PharmaSUG 2026 - Paper ET-215

Reproducing the SAS DATE and TIME formats with {fmtr} package in R

Chen Ling, AbbVie Inc.;
David Bosak, Archytas Clinical Solutions, LLC;

ABSTRACT:

As pharmaceutical organizations transition from SAS® to R for clinical reporting, preserving SAS-compatible date, time, and datetime formats remains a critical challenge for regulatory submissions. In current workflows, SAS formats can be applied during XPT export; however, this approach restricts formatting to the transport file and does not allow SAS-consistent formatted values to be generated or reviewed directly within the R environment. As a result, programmers cannot fully inspect, validate, or quality-control formatted outputs in R prior to export.

The open-source {fmtr} R package addresses this gap by providing the first complete, in-R replication of the SAS DATEw., TIMEw.d, and DATETIMEw.d display formats. These implementations allow SAS-compatible formatting to be applied natively within R environments, enabling consistent presentation throughout tables, listings, figures, and validation workflows, not solely at the point of XPT creation.

This paper introduces the DATEw., TIMEw.d, and DATETIMEw.d formats implemented in {fmtr}, supporting a wide range of widths and precision levels. These implementations enable reliable, SAS-compatible DATE, TIME, and DATETIME formatting directly within R, improving reproducibility and confidence in open-source clinical reporting workflows.

1. INTRODUCTION:

SAS® date, time, and datetime display formats have long been a foundational component of clinical trial reporting. Formats such as DATEw., TIMEw.d, and DATETIMEw.d are deeply embedded in regulatory tables, listings, figures, and submission deliverables, where consistent presentation and predictable rounding behavior are essential for review, validation, and compliance. As pharmaceutical organizations increasingly adopt R for statistical programming and reporting [1-4], maintaining compatibility with these established SAS display standards has become a critical requirement.

In current R-based workflows, SAS-compatible formats are most commonly applied during the XPT export process rather than within the R environment itself. Mainstream XPT-generation approaches in R, including write_xpt() from the {haven} package and xportr_format() from the {xportr} package, rely on a variable-level attribute named "format.sas". By assigning this attribute (e.g., attr(dataset$variable, "format.sas") = "DATE7."), the specified SAS format is preserved and written into the resulting XPT file [5]. However, this attribute-based approach does not affect the representation of the data within R: the formatted display is not reflected in the R dataset, console output, or downstream table, listing, and figure (TLF) generation. As a result, programmers can only observe the formatted values after exporting and inspecting the XPT file [6], and cannot apply or review SAS-consistent formatting directly within the R environment. This limitation prevents SAS-style formats from being used for in-R display, reporting, or quality control.

Although some prior work has attempted to reproduce SAS-style date displays directly within R, existing support remains limited. For example, the CRAN package {SASdates} provides helper functions for selected date formats, including date9() and date11(), but it does not provide a general implementation of the DATEw. family across widths, nor does it support SAS-style TIMEw.d or DATETIMEw.d formatting. As a result, comprehensive in-R replication of SAS-compatible date, time, and datetime display behavior has remained unavailable.

In base R, date and time values are represented using POSIXt objects, whose design imposes several limitations when compared with SAS date and time handling. First, POSIXt restricts fractional-second precision to a maximum of six digits, whereas SAS supports up to twelve digits in TIME/DATETIME formats. Second, POSIXt does not permit negative hour values or hours greater than or equal to 24, while SAS time values may legitimately represent negative durations as well as elapsed times exceeding 24 hours. In addition, R and SAS apply different rounding rules for fractional values at boundary conditions. When the last retained digit is exactly 5, R uses a round-to-even strategy, whereas SAS rounds away from zero. For example, rounding 1.25 to one decimal place yields 1.2 in R but 1.3 in SAS. Moreover, even within SAS itself, the TIMEw.d format can exhibit inconsistent rounding behavior at boundary conditions, with outcomes depending on opaque, black-box internal operations rather than a transparent and deterministic rounding rule.

This paper examines the behavior of the SAS DATEw., TIMEw.d, and DATETIMEw.d formats and presents their implementation in the open-source {fmtr} R package [7]. We first review the SAS formatting rules governing width, precision, component construction, and rounding, and then show how these rules are reproduced within R. Particular attention is given to behaviors that are difficult to support with native R date-time classes, including negative elapsed times, times exceeding 24 hours, extended fractional-second precision, and deterministic rounding. Together, these implementations enable SAS-compatible date and time formatting to be generated and evaluated directly within the R workflow.

2. OVERVIEW OF SAS DATE/TIME FORMATS

2.1 DATEw.

The SAS DATEw. format is a commonly used display format for calendar dates, where w specifies the total width of the formatted output. In SAS, the valid range for w is 5 to 11, with DATE7. as the default when no width is explicitly specified. Internally, SAS stores dates as the number of days since January 1, 1960.

Depending on the specified width, DATEw. produces different textual representations of the same date. Smaller widths generate compact displays using abbreviated month and year values, while larger widths include separators and the full four-digit year. For example, DATE7. displays dates in the form DDMMMYY, whereas wider formats such as DATE9. and DATE11. include separators and expanded year representations. Regardless of width, the formatted output always occupies exactly w characters, with spacing and alignment determined by SAS formatting rules.

The following SAS example illustrates how the same date value is displayed using different DATEw. formats:

```
data date;
length Test $7 Value $11;
Date = "01JAN2025"d;

Test = "date5.";
Value = put(date, date5.);
output;

Test = "date7.";
Value = put(date, date7.);
output;

Test = "date9.";
Value = put(date, date9.);
output;

Test = "date11.";
Value = put(date, date11.);
output;

run;
```

SAS code 1. Overview of SAS DATEw. format

The above code produces the following output data:

	Date	Test	Value
1	23742	date5.	01JAN
2	23742	date7.	01JAN25
3	23742	date9.	01JAN2025
4	23742	date11.	01-JAN-2025

Figure 1. The results of SAS code 1

2.2 TIMEw.d

The SAS TIMEw.d format is a display format for time-of-day or elapsed time values, where w specifies the total output width and d specifies the number of digits to display after the decimal point for fractional seconds. SAS stores time values as the number of seconds (including fractional seconds) since midnight.

The valid range for w is 2 to 20. The d value is optional; when specified, it must be an integer from 0 to 19, and the decimal point is included in the output. When no width is explicitly provided, SAS uses TIME8. as the default format. The formatted output always occupies exactly w characters, with spacing and alignment determined by SAS formatting rules.

Depending on w, the display may include only hours, hours and minutes, or hours, minutes, and seconds. For larger widths, TIMEw.d can display full hh:mm:ss values and fractional seconds. Unlike many “clock time” displays, SAS time formatting can represent negative times and elapsed times exceeding 24 hours, allowing hours to extend beyond two digits (e.g., 119:26:40).

```
data time;
length Test $10 Value1 Value2 $15;

Time1 = 5000.125;
Time2= 50000.9;

Test = "time2.";
Value1 = CAT("'", put(Time1, time2.), "'");
Value2 = CAT("'", put(Time2, time2.), "'");
output;

Test = "time4.";
Value1 = CAT("'", put(Time1, time4.), "'");
Value2 = CAT("'", put(Time2, time4.), "'");
output;

Test = "time8.";
Value1 = CAT("'", put(Time1, time8.), "'");
Value2 = CAT("'", put(Time2, time8.), "'");
output;

Test = "time12.2";
Value1 = CAT("'", put(Time1, time12.2), "'");
Value2 = CAT("'", put(Time2, time12.2), "'");
output;

run;
```

SAS code 2. Overview of SAS TIMEw.d format

Test	Value1	Value2
time2.	' 1'	'13'
time4.	'1:23'	' 13'
time8.	' 1:23:20'	'13:53:21'
time12.2	' 1:23:20.13'	' 13:53:20.90'

Figure 2. Results from SAS code 2.

The examples above highlight that TIMEw.d does not always present time values in a fixed hh:mm:ss form. Depending on the magnitude of the time value, the same format may display different combinations of hours, minutes, and seconds. This adaptive behavior is intentional and can lead to noticeably different presentations for different values. The rules governing this behavior are discussed in detail in the later sections.

2.3 DATETIMEw.d

The SAS DATETIMEw.d format is a display format for combined date and time values, where *w* specifies the total output width and *d* specifies the number of digits displayed after the decimal point for fractional seconds. SAS stores datetime values as the number of seconds since midnight on January 1, 1960. The valid range for *w* is 7 to 40, and when no width is specified, SAS uses DATETIME16. as the default. The *d* value is optional; when specified, it can range from 0 to 39, provided it remains less than *w*. SAS requires a minimum width of 16 to display a complete datetime value including date, hour, minute, and second components, and additional width is needed when fractional seconds are included. As with other SAS display formats, the formatted result always occupies exactly *w* characters, with spacing and alignment determined by SAS formatting rules.

```
data datetime;
length Test $12 Value $30;

DT = '10NOV2026:04:08:05.123'dt;;

Test = "datetime7.";
Value = CAT("'", put(DT, datetime7.), "'");
output;

Test = "datetime12.";
Value = CAT("'", put(DT, datetime12.), "'");
output;

Test = "datetime18.";
Value = CAT("'", put(DT, datetime18.), "'");
output;

Test = "datetime18.1";
Value = CAT("'", put(DT, datetime18.1), "'");
output;

Test = "datetime19.";
Value = CAT("'", put(DT, datetime19.), "'");
output;

Test = "datetime20.1";
Value = CAT("'", put(DT, datetime20.1), "'");
output;

Test = "datetime21.2";
Value = CAT("'", put(DT, datetime21.2), "'");
output;


run;
```

SAS code 3. Overview of SAS DATETIMEw.d format

	Test	Value
1	datetime7.	'10NOV26'
2	datetime12.	' 10NOV26:04'
3	datetime18.	' 10NOV26:04:08:05'
4	datetime18.1	'10NOV26:04:08:05.1'
5	datetime19.	' 10NOV2026:04:08:05'
6	datetime20.1	'10NOV2026:04:08:05.1'
7	datetime21.2	'10NOV2026:04:08:05.12'

Figure 3. Results from SAS code 3.

These examples show that DATETIMEw.d expands progressively as more width becomes available. Smaller widths display only the date or date plus hour, while wider formats include the full date and time, and larger widths with a decimal specification add fractional seconds. This behavior is analogous to DATEw. and TIMEw.d, but combines both date and time components within a single display format.

3. BEHIND THE FORMATS

3.1 DATEw.

Behind the scenes, SAS represents date values as the number of days elapsed since January 1, 1960. When a DATEw. format is applied, this numeric value is first converted into its corresponding year, month, and day components using standard calendar arithmetic. Once the year, month, and day components are derived, the DATEw. format applies width-specific display rules to arrange and present these components, allowing the same underlying date value to be rendered in multiple ways depending on the specified width w. The following illustrative example displays a single date value using a range of DATEw. formats from DATE5. through DATE11. See appendix SAS code S1 for the example code.

Test	Value
date5	'01JAN'
date6	' 01JAN'
date7	'01JAN25'
date8	' 01JAN25'
date9	'01JAN2025'
date10	' 01JAN2025'
date11	'01-JAN-2025'

Figure 4. SAS results for format date5 to date11.

The results demonstrate that the day and month components remain consistent across all DATE formats, while the year representation varies by width. For widths $5 \leq w < 7$, the year component is not displayed. For $7 \leq w < 9$, only the last two digits of the year are shown. For $w \geq 9$, the full four-digit year is displayed, and when $w = 11$, separators are added so that the day, month, and year components are separated by hyphens (DD-MMM-YYYY). In addition, for all even widths, SAS inserts a leading blank space at the beginning of the formatted string (e.g., " 01JAN60" for DATE8.), so that the remaining $w - 1$ characters contain the date value and the total output occupies exactly w characters.

3.2 TIMEw.d

The TIMEw.d format is composed of two closely related components and follows a complex set of formatting rules. One component determines how the hour, minute, and second fields are constructed based on the time value and available width, while the other controls how fractional seconds are rounded and displayed. These components are described separately in the following subsections.

3.2.1 Time Component Construction

The time component of TIMEw.d is constructed from the underlying numeric value in seconds after any required rounding step has been resolved. The value is decomposed into hour, minute, and second fields, and the full character form is first built as h:mm:ss. Unlike ordinary clock-time displays, the hour field is not restricted to two digits: it may be a single digit, multiple digits for elapsed times greater than or equal to 24 hours, or include a leading minus sign for negative values. As a result, the visible width of the hour field is dynamic and depends on the magnitude and sign of the value being formatted.

Once the full h:mm:ss string has been formed, TIMEw.d determines how much of that string can be displayed within the requested width. Let hw denote the number of characters before the first colon, that is, the displayed width of the hour field including a minus sign when present. If the specified width is smaller than hw, the value cannot be represented and an overflow marker (**) is returned. If the width is at least hw but less than hw + 3, only the hour field is shown. If the width is at least hw + 3 but less than hw + 6, the display includes hours and minutes in h:mm form. If the width is at least hw + 6 but less than hw + 8, the display includes the full h:mm:ss portion, but all fractional seconds are suppressed. Only when the width reaches hw + 8 can the decimal point and at least one fractional digit be retained. Any unused positions are filled with leading blanks so that the final formatted result occupies exactly w characters.

The following illustrative example displays three time values using a range of TIMEw.d formats from TIME2.1 through TIME11.1. The selected values are chosen so that their formatted hour fields occupy three, two, and one character, respectively, allowing the width-dependent display rules to be compared side by side. See appendix SAS code S2 for the example code.

Test	Value_hw1	Value_hw2	Value_hw3
time2.1	' 1'	'13'	'**'
time3.1	' 1'	' 13'	'119'
time4.1	'1:23'	' 13'	' 119'
time5.1	' 1:23'	'13:53'	' 119'
time6.1	' 1:23'	' 13:53'	'119:26'
time7.1	'1:23:20'	' 13:53'	' 119:26'
time8.1	' 1:23:20'	'13:53:20'	' 119:26'
time9.1	'1:23:20.1'	' 13:53:20'	'119:26:40'
time10.1	' 1:23:20.1'	'13:53:20.1'	' 119:26:40'
time11.1	' 1:23:20.1'	' 13:53:20.1'	'119:26:40.1'

Figure 5. Results for format time2. to time11.

3.2.2 Fractional Second Handling

Fractional seconds in TIMEw.d are shown only when sufficient width remains after the main time component has been constructed. In practice, SAS appears to prioritize display of the integer time portion first and then use any remaining space for the decimal point and fractional digits. If the displayed integer portion has length int_length, then the maximum number of fractional digits that can fit is w - int_length - 1, where the extra one character accounts for the decimal point. Thus, the actual number of fractional digits displayed is determined by the smaller of the requested decimal length d and the maximum length allowed by the available width. When the width is too small, the fractional portion is omitted entirely, even if a decimal specification is present.

For example, consider the value 13:53:20.123. The integer portion, 13:53:20, has length 8. If the format TIME11.3 is applied, the maximum number of fractional digits that can be displayed is w - 8 - 1 = 2, where one additional position is reserved for the decimal point. Therefore, although d = 3 is requested, only two decimal places can appear in the output, and the formatted result is 13:53:20.12.

When fractional digits are displayed, the value is rounded to the retained precision. If rounding causes the fractional portion to carry into the next whole second, the time component is updated accordingly before output is produced. In addition, if the rounded fractional part contains fewer digits than the displayed decimal length, or if the original time value contains no fractional part, trailing zeros are padded so that the output contains exactly the displayed number of decimal places.

3.3 DATETIMEw.d

The DATETIMEw.d format can be understood as a hybrid of DATEw. and TIMEw.d, but it is not formed by simply concatenating the results of those two formats. Instead, the total width is allocated across a combined date-time display, so the date and time portions interact. For widths below 18, no fractional seconds are shown, and the format expands progressively as width increases: the output begins with the date only, then adds the hour, then the hour and minute, and finally the full date:hh:mm:ss form. In this range, the date portion uses ddMMMyy for widths 7 to 8 and ddMMMyyyy for width 9; at larger widths below 18, the shorter ddMMMyy form is retained so that more of the time portion can be displayed. Interestingly, when w = 17 and d > 0, the output includes a decimal point but no fractional digits, because the available width is sufficient for the nonfractional datetime and the decimal point only. The following example applies DATETIME7. through DATETIME17. to the same datetime value, 28JUN2025:12:34:56, to illustrate this behavior. Example code can be found in Appendix SAS code S3.

	Test	Value
1	datetime7.	'28JUN25'
2	datetime8.	' 28JUN25'
3	datetime9.	'28JUN2025'
4	datetime10.	'28JUN25:12'
5	datetime11.	' 28JUN25:12'
6	datetime12.	' 28JUN25:12'
7	datetime13.	'28JUN25:12:34'
8	datetime14.	' 28JUN25:12:34'
9	datetime15.	' 28JUN25:12:34'
10	datetime16.	'28JUN25:12:34:56'
11	datetime17.	' 28JUN25:12:34:56'
12	datetime17.1	'28JUN25:12:34:56.'

Figure 6. Results for datetime7. To datetime17.

For widths of 18 or greater, DATETIMEw.d can be understood as allocating the available width in a fixed order. First, SAS preserves enough space for the minimum nonfractional datetime display, ddMMMyy:hh:mm:ss, which requires 16 characters. Next, any remaining width is allocated to the decimal point and fractional seconds, up to the requested precision. After that, if at least two additional characters are still available, the date portion is expanded from ddMMMyy to ddMMMyyyy, yielding ddMMMyyyy:hh:mm:ss. Finally, if unused positions still remain after the date, time, and fractional portions have been formed, they are padded as leading spaces so that the formatted result occupies exactly w characters. This behavior shows that DATETIMEw.d is not simply a concatenation of DATEw. and TIMEw.d, but a width-managed format in which the date and time portions compete for the same output space. The following example uses 28JUN2025:12:34:56.995 to illustrate this allocation rule. Example code can be found in Appendix SAS code S4.

	Test	Value
1	datetime18.2	'28JUN25:12:34:56.9'
2	datetime21.2	'28JUN2025:12:34:56.99'
3	datetime22.4	' 28JUN25:12:34:56.9950'
4	datetime33.2	' 28JUN2025:12:34:56.99500'

Figure 7. Results for testing datetime width-allocation

The results illustrate the width-allocation rule of DATETIMEw.d when fractional seconds are requested. For DATETIME18.2, the available width is first used to preserve the minimum nonfractional form, ddMMMyy:hh:mm:ss, and the remaining space allows only the decimal point and one fractional digit, producing 28JUN25:12:34:56.9. For DATETIME21.2, after accommodating the decimal point and two fractional digits, two additional positions still remain, so the year expands from ddMMMyy to ddMMMyyyy, yielding 28JUN2025:12:34:56.99. In contrast, DATETIME22.4 shows that fractional precision is prioritized over year expansion: after reserving space for four decimal places, only one extra position remains, which is insufficient to expand the year to four digits. As a result, the output remains in the short-year form, 28JUN25:12:34:56.9950, with the unused position padded as a leading blank. Finally, the last result shows that once the date, time, and fractional portions have been fully formed, any remaining width is padded on the left, producing leading spaces before 28JUN2025:12:34:56.99500.

4. INTRODUCTION TO {fmtr} PACKAGE

Given the complexity of SAS display formats, a natural question arises: is it possible to reproduce SAS-style formatting directly within R? The answer is: Yes!

The **{fmtr}** R package was introduced in 2020 to address this exact need by providing tools that replicate the behavior of commonly used SAS formats in an R-native environment. Central to this functionality is the fapply() function, short for *format apply*. The design of fapply() closely mirrors the SAS PUT() function, allowing users to input a value along with a SAS format string and return the formatted result as a character string.

For example, the SAS BEST. format, which is the default format that SAS uses for numbers, displaying numeric values using a compact representation based on available width, can be applied in R as follows:

```
# Apply best6 format
ret <- fapply(12345678.12345, "best6")

# View Results
ret

# [1] "1.23E7"
```

Next, let's also see an example of applying the "8.2" SAS format to decimal value:

```
# Apply 8.2 format
ret <- fapply(12345.12345, "%8.2f")

# View Results
ret

# [1] "12345.12"
```

In this example, the R format string "%8.2f" corresponds directly to the SAS 8.2 format. The percent sign introduces the format specification, the f denotes floating-point formatting, and the 8.2 defines a total width of eight characters with rounding to two decimal places.

For further information on the {fmtr} package, see the following link: https://fmtr.r-sassy.org/index.html

For further explanation of R formatting codes, see the following: https://fmtr.r-sassy.org/reference/FormattingStrings.html

5. IMPLEMENTATION OF DATE/TIME FORMATS WITH {fmtr}

5.1 DATEw.

The implementation of the DATEw. format in R differs from SAS primarily due to differences in how numeric date values are internally represented. In SAS, dates are stored as the number of days since **January 1, 1960**, whereas in R, numeric date values represent the number of days since **January 1, 1970**. As a result, numeric date values originating from SAS must be converted before they can be formatted correctly in R. This conversion can be performed as follows:

```
r_date <- as.Date(sas_num_date, origin = "1960-01-01")
```

Once converted, the DATEw. format can be applied directly using fapply().

If the input value is already a Date or POSIXt object, no conversion is required. In these cases, fapply() operates directly on the date component of the object and applies the specified DATEw. format accordingly.

Figure 8 demonstrates how the DATEw. format is applied to an R Date object using fapply(), with results matching those produced by SAS in Figure 4

```
>   fapply(r_date,"date5.")
[1] "01JAN"
>   fapply(r_date,"date6.")
[1] " 01JAN"
>   fapply(r_date,"date7.")
[1] "01JAN25"
>   fapply(r_date,"date8.")
[1] " 01JAN25"
>   fapply(r_date,"date9.")
[1] "01JAN2025"
>   fapply(r_date,"date10.")
[1] " 01JAN2025"
>   fapply(r_date,"date11.")
[1] "01-JAN-2025"
```

Figure 8. R results of applying format date5. To date11. with fapply()

5.2 TIMEw.d

The {fmtr} implementation of the TIMEw.d format is designed to closely replicate SAS display behavior while operating natively within the R environment. The fapply() function accepts numeric values, POSIXt, hms and difftime objects as input. Internally, POSIXt and hms objects are first converted to their numeric representations in seconds before entering the formatting procedure. This unified numeric processing ensures consistent behavior across input types. The following subsections illustrate how {fmtr} handles negative time values, elapsed times exceeding 24 hours, and rounding behavior in detail.

5.2.1 Handling with negative time

One limitation of base R is that native POSIXt objects represent calendar date-times and do not naturally support the display of negative elapsed times. The {fmtr} implementation of TIMEw.d overcomes this limitation for numeric, hms, and difftime inputs by treating them as signed numbers of seconds and formatting them directly in a SAS-compatible way. When a negative value is supplied, fapply() formats the corresponding absolute time value first and then prefixes the result with a minus sign. The sign is treated as part of the displayed hour field and therefore counts toward the total output width. As a result, negative values follow the same width-dependent rules as positive values, but require one additional character. The example below shows that negative elapsed times can be formatted directly for numeric, hms, and difftime inputs. Depending on the width, the same value may be displayed as -1, -1:23, -1:23:20, or -1:23:20.125.

```
> fapply(-5000.125, "time2.")
[1] "-1"
> fapply(-5000.125, "time5.")
[1] "-1:23"
> fapply(-5000.125, "time8.")
[1] "-1:23:20"
> fapply(-5000.125, "time12.3")
[1] "-1:23:20.125"
```

```
> x_hms <- as_hms(-5000.125)
> fapply(x_hms, "time12.3")
[1] "-1:23:20.125"
> x_diff <- as.difftime(-5000.125, units = "secs")
> fapply(x_diff, "time12.3")
[1] "-1:23:20.125"
```

5.2.2 Handling with time exceeding 24 hours

Base R does not naturally support SAS-style time displays with hours greater than or equal to 24, because native POSIXt objects represent clock-based date-times rather than elapsed times. The {fmtr} implementation of TIMEw.d addresses this limitation for numeric, hms, and difftime inputs by treating them as elapsed seconds. As a result, the hour field is not restricted to 0 to 23, but may extend as needed. The example below shows that elapsed times greater than 24 hours can be formatted directly for numeric, hms, and difftime inputs. Depending on the width, the value may be displayed as 119, 119:26, 119:26:40, or 119:26:40.125, which also matches the SAS results shown in Figure 5.

```
> fapply(430000.125, "time2.")
[1] "**"
> fapply(430000.125, "time3.")
[1] "119"
> fapply(430000.125, "time6.")
[1] "119:26"
> fapply(430000.125, "time9.")
[1] "119:26:40"
> fapply(430000.125, "time13.3")
[1] "119:26:40.125"
```

```
> x_hms <- as_hms(430000.125)
> fapply(x_hms, "time13.3")
[1] "119:26:40.125"
> x_diff <- as.difftime(430000.125, units = "secs")
> fapply(x_diff, "time13.3")
[1] "119:26:40.125"
```

5.2.3 Handling with rounding

Another limitation of native R time formatting is that visible fractional seconds are capped at six digits. In base R, the %OSn specification for POSIXt output supports only 0 <= n <= 6 decimal places, so native formatting of POSIXt, hms, and difftime objects cannot display more than six fractional digits. In contrast, the {fmtr} implementation of TIMEw.d supports fractional-second display up to 12 digits. For numeric inputs, this precision is handled directly. For hms and difftime inputs, it applies when the underlying stored value retains more than six decimal places, even though base R does not natively display them.

A second challenge is the rounding behavior of SAS itself. During testing, we observed that TIMEw.d does not always behave like a simple decimal rounding applied uniformly to the displayed fractional seconds. For example, under TIME12.3, the values 99.3765 and 67.3765 have the same apparent fractional part, .3765, yet SAS formats them differently as 0:01:39.377 and 0:01:07.376, respectively. At first, this behavior appeared to be explainable by floating-point representation. However, further investigation showed that this is not the case: the two values have the same binary fractional representation, 0100000001011000110110000001100010010011011101001011110001101010, and when converted back, both yield the same decimal fractional part, 0.37649999999999295142. This suggests that the inconsistent behavior is not caused simply by differences in floating-point presentation, but by additional internal processing within the SAS formatting routine.

A practical solution in SAS is to round the numeric time value first and then apply the format. The {fmtr} implementation follows the same principle. The input is first rounded using common::roundup(), which applies a SAS-like round-up rule, and the TIMEw.d format is then applied to the rounded result. This makes the rounding step explicit and deterministic within R, while avoiding the inconsistent boundary behavior observed when relying on SAS formatting alone.

5.3 DATETIMEw.d

The {fmtr} implementation of DATETIMEw.d applies SAS-style datetime formatting directly within R for both numeric and POSIXt inputs. Internally, POSIXt values are converted to their numeric representations in seconds before formatting, allowing the same DATETIMEw.d logic to be applied consistently across supported input types.

Figure 9 demonstrates how the DATETIMEw.d format is applied to an R POSIXt object using fapply(), with results matching those produced by SAS in Figure 7.

```
> dt <- as.POSIXct("2025-06-28 12:34:56.995",tz="UTC")
> fapply(dt, "datetime18.2")
[1] "28JUN25:12:34:56.9"
> fapply(dt, "datetime21.2")
[1] "28JUN2025:12:34:56.99"
> fapply(dt, "datetime22.4")
[1] "28JUN25:12:34:56.9950"
> fapply(dt, "datetime33.5")
[1] "         28JUN2025:12:34:56.99500"
```

Figure 9. R results of applying datetime formats with fapply()

Although the implementation reproduces most SAS DATETIMEw.d behavior, two differences should be noted. First, SAS uses UTC as the default time zone, whereas the {fmtr} implementation uses the local system time zone for numeric inputs and for POSIXt inputs without an explicit time zone. Second, SAS and R use different numeric origins for datetime values: SAS counts seconds from 1960-01-01 00:00:00 UTC, while R uses 1970-01-01 00:00:00 UTC. As a result, numeric datetime values originating from SAS must be converted before they can be formatted correctly in R.

In the example below, when no time zone is specified, the numeric input 1 is formatted according to the local system time zone. In my case, with the local time zone set to EST, it is displayed as 31DEC1969:19:00:01. After changing the time zone to "UTC", the same value is displayed as one second after the R reference datetime, 1970-01-01 00:00:00.

```
> fapply(1,"datetime19.")
[1] "31DEC1969:19:00:01"
> Sys.setenv(TZ = "UTC")
> fapply(1,"datetime19.")
[1] "01JAN1970:00:00:01"
```

6. CONCLUSION

In conclusion, this paper describes the behavior of the SAS DATEw., TIMEw.d, and DATETIMEw.d formats and demonstrates how they can be implemented natively in R through the {fmtr} package. The resulting implementations reproduce key SAS formatting behaviors across a broad range of widths and precisions, including width-dependent display rules, fractional-second handling, negative elapsed times, times exceeding 24 hours, and explicit rounding control.

By making these formats available directly within R, {fmtr} allows formatted values to be generated, inspected, and validated throughout the reporting workflow rather than only at the point of XPT export. This improves transparency during development and quality control, and provides a practical framework for producing SAS-compatible outputs in open-source clinical reporting. Although some differences between SAS and R remain, such as time zone defaults and numeric date-time origins, the implementations described here substantially extend native R formatting capabilities and support more consistent, reproducible reporting workflows.

REFERENCES

[1] J. Yan and M. Yang, "Schema-Preserving Generation of Clinical TLF Templates and Executable R Code via Iterative LLM-Guided Debugging," in *PharmaSUG 2026 Conference Proceedings*, Boston, 2026.

[2] J. Zhang and A. Zhang, "Improving AI SAS-to-R Code Migration via an Intermediate Design Document Layer," in *PharmaSUG 2026 Conference Proceedings*, 2026.

[3] C. Ling and Y. Wang, "Writing SAS MACROs in R? R functions can help!," in *PharmaSUG 2025 conference proceedings*, San Diego, CA, 2025.

[4] J. Yan and T. Tian, "Automating SAS and R Code Interpretation and Debugging: A Practical

Pipeline for Statistical Programmers," in *PHUSE US CONNECT*, Orlando, 2025.

[5] Y. Wang and C. Ling, "SAS and R in Action: Comparison of XPT File Creation," in *SESUG Conference Proceedings*, Cary, 2025.

[6] Y. Wang and C. Ling, "Controlling attributes of .xpt files generated by R," in *PharmaSUG 2025 conference proceedings*, San Diego, CA, 2025.

[7] D. Bosak and C. Ling, "fmtr: Easily Apply Formats to Data," 2026. [Online]. Available: R package version 1.7.3, https://github.com/dbosak01/fmtr.

[8] C. Ling and Y. Wang, "TLFQC: A High-compatible R Shiny based Platform for Automated and Codeless TLFs Generation and Validation," in *PharmaSUG 2025 Conference Proceedings*, San Diego, 2025.

CONTACT INFORMATION

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

Chen Ling

AbbVie Inc.

chen.ling@abbvie.com

David J. Bosak

Archytas Clinical Solutions, LLC

dbosak01@gmail.com

APPENDIX

```
%macro TestDateFullRange;
%let v1 = "01JAN2025"d;
data TestDateFullRange;
length Test $6 Value $20;
%do nm = 5 %to 11;
 Test = "date&nm.";
 Value = CAT("'", put(&v1., date&nm..), "'");
 output;
%end;
run;
%mend;
%TestDateFullRange;
```

SAS code S1. Example SAS code for testing date format full range

```
%macro TestTimeComponentRange;
%let v1 = 5000.125;
%let v2 = 50000.125;
%let v3 = 430000.125;

data TestTimeComponentRange;
length Test $10 Value_hw3 Value_hw2 Value_hw1 $20;
%do nm = 2 %to 11;
 Test = "time&nm..1";
 Value_hw1 = CAT("'", put(&v1., time&nm..1), "'");
 Value_hw2 = CAT("'", put(&v2., time&nm..1), "'");
 Value_hw3 = CAT("'", put(&v3., time&nm..1), "'");
 output;
%end;
run;
%mend;
% TestTimeComponentRange;
```

SAS code S2. Example SAS code for testing time component construction across widths

```
%macro TestDatetimeUnder18;
%let v1 = '28JUN2025:12:34:56'dt;

data TestDatetimeUnder18;
  length Test $12 Value $30;

  %do nm = 7 %to 17;
    Test = cats("datetime", &nm., ".");
    Value = cat("'", put(&v1., datetime&nm..), "'");
    output;
  %end;

  test = "datetime17.1";
  value = cat("'",put(&v1., datetime17.1), "'");
  output;
run;
%mend;

%TestDatetimeUnder18;
```

SAS code S3. Example SAS code for testing DATETIME formats with w < 18

```
%macro TestDatetimeAllocation;
%let v1 = '28JUN2025:12:34:56.995'dt;

data TestDatetimeAllocation;
  length Test $12 Value $100;



  Test = "datetime18.2";
  Value = CAT("'", put(&v1., datetime18.2), "'");
  output;

  Test = "datetime21.2";
  Value = CAT("'", put(&v1., datetime21.2), "'");
  output;

  Test = "datetime22.4";
  Value = CAT("'", put(&v1., datetime22.4), "'");
  output;

  Test = "datetime33.5";
  Value = CAT("'", put(&v1., datetime33.5), "'");
  output;
run;
%mend;

%TestDatetimeAllocation;
```

SAS code S4. Example SAS code for testing datetime component construction across widths