

Time Class

Software design description v.1

Douglas Wilhelm Harder

October 2, 2026

1 Purpose

This document describes the design of `Time_hms`, the first of two classes implementing the accompanying requirements specification. The requirements specification states what operations a duration must support and what results those operations must produce. A software design description explains how a particular implementation will achieve those results.

In this first design, the internal representation follows the way humans commonly describe durations: a sign, a number of hours, a number of minutes, and a number of seconds. For example, a duration of two hours, fifteen minutes, and thirty seconds can be represented directly by storing those three components. This initially seems the most natural choice.

A second class, `Time_s`, will implement the same observable behavior using a different representation. The objective is not to introduce every possible operation on durations. It is to compare two reasonable designs that satisfy the same modest specification, and to discover that the choice of representation affects the complexity of the implementation.

2 Design overview

A `Time_hms` object will contain four private data members:

```
std::uint32_t hours_;
std::uint8_t  minutes_;
std::uint8_t  seconds_;
bool          is_negative_;
```

The hour, minute, and second fields store the *magnitude* of the duration; the Boolean records its sign. The number of hours is not restricted to 23, because the object represents a duration rather than a time of day. Thus, 37:15:42 is a perfectly reasonable value.

DES-1.1 The hours field shall have type `std::uint32_t`. Minutes and seconds shall have type `std::uint8_t`; each requires only values from 0 through 59. The sign shall have type `bool`.

DES-1.2 The three component fields shall always store non-negative values. A negative duration shall be represented by setting `is_negative_` to `true`, not by storing a negative component.

The advantage of this representation is immediately apparent: the public constructor receives essentially the same components that the class stores; the accessors can return stored fields; and the output routine can format those fields without first decomposing a larger numerical value. The cost of this convenience will become apparent when we implement arithmetic.

3 Why a separate sign is required

It might appear simpler to use a signed integer for the hours and allow its sign to describe the entire duration. For example, one could imagine representing `-2:15:30` with:

```
hours_   = -2
minutes_ = 15
seconds_ = 30
```

But this convention fails for durations shorter than one hour. The duration `-0:30:00` has zero whole hours and is nevertheless negative. The problem is even more obvious for `-0:00:01`: both the hour and minute components are zero. In integer arithmetic, `-0` and `0` are the same value. No sign can be recovered from an integer hour component equal to zero.

One could invent special rules or use negative minutes and seconds, but then a single duration could have multiple representations. For example, would negative ninety seconds be represented as negative one minute and thirty seconds, or as zero minutes and negative ninety seconds? Such alternatives would complicate comparisons, output, and arithmetic even further.

DES-2.1 The sign of the complete duration shall therefore be stored separately in `is_negative_`. All three numerical components shall describe only the magnitude.

Displayed value	<code>is_negative_</code>	<code>hours_</code>	<code>minutes_</code>	<code>seconds_</code>
<code>2:15:30</code>	false	2	15	30
<code>-2:15:30</code>	true	2	15	30
<code>0:30:00</code>	false	0	30	0
<code>-0:30:00</code>	true	0	30	0
<code>0:00:01</code>	false	0	0	1
<code>-0:00:01</code>	true	0	0	1

This also explains why the public constructor accepts only non-negative components. Writing `Time_hms{ -0, 3, 5 }` would supply exactly the same first argument as `Time_hms{ 0, 3, 5 }`. A negative duration is instead obtained by negating a non-negative duration, for example:

```
-Time_hms{ 0, 3, 5 }
```

4 Class invariant

A class invariant is a collection of conditions that must remain true for every valid object. The class may temporarily use intermediate values while performing a calculation, but after construction or a modifying operation the stored representation must again satisfy the invariant.

DES-3.1 The stored minute and second components shall each be between 0 and 59, inclusive. An object with `minutes_ == 75` is not a valid normalized representation, even if a human could interpret it as one hour and fifteen minutes.

DES-3.2 The magnitude shall be the duration described by the stored hours, minutes, and seconds. Conceptually, its length in seconds is

$$3600\text{hours_} + 60\text{minutes_} + \text{seconds_}.$$

This expression describes the meaning of the fields; it does *not* prescribe converting objects into a total number of seconds for arithmetic.

DES-3.3 The sign shall be negative exactly when `is_negative_` is true. In particular, zero shall have only one representation:

```
hours_      = 0
minutes_    = 0
seconds_    = 0
is_negative_ = false
```

A representation with all three numerical components zero and `is_negative_ == true` is prohibited. This is sometimes called *negative zero*. It would create unnecessary complications: two objects could represent the same duration but compare as unequal, or output could display `-0:00:00`.

DES-3.4 Every constructor and every operation that changes an object shall preserve the class invariant. In particular, arithmetic that produces zero shall clear the sign.

5 Public construction

The public constructor matches the three non-negative components that users naturally supply:

```
Time_hms( std::uint32_t hours,  
          unsigned int  minutes,  
          unsigned int  seconds );
```

DES-4.1 The constructor shall initialize the hour component from the supplied hours and set `is_negative_` to false.

DES-4.2 It shall reject minute or second arguments outside the range 0 through 59 by throwing `std::invalid_argument`. The implementation tests for values at least 60; since the argument types are unsigned, negative values are not part of their representable ranges.

DES-4.3 Once the minute and second arguments have been validated, they can safely be converted to `std::uint8_t` and stored. The conversion is necessary because the constructor parameters are wider than the corresponding private fields.

For example, `Time_hms{ 37, 15, 42 }` constructs the duration 37:15:42. By contrast, `Time_hms{ 1, 75, 0 }` throws an exception; it does not silently normalize 75 minutes to one hour and fifteen minutes. Keeping input validation separate from arithmetic normalization simplifies the meaning of construction.

The hour argument is a 32-bit unsigned integer. The design does not introduce additional checks for extremely large hours. The exercise concerns ordinary durations, not the boundaries of integer representation.

5.1 Why the constructor does not take signed hours

Suppose we had instead declared the hour parameter as a signed integer. A caller might then write `Time_hms{ -2, 3, 5 }` and expect -2:03:05. However, `Time_hms{ -0, 3, 5 }` would still be indistinguishable from `Time_hms{ 0, 3, 5 }`. This is a limitation of the proposed interface, not of the number of bytes used for the hour field. It is clearer to construct a magnitude and apply unary minus when a negative duration is needed.

6 Private construction

Arithmetic can produce negative durations, although the public constructor creates only non-negative ones. A private constructor permits the implementation to create a signed result directly:

```
Time_hms( std::uint32_t hours,
          std::uint8_t  minutes,
          std::uint8_t  seconds,
          bool          is_negative );
```

DES-5.1 The private constructor shall initialize the component fields and the sign. It assumes that its caller supplies valid, normalized components.

DES-5.2 If all three components are zero, the constructor shall set `is_negative_` to false, regardless of the supplied sign. Thus, it enforces the unique representation of zero.

The current implementation also includes a constructor that accepts a signed 64-bit count of seconds. That constructor decomposes the magnitude into components and records the sign. It is separate from the arithmetic strategy used in this design: *adding and subtracting two Time_hms objects does not convert them to seconds and back*. In decomposing a signed value, the implementation avoids directly negating the smallest signed 64-bit integer. If this constructor were expected to accept durations whose hours exceed the 32-bit field, a bounds check would be appropriate at the conversion to hours. Such extremes are not part of this exercise.

7 Accessors and sign

The public accessors are:

```
bool          negative() const;
std::uint32_t hours()    const;
std::uint8_t  minutes()  const;
std::uint8_t  seconds()  const;
```

DES-6.1 The component accessors shall return the corresponding stored fields. The components are always the magnitude, even for a negative duration. Thus, for `-2:15:30`, the accessors return 2, 15, and 30.

DES-6.2 The `negative()` function shall return `is_negative_`. Because negative zero is prohibited, this Boolean unambiguously describes the sign.

DES-6.3 None of these operations shall modify the object. Their implementations are particularly straightforward because the requested information is already stored separately.

8 Comparing durations

The class provides the six usual comparison operators: `==`, `!=`, `<`, `<=`, `>=`, and `>`.

DES-7.1 Equality shall compare the sign and all three components. This is sufficient because the class invariant ensures that each valid duration has one normalized representation. Inequality can simply negate equality.

DES-7.2 For less-than, first compare signs. Every negative duration is less than every non-negative duration. When signs agree, compare the magnitudes in order: hours first, then minutes, then seconds. The first unequal component determines the larger magnitude.

DES-7.3 If both durations are negative, the ordering of their magnitudes must be reversed. For example, `-3:00:00` is less than `-2:00:00`, even though three hours is a greater magnitude than two hours.

Comparison	Result
<code>2:15:30 < 3:00:00</code>	true
<code>-2:15:30 < 3:00:00</code>	true
<code>-3:00:00 < -2:15:30</code>	true
<code>-0:00:01 < 0:00:00</code>	true
<code>0:00:00 == -0:00:00</code>	negative zero cannot occur

DES-7.4 The remaining ordering operators shall reuse less-than and equality, rather than duplicating the component comparison. For example, `a > b` can be expressed as `b < a`.

Notice that comparison requires more reasoning than with a representation that stores one signed numerical value. The code must account for both the sign and the ordering of the individual fields.

9 Unary plus and minus

Unary plus and minus have the usual arithmetic meanings.

DES-8.1 Unary plus shall return an unchanged copy of the object. Conceptually, its implementation is simply `return *this;`.

DES-8.2 Unary minus shall preserve the three magnitude components and reverse the sign. The private constructor can be used to construct the result.

DES-8.3 Negating zero shall produce zero, not negative zero. The private constructor's normalization makes this case automatic.

For example, `-Time_hms{ 0, 3, 5 }` produces `-0:03:05`, while `-Time_hms{ 0, 0, 0 }` produces `0:00:00`.

10 Binary arithmetic and compound assignment

This is where the apparent simplicity of the representation begins to disappear. Unlike the component accessors, addition and subtraction cannot simply manipulate one field. They must maintain the ranges of minutes and seconds, and they must account for the signs of both operands.

DES-9.1 The binary addition and subtraction operators shall make a copy of the left operand, modify the copy using the corresponding compound-assignment operator, and return it. Conceptually:

```
Time_hms result{ *this };  
return result += other;
```

For binary subtraction, the last statement is `return result -= other;`. This avoids implementing the same arithmetic twice.

DES-9.2 The compound-assignment operators `+=` and `-=` shall perform the actual component arithmetic, modify the left operand, and return `*this` by reference.

10.1 Addition when the signs agree

If both durations are positive, their magnitudes must be added. If both are negative, their magnitudes must also be added, but the result remains negative. Thus the component calculations are identical in both cases.

DES-9.3 Add the seconds first. If their sum is at least 60, carry one minute and retain the remainder as the seconds component. Then add the minutes together with the carry. If that sum is at least 60, carry one hour and retain the remainder as the minutes component. Finally add the hours and any carry.

For example:

```
  1:59:45  
+ 0:00:30  
-----  
  2:00:15
```

The seconds sum to 75, so 15 seconds remain and one minute is carried. The minutes then sum to 60, so zero minutes remain and one hour is carried. The final result is two hours, zero minutes, and fifteen seconds.

The same carrying is needed for `-1:59:45 + (-0:00:30)`, but the result is `-2:00:15`.

DES-9.4 Intermediate second and minute sums shall be held in a wider unsigned integer type, not in the 8-bit component fields. Otherwise, a sum could overflow an 8-bit field before the carrying calculation is completed. For example, the sum of two 8-bit fields is promoted to

`int` by C++ before arithmetic; a local `unsigned int` can safely hold the small result. The implementation need not use an explicit cast for these sums.

DES-9.5 An additional check could be placed at the hour addition if the design were required to detect a result exceeding the range of `std::uint32_t`. The academic implementation intentionally omits this check.

10.2 Addition when the signs differ

Consider $3:20:00 + (-1:45:00)$. Although the operation is called addition, the magnitudes must be subtracted. The result is $1:35:00$. Conversely, $1:45:00 + (-3:20:00)$ produces $-1:35:00$.

DES-9.6 When the signs differ, compound addition shall delegate to compound subtraction using the additive inverse of the right operand. Conceptually:

```
if ( is_negative_ != other.is_negative_ ) {  
    return *this -= -other;  
}
```

The negation makes the signs agree for the delegated subtraction. Thus the two compound operators can share the same sign-handling strategy without endless recursion.

10.3 Subtraction when the signs differ

Subtracting a negative duration adds its magnitude: $2:15:00 - (-0:30:00)$ is $2:45:00$. Likewise, subtracting a positive duration from a negative one makes the negative magnitude larger: $-2:15:00 - 0:30:00$ is $-2:45:00$.

DES-9.7 When the signs differ, compound subtraction shall delegate to compound addition using the additive inverse of the right operand:

```
if ( is_negative_ != other.is_negative_ ) {  
    return *this += -other;  
}
```

Again, negating the right operand makes the signs agree for the delegated addition.

10.4 Subtraction when the signs agree

This is the most complicated case. When both signs agree, the class must subtract magnitudes. Before doing so, it must determine which magnitude is larger, because the result may reverse sign.

DES-9.8 Compare the magnitudes lexicographically: first hours, then minutes if the hours agree, then seconds if both preceding components agree. If the left magnitude is at least the right magnitude, subtract the right magnitude from the left. Otherwise, subtract the left magnitude from the right and reverse the sign of the result.

For example:

```
  2:00:00
- 0:00:01
-----
  1:59:59
```

The left seconds component is too small to subtract one second. We must borrow one minute, but the left minutes component is also zero. Therefore we must borrow one hour, making 60 minutes available, and then borrow one of those minutes to obtain 60 seconds. The resulting components are one hour, 59 minutes, and 59 seconds.

DES-9.9 When subtracting magnitudes, the implementation shall borrow 60 seconds from one minute if the seconds component of the minuend is too small. It shall borrow 60 minutes from one hour if the available minutes, after accounting for any earlier borrowing, are insufficient.

The current implementation uses temporary variables to record whether a minute or an hour has been borrowed. This keeps the arithmetic understandable and prevents an intermediate negative result from being assigned to an unsigned 8-bit field.

DES-9.10 If the right magnitude is larger, the roles of the two magnitudes are reversed for the subtraction, and the sign of the result is reversed. For example, $1:30:00 - 2:00:00$ produces $-0:30:00$.

Negative operands require the same reasoning with the opposite interpretation of the signs. For example:

```
-3:00:00 - (-1:30:00) = -1:30:00
-1:30:00 - (-3:00:00) =  1:30:00
```

DES-9.11 After subtraction, if all three components are zero, `is_negative_` shall be set to false. This is required even when the operands were both negative or when the subtraction reversed their order.

10.5 Why not convert to seconds first?

A tempting alternative is to convert each duration into a single signed number of seconds, add or subtract those numbers, and then convert the result back into hours, minutes, and seconds. That would make the arithmetic substantially easier. However, doing so would obscure the particular trade-off this design is intended to illustrate.

DES-9.12 Addition and subtraction of two `Time_hms` objects shall therefore operate directly on the stored components. The implementation shall not convert either operand into a total number of seconds as an intermediate representation.

The point is not that component-based arithmetic is impossible or incorrect. It is that choosing a representation convenient for construction and display can make other operations more elaborate.

11 Increment and decrement

The class also supports prefix and postfix `++` and `--`. For a duration, these operators change the represented value by exactly one second, not by one hour or one minute.

DES-10.1 Prefix increment shall add a one-second duration to the current object and return the modified object by reference. Prefix decrement shall subtract a one-second duration and likewise return by reference. Both can reuse the existing compound-assignment operators:

```
*this += Time_hms{ 0, 0, 1 };  
return *this;
```

For decrement, the operation is `--` instead of `+=`.

DES-10.2 Postfix increment and decrement shall save the original object, apply the corresponding prefix operation, and return the saved value. The unused `int` parameter distinguishes postfix from prefix in C++.

For example, incrementing `0:59:59` produces `1:00:00`. Decrementing `0:00:00` produces `-0:00:01`. Incrementing `-0:00:01` produces `0:00:00`, with no negative zero. Reusing compound assignment ensures that these cases do not require another carrying and borrowing algorithm.

12 String representation

The public `to_string()` function shall produce the required `[-]H:MM:SS` representation directly from the stored fields.

DES-11.1 If `is_negative_` is true, the string shall begin with a minus sign. Otherwise no sign shall be added.

DES-11.2 The hours shall be represented with at least one decimal digit and without a fixed width. Thus `0:00:00`, `2:05:09`, and `37:15:42` are all valid.

DES-11.3 The minutes and seconds shall each occupy exactly two digits. A leading zero shall be added when the stored component is less than 10.

For example, the fields

```
hours_      = 0
minutes_    = 5
seconds_    = 9
is_negative_ = true
```

shall produce `-0:05:09`.

Because the class invariant prohibits negative zero, the zero duration will always be displayed as `0:00:00`.

13 Stream output

The non-member stream insertion operator shall have the form:

```
std::ostream &operator<<( std::ostream &out,
                          Time_hms const &time );
```

DES-12.1 It shall insert the string returned by `time.to_string()` into the supplied stream and return that stream by reference. This permits expressions such as `std::cout << time << std::endl;`.

Keeping the stream operator separate from the string conversion avoids duplicating the formatting rules. It also lets the same formatting be used in assertions, messages, and output streams.

14 Private interface

The private portion of the class consists of the four stored fields and constructors used to create valid signed results:

```
private:
    std::uint32_t hours_;
    std::uint8_t  minutes_;
    std::uint8_t  seconds_;
    bool          is_negative_;

    Time_hms( std::uint32_t hours,
              std::uint8_t  minutes,
              std::uint8_t  seconds,
              bool          is_negative );
```

The implementation also includes a constructor accepting a signed 64-bit number of seconds, as discussed earlier. It is not necessary for the component arithmetic and is not part of the shared public requirements.

Notice what this private interface does *not* include: there is no `total_seconds()` helper and no `from_seconds()` arithmetic helper. Those functions belonged to an earlier proposed strategy in which arithmetic was performed on converted values. The present implementation uses carrying, borrowing, and sign handling directly.

15 Storage considerations

The minute and second components each require values from 0 to 59, so an 8-bit unsigned type is sufficient for each. The sign needs only two states, so a Boolean is suitable. The hour component uses a 32-bit unsigned type and is not restricted to the hours of a single day.

DES-13.1 The sizes of the individual fields do not necessarily add up to the size of the complete object. The compiler may insert padding between or after fields to satisfy alignment requirements. Consequently, `sizeof(Time_hms)` is determined partly by the target platform and the compiler's layout rules.

For example, four bytes for hours, one for minutes, one for seconds, and one for the sign account for seven bytes of data, but a typical implementation may occupy eight bytes after padding. The exact object size should be observed with `sizeof`, not assumed from the sum of field widths.

This design is not necessarily more compact than storing one signed 64-bit integer. Its advantage is the directness of its component representation, not a guaranteed reduction in storage.

16 Design consequences

The representation closely matches the way people write and speak about durations:

```
[-]hours:minutes:seconds
```

Construction of a non-negative duration is straightforward. The public constructor validates minutes and seconds, then stores the components. The accessors return the corresponding fields. Formatting a duration also follows naturally from those fields.

But negative durations require a separate sign. That sign must be considered in equality, ordering, unary minus, addition, and subtraction. The representation must also prohibit negative zero so that a single duration cannot have two stored forms.

Most importantly, addition and subtraction require careful treatment of carries, borrows, magnitude comparisons, and sign changes. Adding two positive durations is already more involved than adding two integers. Subtracting two signed durations is more complicated still, especially when the magnitudes are reversed or the result crosses zero.

We have deliberately retained this component-based arithmetic instead of converting the objects into seconds. The implementation therefore exposes the full consequences of the representation chosen at the beginning of the design.

The subsequent `Time_s` design will investigate the opposite trade-off. By storing the entire duration as a single signed number of seconds, arithmetic and comparison become much simpler. The cost is that hours, minutes, seconds, and formatted output must be derived from that internal value. Both designs satisfy the same public requirements; the difference lies in where the complexity occurs.