2013-04-29T16:30:00/P0Y0M5DT0H0M0S
2013-04-29T16:30:00/P0000-00-05T00:00:00
;;
run;
```

A partial log:

```
2012-12-13T08:30:00/2013-04-29T16:30:00
isoDate=2012C13083000FFD2013429163000FFD
isoDate=2012-12-13T08:30:00/2013-04-29T16:30:00


P5D/2013-04-29T16:30:00
isoDate=FFFFF05FFFFFFFFC2013429163000FFD
isoDate=P5D/2013-04-29T16:30:00
```

Incomplete and missing intervals and date times

All examples above showed complete dates, date times and intervals, however the standard does allow for incomplete components of a date, date time or interval. The SAS $N8601* informats can read and store values that have missing components. For these informats to properly read and store the values with missing components the missing parts must be designated with a hyphen (**-**) or and **x**; the hyphen (-) replaces all the digits in the component whereas the x only replaces one digit. For example the four digit missing year can be replace by a single hyphen, or by four x's. The following code shows some example of missing components and $N6801E. informat and format.

```
data iso;
     infile datalines pad;
     input @1 charDate :$60.
           @1 isoDate $N8601E50. ;
     put chardate  / isoDate= / isoDate=$N8601E.  //;
     datalines;
2013---29Txx:-:-
2013-04-T12:-:05
2013
P--00-05T00:00:00/2013-04-29T16:30:00
Pxxxx---05T00:00:00/2013-04-29T16:30:00
2013-04-29T16:30:00/PxYxM5DT0H0M0S
2013-04-29T16:30:00/P--xx-05T00:00:00
;;
run;
```

A partial log:

```
Pxxxx---05T00:00:00/2013-04-29T16:30:00
isoDate=FFFFF05000000FFC2013429163000FFD
isoDate=P5DT0H0M0S/2013-04-29T16:30:00


2013-04-29T16:30:00/PxYxM5DT0H0M0S
isoDate=2013429163000FFDFFFFF05000000FFC
isoDate=2013-04-29T16:30:00/P5DT0H0M0S


2013-04-29T16:30:00/P--xx-05T00:00:00
isoDate=2013429163000FFDFFFFF05000000FFC
isoDate=2013-04-29T16:30:00/P5DT0H0M0S
```

CALL IS8601_CONVERT()

The call IS8601_CONVERT() routine is a general purpose routine to create various ISO8601 intervals, durations, start date, end date, and various date, datetime, and duration component.,

The syntax of the routine is:

CALL IS8601_CONVERT(convert-from, convert-to, from-variables , to-variables, <date-time-replacements>)

***convert-from*:** string indicating the source for the conversion is an interval, a datetime and duration value, or a duration value.

convert-to: string indicating what the result of the conversion should be.

from-variables: Specify one variable for an interval value, or two variables for datetime and duration values.

to-variables: Specify one variable for an interval value, or two variables for datetime and duration values.

<***date-time-replacements***>: optional. Specifies date or time component values to use when a month, day, or time component is omitted. If entered, they are separated by commas.

The values for convert-from can be:

- ‘intvl’: the source value for the conversion is an interval value.
- ‘dt/du’: the source value for the conversion is a datetime/duration value.
- ‘du/dt’: the source value for the conversion is a duration/datetime value.
- ‘dt/dt’: the source value for the conversion is a datetime/datetime value.
- ‘du’: the source value for the conversion is a duration value.

There are also undocumented values

- DTn: the source value for the conversion is a series of n datetime component values.
- DUn: the source value for the conversion is a series if n duration component values.
- Dn: the source value for the conversion is a series of n date component values.

convert-to accepts the following values:

- ‘intvl’: create an interval value.
- ‘dt/du’: create a datetime/duration interval.
- ‘du/dt’: create a duration/datetime interval.
- ‘dt/dt’: create a datetime/datetime interval.
- ‘du’: create a duration.

- ‘start’: create a value that is the beginning datetime or duration of an interval value.
- ‘end’: create a value that is the ending datetime or duration of an interval value.

There are also undocumented values

- DTn: create a series of n datetime component values.
- DUn: create a series if n duration component values.
- Dn: create a series of n date component values.

For a complete description of the syntax and arguments see the SAS online help.

Some example calls are (note the N and C suffix are clues that the variables are numeric (N) or character (C)):

- call is8601_convert(‘dt/du’, ’intvl’, startN, duC, intervalC);
  convert the date time and duration (‘dt/du’) variables startN and duC to an interval (’intvl’) intervalC
- call is8601_convert(’intvl’, ‘dt/du’, intervalC, startN, duC);
  convert the interval (’intvl’) intervalC to date time and duration (‘dt/du’) variables startN and duC
- call is8601_convert(’du/dt’, ’start’, duC, endN, startC);
  convert the duration and date time (‘du/dt’) variables duC and endN to a start of period (‘start’) startC
- call is8601_convert(’dt/dt’, ’du’, startN, endN,duC);
  convert the two date time (‘dt/dt’) variables startC and endC to a duration (’du’) duC

With eight possible convert-from values and ten possible convert-to values, we cannot enumerate all of the possibilities for IS8601_CONVERT(), however you should have a basic understanding of its workings. This is a routine that is well suited to the “experimental method” for better understanding.

CONCLUSION

SAS date, datetime, and time variables are often one of the most difficult parts of SAS for new programmers. When ISO8601 is added to the mix even more confusion can arise. In this paper we have tried to show that by understanding the difference between the underlying data representation and the visual representation many of the problems new users have with SAS should disappear. We have also attempted to explain the ISO8601 date standard and how it is implemented in SAS. Once you have an understanding of how these variables are represented, applying SAS functions to manipulate and transform the values becomes easy.

.

ACKNOWLEDGMENTS

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Name: Peter Eberhardt
Enterprise: Fernwood Consulting Group Inc.
Address: 288 Laird Dr.
City, State ZIP: Toronto
Work Phone: (416)429-5705
Fax:
E-mail: peter@fernwood.ca
Web: www.fernwood.ca
Twitter: rkinRobin

Name: Xiaojin Qin
Enterprise: Covance Pharmaceutical R&D Co., Ltd.
Address: Gemdale Plaza, Chaoyang District

City, State ZIP: Beijing 100022
Work Phone: (86-10) 57323125
Fax: (86-10) 57323167
E-mail: xiaojin.qin@covance.com
Web: http://www.covance.com

REFERENCES

ISO8601:2004 "*Data elements and interchange formats — Information interchange — Representation of dates and times*", ISO 2004, Geneva, Switzerland

SAS Institute Inc. *SAS® 9.3 Functions and CALL Routines: Reference*. (Cary, NC: SAS Institute Inc., 2011)

Available at support.sas.com/documentation/cdl/en/lefunctionsref/63354/PDF/default/lefunctionsref.pdf.

SAS Institute Inc. 2011. *SAS® 9.3 Formats and Informats: Reference* . (Cary, NC: SAS Institute Inc., 2011)

Available at http://support.sas.com/documentation/cdl/en/leforinforref/63324/PDF/default/leforinforref.pdf.

SAS Institute Inc. 2008. SAS® 9.2 National Language Support (NLS): Reference Guide. (Cary, NC: SAS Institute Inc, 2009)

Eberhardt, Peter "Why the Bell Tolls 108 Times: An Introduction to SAS® Dates"

Proceedings of the SAS® Global Forum 2013 Conference

Tabachneck, Arthur S., Matthew Kastin，Xia Ke Shan "Sometimes One Needs an Option with Unusual Dates"

Proceedings of the SAS® Global Forum 2012 Conference