Paper SS-07

Why the Bell Tolls 108 times?
Stepping Through Time with SAS®

Peter Eberhardt,
Fernwood Consulting Group Inc, Toronto, ON, Canada
Yunbo Sun,
Canada Post, Ottawa, ON, Canada

ABSTRACT

For many SAS programmers, new or even advanced, the use of SAS date and datetime variables is often very confusing. This paper addresses the problems that the most of programmers have. It starts by looking at the basic underlying difference between the data representation and the visual representation of date, datetime and time variables. From there it discusses how to change data representations into visual representations through the use of SAS formats. Since date manipulation is core to many business process, paper also discusses date arithmetic first by demonstrating the use of simple arithmetic to increment dates; then by moving on to SAS functions which create, extract and manipulate SAS date and datetime variables. Finally, paper demonstrates the use of the %sysfunc macro function and the %let statement to present date, datetime and time variables. This paper is introductory and focuses on new SAS programmers, however, some advanced topics also covered.

This paper is introductory and is intended for new SAS programmers.

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.

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.

Throughout the paper I will usually refer to SAS date variables; in my experience date variables are far more common and the need to manipulate them is subsequently more common. In general much of the discussion can be extended to datetime and time variables.

YOUR FIRST DATE

In a SAS DATA step it is not uncommon to assign a constant to a variable:

```
1    data constants;
2        numVar = 1;
3        charVar = "Hello";
4        put numVar= charVar=;
5    run;

numVar=1 charVar=Hello
```

We can do the same thing with a SAS dates:

```
7    data dates;
8        dateVar = "05apr2011"d;
9        put dateVar= ;
```

```
10   run;

dateVar=18722
```

Two important things are quickly noted in this simple example. First, to assign a date constant (often called a ***date literal***) we need to follow a specific formula, and second, the value printed in the log does not appear to have any resemblance to the value we assigned, Looking at the assignment, the 'formula' is simple: the date is in ddmmmyyyy (or ddmmmyy) format, it is enclosed in quotes (either single or double will work) and there is the letter ***d*** following. The ***d*** tells SAS that the string is a date, not a character string. The date string has to be in ddmmmyyyy (or ddmmmyy) format; the month can me upper case, lower case, or a mixed case. This part is relatively clear to most new SAS programmers. Looking at the value of dateVar displayed in the log we see a number, 18722, that does not appear to have any relationship to the value we assigned, 05Apr2011. What is 18722?

DATE REPRESENTATIONS

The value 18722 is the ***data representation*** of 05Apr2011. Specifically it is an integer representing the number of days from a reference point; in SAS the reference point is 01Jan1960. The value 18277 tells us that 05Apr2011 is 18,277 days from 01Jan1960. Having dates represent the number of days since a reference date 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 that 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.

```
11   data dates;
12       today = "05apr2011"d;
13       tomorrow = today + 1;
14       nextWeek = "12apr2011"d;
15       elapsed = nextWeek - today;
16       put today= tomorrow= nextweek= elapsed=;
17   run;

today=18722 tomorrow=18723 nextWeek=18729 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 18722 and 18733 are essentially of little usable value, 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:

```
19   data dates;
20       today = "05apr2011"d;
21       tomorrow = today + 1;
22       nextWeek = "12apr2011"d;
23       elapsed = nextWeek - today;
24       put today=      today= yymmdd10.    today= worddate23. /
25           tomorrow=   tomorrow= yymmdd10. tomorrow= worddate23. /
26           nextweek=   nextweek= yymmdd10. nextweek= worddate23. /
27           elapsed=;
28   run;

today=18722 today=2011-04-05 today=April 5, 2011
tomorrow=18723 tomorrow=2011-04-06 tomorrow=April 6, 2011
nextWeek=18729 nextWeek=2011-04-12 nextWeek=April 12, 2011
```

```
elapsed=7
```

From this example we can see the data representation of ***today*** is 18722, however we can have multiple visual representations; in this case we show two – ***2011-04-05***, and ***April 5, 2011***. 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.

DATING SERVICES

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?

To convert from visual representations of dates, the sort of 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:

```
data readdates;
     input toDay          yymmdd10.  +1
           tomorrow          date9.  +1
             ;
    put toDay= tomorrow= ;
datalines;
2011-04-05 06apr2011
2011-04-06 07apr2011
;;
run;
toDay=18722 tomorrow=18723
toDay=18723 tomorrow=18724
```

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.

BUT SHE'S NOT MY TYPE

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.

```
42   data baddates;
43        dateVar = "2011-04-05";
44        put dateVar=;
45        numDate = input(dateVar, yymmdd10.);
46        put numDate= numDate= yymmdd10. numDate= date9.;
47   run;

dateVar=2011-04-05
numDate=18722 numDate=2011-04-05 numDate=05APR2011
```

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;
```

GOING STEADY

We have seen how to change the visual representation of the date variable by applying a format. What if we want to use the same format all of the time, do we need to specify the same format each time? Fortunately we can permanently assign a format to a variable so we can display the variable without the need to explicitly specify the format to use. Let's look at the example above where the date was originally read in as a character variable and only displayed in reports. We converted the variable from a character representation to a SAS date representation so we could use the date in calculations. Although we can now do some date arithmetic, the existing reports will look odd. Why? Originally when the variable dateVar was displayed the character string would be displayed (for example 2011-04-045), Now that we have converted the variable to a proper SAS date, a number (for example 18613) will be displayed. If hard core SAS geeks would have problems knowing what18613 is, what are the chances the CEO will?

To ensure a date variable always displays the same way we assign a permanent format to the variable:

```
55   data dates;
56       dateVar = "05apr2011"d;
57       format dateVar yymmdd10.;
58       put dateVar= ;
59   run;

dateVar=2011-04-05
```

From the example we see the use of the format statement; once the format is specified the variable displays using this format. Note that the format statement can go anywhere in the DATA step and it will still work.

Applying the permanent format works seamlessly – until you need to display in a different format. When you need to display the variable in a different format you can override the permanent format by specifying a format when the variable is displayed as this example shows.

```
61   data dates;
62       dateVar = "05apr2011"d;
63       put dateVar= dateVar= date9.;
64       format dateVar yymmdd10.;
65   run;

dateVar=2011-04-05 dateVar=05APR2011
```

In this example we apply a permanent format of yymmdd10. to dateVar; however, we used an override format of date9. to get a different display.

LUNCH DATE

To this point we have been talking about SAS date variables; each value represents a specific day. How do we represent a value that is both the day and the time, for example 05Apr2011 at noon? SAS has a ***datetime*** variable to represent that.

```
66   data datetimes;
67       toDayAtNoon = "05apr2011:12:00:00"dt;
68       put toDayAtNoon=;
69   run;

toDayAtNoon=1617624000
```

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. What about the value 1617624000? It does not look a lot like the 18722 date value for the same day.

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

```
75   data datetimes;
76       toDayAtNoon = "05apr2011:12:00"dt;
77       put toDayAtNoon= toDayAtNoon= datetime23.;
78   run;

toDayAtNoon=1617624000 toDayAtNoon=05APR2011:12:00:00
```

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

WHAT TIME WAS THE LUNCH DATE

A SAS time variable represents the time of day,

```
79   data times;
80        twelveThirty = "12:30:00"t;
81        put twelveThirty=;
82        twelveThirtyPlus = "12:30:00.5"t;
83        put twelveThirtyPlus=;
84   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:

```
106  data times;
107       twelveThirty = "12:30:00"t;
108       put twelveThirty= twelveThirty= time. twelveThirty= timeampm.;
109       twelveThirtyPlus = "12:30:00.5"t;
110       put twelveThirtyPlus= twelveThirtyPlus= time. twelveThirtyPlus=
timeampm14.1;
111  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. I am going split them into two arbitrary groups – one I will call informational and one I will call computational. We will look at these groups next.

INFORMATIONAL FUNCTIONS

The functions I 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. Now let's see how we can use the function to add the current date to a title in a report.

Since we want to use the date as part of a title we need to get the date and convert it to a macro variable. SAS not only provides the function to return the current date, but also a means to easily create a formatted macro variable from the function. A simple and common way to do this is through a DATA _NULL_ step:

```
data _null_;
    td = date();
    call symput('runDate', put(td, yymmdd10.));
run;
title "Peter's Report ... Processed on &rundate";
```

In this simple DATA _NULL_ the current date is captured with the ***date ()*** function; the CALL SYMPUT then creates a macro variable, runDate, with the current date formatted as YYMMDD10. Following the DATA step we can use this macro variable in a title statement. This is simple and effective – but can be done more simply as we will see shortly. Before we do, let's look for some time.

The ***time()*** function returns the current time. As we created a macro variable to display the current date in a title we can create a macro variable to display the current time.

```
data _null_;
    td = today();
    ct = time();
    call symput('runDate', put(td, yymmdd10.));
    call symput('runTime', put(ct, timeampm8.));
run;
title " Peter's Report ... Processed on &rundate Starting at &runTime";
```

This example expands on the previous by creating and using a macro variable, runTime, formatted to look something like 9:30 PM. In this example I used a 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 my needs.

These two examples were simple but effective. However we used a very powerful mechanism, the DATA step, to create the macro variables. Let's see another way to do the same thing:

```
%let runDate = %sysfunc(today(),yymmdd10.);
%let runTime = %sysfunc(time(),timeAMPM8.);
title "Peter'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.

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 I 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 I have extracted data from SQL Server and then tried matching a SQL date variable on a SAS date variable. Many times have I subsequently wondered why I had no matches after the merge – at least until I 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);
    sasTime = timepart(sqlDate);
    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 I am going to focus exclusively on SAS dates; in my experience I regularly have to manipulate dates – calculate intervals, determine next threshold date etc.. When I use datetime and time variables they are used to help uniquely identify and/or sequence records.

Boundaries and Intervals

When dealing with dates we commonly need to know two things; first, how many intervals are there between two date, and second 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 give me a date 3 months from now. In order to answer these issues we need to understand what the boundary, or starting point, for each of these is:

- 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.

```
1    data _null_;
2        start = '01jan2010'd;
3        end = '01jul2010'd;
4        days1 = end - start;
5        days2 = intck('day', start, end);
6        weeks = intck('WEEK', start, end);
7        months = intck('month', start, end);
8        year = intck('year', start, end);
9        put days1= days2= weeks= months= year=;
10   run;

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

In this example we are looking at two dates – 01Jan2010 and 01Jul2010. 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 4) 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 30Jun2010 instead of 1Jul2010 (one day earlier) we get the following:

```
11   data _null_;
12       start = '01jan2010'd;
13       end = '30jun2010'd;
14       days1 = end - start;
15       days2 = intck('day', start, end);
16       weeks = intck('WEEK', start, end);
17       months = intck('month', start, end);
18       year = intck('year', start, end);
19       put days1= days2= weeks= months= year=;
20   run;

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

Here we see the number of days difference is one less, 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 1Jan2010 to 31Dec2010 we get:

```
21   data _null_;
22       start = '01jan2010'd;
23       end = '31dec2010'd;
24       days1 = end - start;
25       days2 = intck('day', start, end);
26       weeks = intck('WEEK', start, end);
27       months = intck('month', start, end);
28       year = intck('year', start, end);
29       put days1= days2= weeks= months= year=;
30   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

```
32   data _null_;
33       start = '31dec2010'd;
34       end = '01jan2011'd;
35       days1 = end - start;
36       days2 = intck('day', start, end);
37       weeks = intck('WEEK', start, end);
38       months = intck('month', start, end);
39       year = intck('year', start, end);
40       put days1= days2= weeks= months= year=;
41   run;

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

Now we go from 31Dec2010 to 01Jan2011, 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).

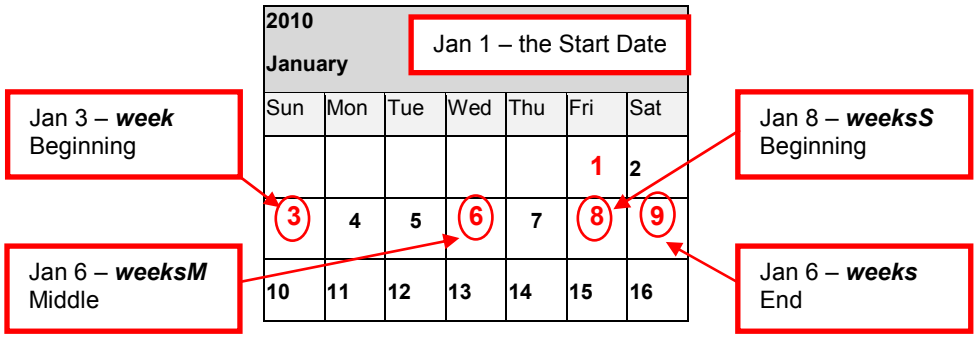
From these simple examples it is easy to see how ***intck()*** calculates intervals; it is also easy to see how you have to be careful that you understand how intervals and boundaries work or your results may not be what you expect.

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.

```
50   data _null_;
51       start = '01jan2010'd;
52       week = intnx('week', start, 1);
53       weeksE = intnx('week', start, 1, 'e');
54       weeksS = intnx('week', start, 1, 's');
55       weeksM = intnx('week', start, 1, 'M');
56       put week= yymmdd10. weeksE= yymmdd10. weeksS= yymmdd10. weeksM= yymmdd10.;
57   run;

week=2010-01-03 weeksE=2010-01-09 weeksS=2010-01-08 weeksM=2010-01-06
```

In this example we are starting with 01Jan2010 and adding one week to it – four different ways. The first way (line 52) 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 52 should return Sunday Jan 3, 2010 – which it does.



The 'end of week' alignment (weeksE) results in Sat Jan 9, the ' same day of week' alignment (weeksS) results in Fri Jan 8, and the 'middle of week' alignmen (weeksM) results in Wed Jan 6. 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’

```
1    data _null_;
2        start = '11jan2010'd;
3        monthStart = intnx('month', start, 0, 'b');
4        monthEnd   = intnx('month', start, 0, 'e');
5        put monthStart= yymmdd10. monthEnd= yymmdd10.;
6    run;

monthStart=2010-01-01 monthEnd=2010-01-31
```

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 not examine the options, but we will see an example that will demonstrate how useful it can be. 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:

```
28   data _null_;
29   start = '11JUN2010'd;
30   yearStart = intnx('year', start, 0, 'b');
31   fiscalStartApr= intnx('year.4',  start, 1, 'b');
32   fiscalStartOct= intnx('year.10', start, 1, 'b');
33   fiscalEndApr   = intnx('year.4',  start, 1, 'e');
34   fiscalEndOct   = intnx('year.10', start, 1, 'e');
35   put yearStart= yymmdd10. /
36   fiscalStartApr= yymmdd10. fiscalEndApr= yymmdd10. /
37   fiscalStartOct= yymmdd10. fiscalEndOct= yymmdd10.;
38   run;

yearStart=2010-01-01
fiscalStartApr=2011-04-01 fiscalEndApr=2012-03-31
fiscalStartOct=2010-10-01 fiscalEndOct=2011-09-30
```

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; I have only attempted to show that with a little scratching the surface can be greatly enlarged.

CONCLUSION

SAS date, datetime, and time variables are often one of the most difficult parts of SAS for new programmers. In this paper I 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. Once you have an understanding of how these variables are represented, applying SAS functions to manipulate and transform the values becomes easy.

CONTACT INFORMATION

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

Peter Eberhardt
Fernwood Consulting Group Inc.
288 Laird Dr
Toronto, ON Canada M4G 3X5
(416)429-5705
peter@fernwood.ca
www.fernwood.ca
www.BrewingMadeEasy.com

Yunbo Sun
Canada Post
2701 Riverside Dr
Ottawa, ON Canada K1A 0B1
(613)734-8744
Yunbo.sun@canadapost.ca

APPENDIX A

In China, one of traditional activities is at the Chinese New Year's Eve, people gather together at temples and wait for the bell to ring 108 times. People believe this can expel unhappiness and bring good luck in the New Year.

Why the bell is struck 108 times? It was recorded in the "Manuscript of Seven Categories" made during the Ming Dynasty that 108 times of striking the bell was equivalent to one year, as there were twelve months, twenty-four Qi (solar terms) and seventy-two Hou (12+24+72=108).

In the ancient system of timekeeping, five days were called a Hou (pentad), and six Hou made up a month, so there were seventy-two Hou periods in one year, therefore the number 108 was calculated according to these time measurements.