

Inline Formatting with ODS Markup

Eric Gebhart, SAS Institute Inc., Cary, NC

ABSTRACT

Explore the power of inline formatting in ODS. And, learn about the super powers of inline formatting with ODS markup and ODS measured; see how easy it is to make your reports look better than ever.

With new and improved syntax, ODS inline formatting is better and more powerful. Learn how to use it and extend it to do even more than you dreamed possible.

‘Twas Brillig, and the Slithy Toves, did gyre and gimble in the wabe

Inline formatting has been around for a few years now, but as of 9.2, it has a new syntax and is more powerful than ever.

However, it can look a bit daunting—much like Lewis Carroll’s Jabberwock, the Jubjub bird, and frumious Bandersnatch. You might be better off running the other way. However, inline formatting isn’t nearly as bad as these characters.

He left it dead, and with its head, he went galumphing back

This paper explains the new syntax for format functions available in most ODS destinations. It explores how these functions work within ODS markup destinations. By putting that knowledge to good use, you can create format functions that more powerfully manipulate an ODS markup destination. This knowledge gives you insight into how ODS markup destinations can be used in new ways. And, last but not least, inline formatting will no longer be as scary as the Jabberwock.

He took his vorpal sword in hand: long time the manxome foe he sought

The first step is to set the ODS escape character to something that won’t interfere with other text in your SAS program. Favorite characters are ~ and ^. There is another way to escape—*ESC*—but a single escape character is easier to type and read. The ODS ESCAPECHAR statement is used to set the escape character.

```
ODS ESCAPECHAR=~;
```

The rest of the syntax is fairly simple. The inline format begins after the escape character with an open curly brace {. The first thing inside the brace is the format function’s name. After that, the arguments for the function are listed, followed by a closing curly brace }.

Here is a simple example that uses an inline format function in a title:

```
title 'This is ^{style [color=red] Red}';
```



STYLE is one of the most powerful built-in format functions. It has the same style override capabilities that have been available in PROC TABULATE, PROC REPORT, and PROC PRINT for years. The usual style caveats from one destination to another apply. Among the more useful style attributes are COLOR, FONT_STYLE, FONT_FAMILY, FONT_WEIGHT, and TEXT_DECORATION, which includes STRIKETHROUGH, UNDERLINE, OVERLINE, and BLINK. Your mileage may vary depending on the destination. BACKGROUND might work, but again, it depends on the destination.

In addition to STYLE, there are other functions available for most destinations:

```
~{dagger}  
~{sigma}  
~{unicode <Hex|name>}  
~{super text}  
~{sub text}  
~{raw type text}  
~{style <style><[attributes]>}  
~{nbspspace count}  
~{newline count}
```

DAGGER and SIGMA are old functions that have been all but replaced by the new UNICODE function. The UNICODE function can print any special character, many of them by name, and if not by name, then by hex value. SUPER and SUB have been around for a while, and they superscript and subscript text. The RAW function is useful for RTF and other markup languages for quickly and easily inserting raw markup code into the output file. The first argument to RAW (if provided) is the destination type intended for RTF, HTML, and LaTeX. The destination type keeps a RAW inline format directed to its destination. RAW is not as useful as it once was because of the way format functions work with ODS markup (which now includes the new TAGSETS.RTF destination). There are better ways to do this. STYLE is probably the single, most useful and powerful format function. NBSPACE inserts a nonbreaking space for destinations that support it. If there is a count, NBSPACE inserts that many nonbreaking spaces. The NEWLINE function inserts a new line. If there is a count, NEWLINE inserts that number of new lines.

One last thing before we move on, and it is just a little thing—the new syntax for format functions enables nesting. The output of one function can be the input to another. This enables scoping of style and simpler coding of complex formatting.

ONE, TWO! ONE, TWO! AND THROUGH AND THROUGH, THE VORPAL BLADE WENT SNICKER-SNACK!

Here is an example using most of these functions:

```
title "Examples of Functions";

title2 'This is ^{style [color=red] Red}';

title3 'Example of ^{nbspspace 3} Non-Breaking Spaces Function';

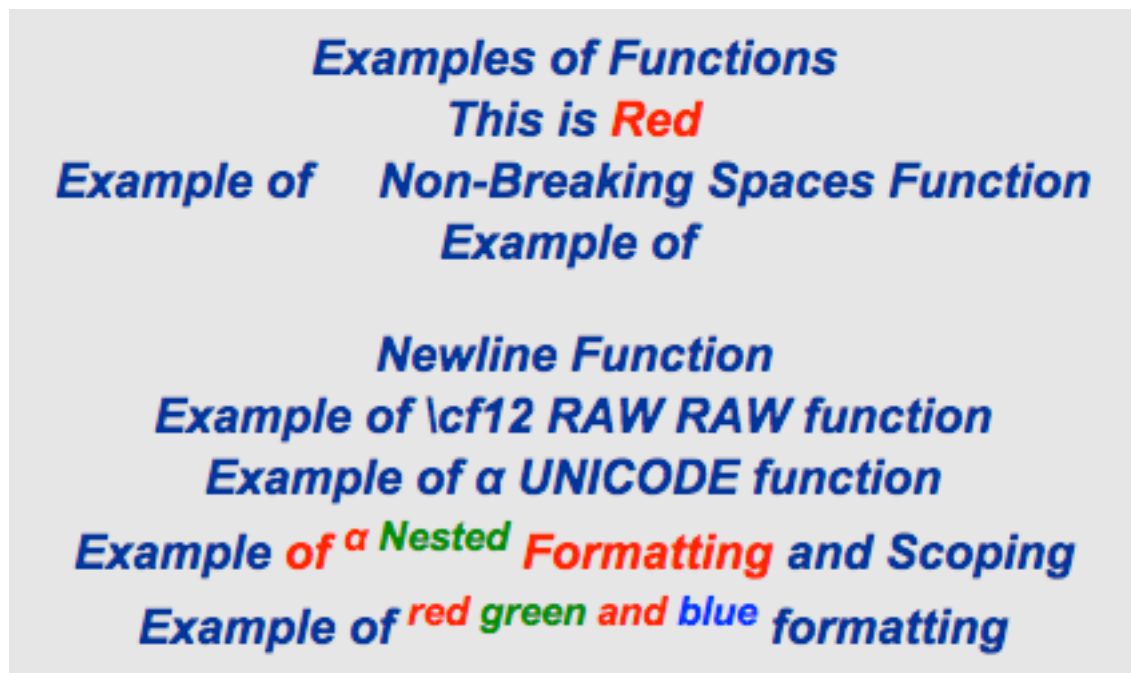
title4 'Example of ^{newline 2} Newline Function';

title5 'Example of ^{raw \cf12 RAW} RAW function';

title6 'Example of ^{unicode 03B1} UNICODE function';

title7 "Example ^{style [foreground=red] of ^{super ^{unicode ALPHA} ^{style [foreground=green] Nested}} Formatting} and Scoping";

title8 "Example of ^{super ^{style [foreground=red] red ^{style [foreground=green] green } and ^{style [foreground=blue] blue}}} formatting";
```



Underlining works everywhere, but strike-through and overlines are not as consistent. Not every target output type supports all of the things that can be done with ODS styles.

```

options nonumber nodate;
ods html file="underline.html";

ods escapechar='^';
title underlin=1 "an underlined title using underlin=";
title3 "^{style [textdecoration=line_through]a line-through title}";
title5 "^{style [textdecoration=overline]An overlined title}";
title7 "^{style [textdecoration=underline]Switching from underline to}
        ^{style [textdecoration=line_through]line-through, then}
        ^{style [textdecoration=overline]overline}.";

proc print data=sashelp.class(obs=1);run;

ods pdf text="^{style [just=r textdecoration=underline color=red]Here is some
random underlined text.}";

ods _all_ close;

```

an underlined title using underlin=.

~~a line-through title~~

An overlined title

Switching from underline to line-through, then overline.

The UNICODE function is one of the most sought-after functions. Here is an example that shows how inline formatting can be embedded in data:

```

ods escapechar='^';

/* Create a table of unicode characters */
data work.unicode;
input @1 name $25. @27 value $4.;
datalines;
Snowman                2603
Black Knight           265E
White Rook             2656
Snowflake              2744
Two Fifths             2156
Greater Than or Equal To 2267
;

/* Create table that will show the name, unicode value, and actual symbol */
proc template;
define table unitable;
  define column name;
    header = 'Name';
  end;
  define column value;
    style={textalign=center};
    header = 'Value';
  end;
  define column symbol;
    style={textalign=center};
    header = 'Symbol';
    compute as '^{unicode ' || value || '}'';
  end;
end;

```

```

end;
run;





/* Make the fonts big */
proc template;
define style styles.bigprinter; parent=styles.printer;
  class systemtitle, data, header /
    fontsize = 40pt
  ;
end;
run;

/* Generate report */
ods pdf file="unicode.pdf" style=styles.bigprinter;
ods html file="unicode.html" style=styles.bigprinter;

data _null_;
  set work.unicode;
  file print ods=(template='unitable');
  put _ods_;
run;

ods _all_ close;

```

| Name | Value | Symbol |
|--------------------------|-------|---------------------------------------------------------------------------------------|
| Snowman | 2603 |  |
| Black Knight | 265E |  |
| White Rook | 2656 |  |
| Snowflake | 2744 |  |
| Two Fifths | 2156 | $\frac{2}{5}$ |
| Greater Than or Equal To | 2267 | \geq |

BEWARE THE JABBERWOCK, MY SON! THE JAWS THAT BITE, THE CLAWS THAT CATCH!

One of the problems with style is that not every target output type is capable of every nuance that ODS styles support.

ODS styles are mostly a superset of what all destinations are capable of. A prime example is Microsoft Excel. There are only a few style attributes that Excel supports. This is a limitation of the SPREADSHEETML specification.

Excel supports color, font face, the font weight bold, and underline, but not overline or strike-through.

SO RESTED HE BY THE TUMTUM TREE, AND STOOD AWHILE IN THOUGHT

What is so special about ODS markup? It's the tagsets. More important, it's how format functions are defined in ODS markup destinations. They are defined as events in the tagset. This realization might make your head spin.

Starting simple, using the HTML destination as an example, let's see what the format functions look like. Knowing a bit of HTML is helpful.

Here is the tagset code for DAGGER:

```
define event dagger;
  start:
    put '&dagger;' ;
end ;
```

Here is the tagset code for SUPER and SUB:

```
define event super;
  start:
    put "<sup>" ;
    put VALUE;
    put "</sup>"
end ;

define event sub;
  start:
    put "<sub>" ;
    put VALUE;
    put "</sub>" ;
end ;
```

These functions are only a little complicated. They have text that they need to handle, but nothing too bad.

The NBSPACE and NEWLINE functions are nearly identical, but quite a bit more complicated. If you can get past the tagset syntax, you see good ole DATA step functions doing DATA step things. First, convert the value to a number, make sure it's no bigger than 256 and no less than 0, and then loop for that many times printing &NBSP for a nonbreaking space. Or, in the case of a new line, print
 for a break in the line.

```
define event nbospace;
  do /if value;
    break /if index(value, "-");
    eval $ncount inputn(value, "3.");
    do /if $ncount > 256;
      eval $ncount 256;
    done;
  else;
    eval $ncount 1;
  done;

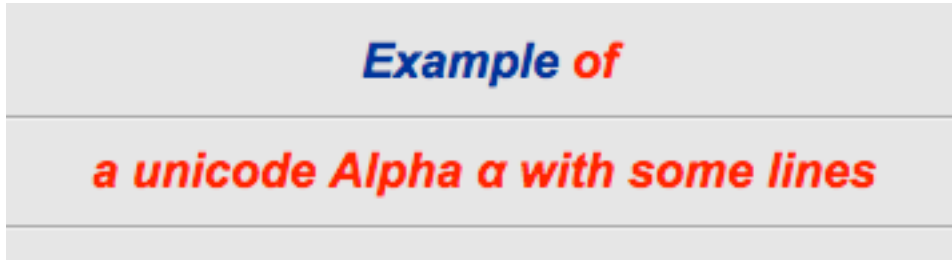
  do /while $ncount;
    put '&nbsp;' ;
    eval $ncount $ncount-1;
  done;
  unset $ncount;
end;
```

Now, the question is: "If format functions are just tagset events, and tagsets are chock-full of events, what other events are there that might be useable as format functions?" The short answer is almost any event. Some make

sense, some don't, but you can always make one that does. You could use inline formatting to create an entire table with rows and cells, but I wouldn't recommend it. There are easier ways to do that.

Just a quick look around in the HTML tagset reveals two functions that could be useful—PAGEBREAK and LINE. PAGEBREAK prints the contents of the style attribute PAGEBREAKHTML, which is usually a line of some sort. LINE prints a simple <HR> line. Let's see what happens if we try that:

```
title1 "Example ^{style [foreground=red] of ^{line} a unicode Alpha ^{unicode ALPHA} ^{newline} ^{line}"
```

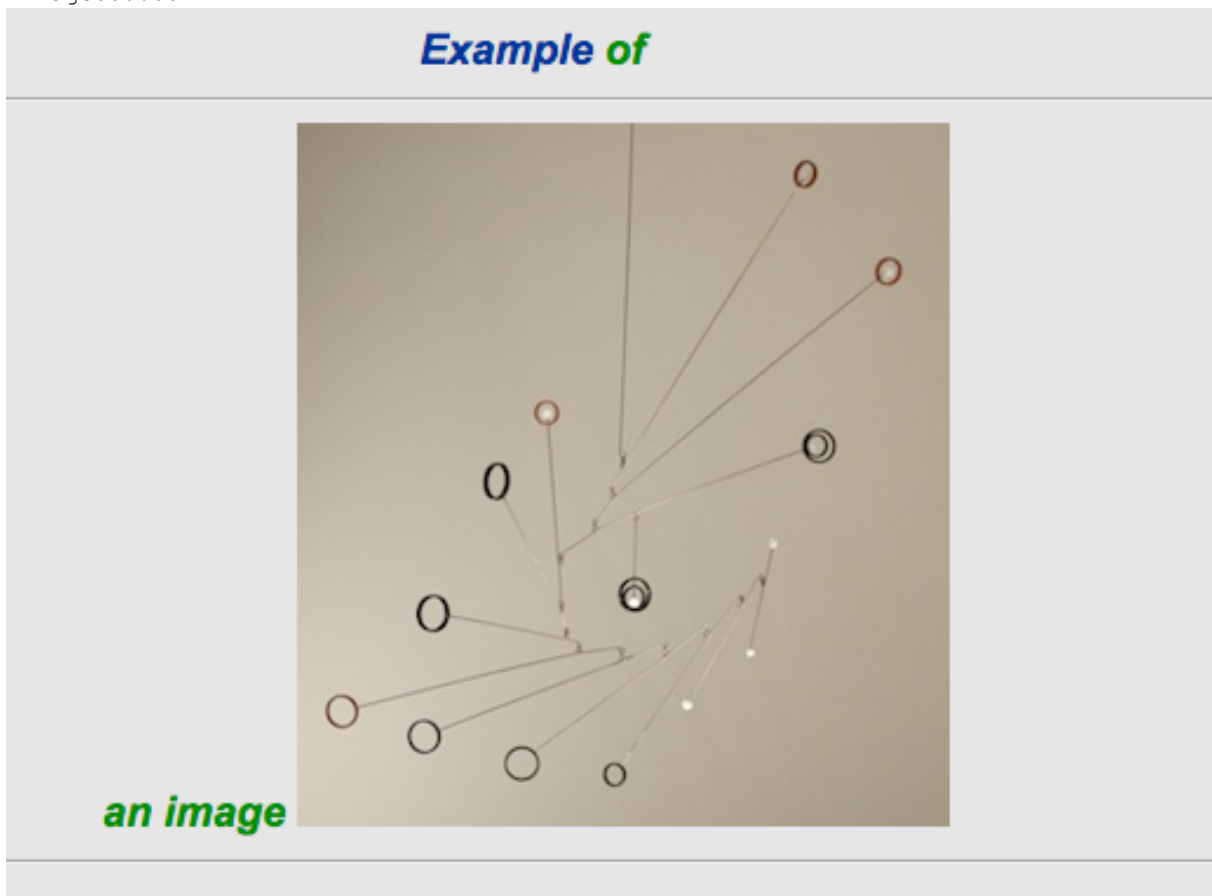


If that works, what else can we do? Lots! How about an IMAGE function?

```
define event inline_image;  
  putq "<img" "src=" value ">";  
end;
```

```
title2 "Example ^{style [color=green] of ^{line} an image {inline_image Droplet.jpg} ^{newline} ^{line}}"
```

```
<image.....>
```



Droplet—A Mobile I Made January, 2009

We've seen SUPER and SUB, and they look like this:

```
title2 'This is ^{super SUPER text}
      ^{sub This is SUB text}
      ^{super More SUPER text}
      ^{sub More SUB text}';
```

This is SUPER text This is SUB text More SUPER text More SUB text

How about a super-duper function? Something that draws a box around the text, and then superscripts and subscripts each word alternately! Silly, I know, but it can be done. Here's how:

```
proc template;
  define tagset tagsets.sdhtml;
    parent=tagsets.html4;

    define event superDuper;

      put '<span style="border-width:1; border-style:solid; border-color:
black;"> ' nl;

      eval $count 1;
      do /if index(value, ' ');
        set $word scan(value, 1, ' ');
        do /while !cmp($word, ' ');

          do /if mod($count, 2);
            put '<sup>' $word '</sup>';
          else;
            put '<sub>' $word '</sub>';
          done;

          eval $count $count + 1;
          set $word scan(value, $count, ' ');
        done;

      else /if $event;
        put '<sup>' value '</sup>';
      done;

      put nl "</span>" nl;

    end;
  end;

run;

ods escapechar="^";

title 'This is ^{superduper SUPER DUPER text}
      This is ^{sub SUB text}';

title2 'This is ^{super SUPER text}
      ^{sub This is SUB text}
      ^{super More SUPER text}
      ^{sub More SUB text}';

ods tagsets.sdhtml file="superduper.html";

proc print data=sashelp.class;
run;

ods _all_ close;
```

*This is **SUPER DUPER**^{text} This is SUB text*

AND, AS IN UFFISH THOUGHT HE STOOD, THE JABBERWOCK, WITH EYES OF FLAME, CAME WHIFFLING THROUGH THE TULGEY WOOD, AND BURBLED AS IT CAME!

The previous example seems like a frivolous use of the super powers of inline format functions and tagsets, but it does show that almost anything can be done. What else can we do? I'm still coming up with ideas, and more frequently than not, necessity is the mother of invention. Embedding images, blocking and unblocking events to change the tagset's behavior, setting and unsetting options on a tagset to change its behavior. There is no rule that says an inline format function has to create output. One of the best and most useful things to do with an inline format function is to read external files of text or markup and place them into output.

Frequently, especially for HTML, you want to put HTML code that is created and maintained by the Web site police into certain places within the report to make the output conform to guidelines. That is easy to do with an inline format function. That same function could place disclaimers and paragraphs of text that are easily maintained elsewhere. That same function could include Flash code in the HTML file, which is something that is becoming more common.

There is tagset code that knows how to read a file. It is just a tagset version of the SAS manual page for FGET. There is cleverness here that makes the READFILE function more manageable. The PROCESS_DATA event is used to print the lines that the READFILE function reads. That makes it possible to leave the READFILE event alone, and repurpose it for other uses by modifying the PROCESS_DATA event. The last thing we need is an actual function to use as an inline format function—that is the INCLUDE_FILE event.

```
define event readfile;

/*-----eric-*/
/*-- Set up the file and open it.                --*/
/*-----13Jun03-*/

set $filrf "myfile";
eval $rc filename($filrf, $read_file);

/*
do /if $debug_level >= 5;
    putlog "File Name" ":" $rc " : " $read_file;
done;
*/

eval $fid fopen($filrf);

/*
do /if $debug_level >= 5;
    putlog "File ID" ":" $fid;
done;
*/

/*-----eric-*/
/*-- datastep functions will bind directly to the --*/
/*-- variable space as it exists.                --*/
/*--                                           --*/
/*-- Tagset variables are not like datastep      --*/
/*-- variables but we can create a big one full  --*/
/*-- of spaces and let the functions write to it. --*/
/*--                                           --*/
/*-- This creates a variable that is 200 spaces so --*/
/*-- that the function can write directly to the --*/
/*-- memory location held by the variable.        --*/
/*-- in VI, 200i<space>                          --*/
/*-----27Jun03-*/
set $file_record "

";

/*-----eric-*/
```



```

/*-- Loop over the records in the file                                --*/
/*-----13Jun03-----*/
do /if $fid > 0 ;

    do /while fread($fid) = 0;

        set $rc fget($fid,$file_record ,200);

        /*
        do /if $debug_level >= 5;
            putlog 'Fget' ':' $rc 'Record' ':' $file_record;
        done;
        */

        set $record trim($file_record);

        trigger process_data;

        /* trimn to get rid of the spaces at the end. */
        /*put trimn($file_record ) nl;*/

    done;
done;

/*-----eric-----*/
/*-- close up the file.  set works fine for this.                    --*/
/*-----13Jun03-----*/

set $rc close($fid);
set $rc filename($filrf);

end;

define event process_data;
    put $record;
end;

define event include_file;
    set $read_file value;
    trigger readfile;
end;

```

As an example, lets read the poem, Jabberwocky, into our html output.

```

ods html file="jabberwocky.html";

ods text="~{style [font_style=italic] ~{include_file jabberwocky.txt}}";

ods html close;

```

"Twas brillig, and the slithy toves Did gyre and gimble in the wabe: All mimsy were the borogoves, And the mome raths outgrabe. "Beware the Jabberwock, my son! The jaws that bite, the claws that catch! Beware the Jubjub bird, and shun The frumious Bandersnatch!" He took his vorpal sword in hand: Long time the manxome foe he sought --So rested he by the Tumtum tree, And stood awhile in thought. And, as in uffish thought he stood, The Jabberwock, with eyes of flame, Came whiffing through the tulgey wood, And burbled as it came! One, two! One, two! And through and through The vorpal blade went snicker-snack! He left it dead, and with its head He went galumphing back. "And, has thou slain the Jabberwock? Come to my arms, my beamish boy! O frabjous day! Callooh! Callay!" He chortled in his joy. 'Twas brillig, and the slithy toves Did gyre and gimble in the wabe; All mimsy were the borogoves, And the mome raths outgrabe.

OK. Close, but not quite. I really did want a new line between each line that was read. But, that is only for this example. What if I really wanted those lines to run together? We could create another inline function to change the behavior of PROCESS_DATA:

```

define event include_newlines;
    do /if cmp (value, 'True');
        set $include_newlines='True';
    else;
        unset $include_newlines;
    end;
end;

```

```
done;  
end;
```

PROCESS_DATA can now be modified to include new lines or not:

```
define event process_data;  
  put $record;  
  put "<br>" /if $include_newlines;  
end;
```

Using the function to change the behavior of the INCLUDE_FILE format function looks like this:ods html
file="jabberwocky.html";

```
ods text="{style [font_style=italic] ~{include_newlines True} ~{include_file  
jabberwocky.txt}}";  
  
ods html close;
```

That's more like it. We now have some nice HTML.

*'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe:
All mimsy were the borogoves,
And the mome raths outgrabe.*

*"Beware the Jabberwock, my son!
The jaws that bite, the claws that catch!
Beware the Jubjub bird, and shun
The frumious Bandersnatch!"*

*He took his vorpal sword in hand:
Long time the manxome foe he sought --
So rested he by the Tumtum tree,
And stood awhile in thought.*

*And, as in uffish thought he stood,
The Jabberwock, with eyes of flame,
Came whiffling through the tulgey wood,
And burbled as it came!*

*One, two! One, two! And through and through
The vorpal blade went snicker-snack!
He left it dead, and with its head
He went galumphing back.*

*"And, has thou slain the Jabberwock?
Come to my arms, my beamish boy!
O frabjous day! Callooh! Callay!"
He chortled in his joy.*

*'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe.*

Jabberwocky, by Lewis Carroll

It is brillig again. Everything is peaceful, mimsy, outgrabe. It is time to reflect on what we have covered. There is the ODS escape character that tells ODS that a function is coming. Functions can be nested and are scoped by curly braces. Not all destinations are capable of all things. ODS markup destinations (which include the new TAGSETS.RTF, HTML, EXCELXP, LATEX, and many others) are unique in that the format functions are simply events within the tagsets. This means anyone can create new functions that can do almost anything. **Contact Information**

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

Eric Gebhart
SAS Institute
SAS Campus Drive
Cary, NC, 27513
E-mail: Eric.Gebhart@sas.com

Web: <http://support.sas.com/rnd/base/ods/odsmarkup>

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

Other brand and product names are trademarks of their respective companies.