# `CTime` User's Manual

by

Software Industry & General Hardware
(sales@s-i-g-h.com)
(document reference: CTime20090323150933PDT)

# Table of Contents

# Legalese

## *License*



Software License Agreement

By using this software you are agreeing to be bound by this license agreement.

If you do not agree to be bound by this license, do not use this software.

Either delete software package completely or return the physical package,

including the distribution media in the original packaging, unopened, to the manufacturer at:

> Software Industry & General Hardware
> ATTN: PKG RETURN
> 23125 Crooked Arrow Drive
> Wildomar, CA USA 92595

This package may not be used or redistributed as part of a commercial release under this license. Contact the owner of the software (S.I.G.H.) for fee information and commercial use licenses. (Contact information: email to **`sales@s-i-g-h.com`**)

Non-commercial use license:

You may use and/or redistribute the java package included in this distribution as is or as part of another software package owned by you as long as the following conditions are met:

1. The new package cannot alter the functionality or use of the existing package.
2. The new package must allow the user access to the “version()” method of the existing package.
3. The new package must allow the user access to the “license()” method of the existing package.
4. This Software License Agreement, in its entirety, must be included in your documentation.
5. The following notice must be included in your documentation.

   “This software is COPYRIGHT © 2009 by Software Industry & General Hardware.
   ALL RIGHTS reserved.

   This software package includes and uses the **`CTime.jar`** package developed by:
   Software Industry & General Hardware (S.I.G.H.).
   For more information on this or any other software package from S.I.G.H.,
   email info@s-i-g-h.com

   This software may not be included in a commercial package.”

## *Commercial Use Information*

For information on commercial use of this software contact Software Industry & General Hardware at: sales@s-i-g-h.com, or write for information to:

Software Industry & General Hardware
23125 Crooked Arrow Drive
Wildomar, CA USA 92595

## *Non-commercial Use Information*

Non-commercial use is governed by the License above.

## *What does all this mean?*

If you are using the software for non-commercial use then have fun learning programming, writing that program to help others or anything you are not intending to sell for profit. Have fun and enjoy the software. Use it to your hearts content.

If you are planning on using this software as part of a package you are going to sell; you may not do so without contacting the owner of the software and obtaining a license from the Software Owner.

If you are planning on distributing this software as part of a non-profit or not-for-profit venture you may do so with impunity.

If you are going to distribute this software as part of a software collection you are not charging for (except for a nominal fee for the media and/or shipping [no more than $10(US)]) you may do so.

If you are going to distribute this software as part of a software collection you are going to sell you may not do so.

## Williams' Notation

This notation was invented by the author to help solve some of the drudgery of note taking in Computer Science classes. The problem this notation solves is one familiar to most computer science students and practitioners. Often one must describe something that is a place-holder. Once very good example is describing something in BNF (Backus-Normal-Form if you have something against Mr. Naur; or Backus-Naur Form if you don't have anything against Mr. Naur [for a more complete discussion please see this Wikipedia entry). Or another is where the author wants to indicate that a file-name would be inserted into a particular position in a command line. For example:

"The student should type: **`cat <insert_a_file_name_here>`**".

The problem isn't the **`insert_a_file_name_here`** clause but rather telling the student NOT to type the left and right angle-brackets. Williams notation is to simply note the non-entered entities at the end of the sentence, enclosed by some meta-character of its own. So the above sentence would become:

"The student should type: **cat <insert_a_file_name_here>. `/< >/`**"

The notation **`/< >/`** indicates the angle brackets are not typed and do not belong to the clause **insert_a_file_name_here**. You can also indicate that a punctuation or phrase is simply intended for emphasis and not to be considered part of the sentence itself. For example:

"The student should type **`cat <insert_a_file_name_here>`**". **`/< > " "/`**

The space between the characters is intentional so that the characters are more easily spotted. This is called "visual acuity" and is something that the author wishes was more deeply considered by computer program language authors. Sometimes a brace and a parenthesis are indistinguishable in print or on a fading screen with glare: {(

Lastly in the notation the meta-character does not have to be a slash[1]. It may be any character you wish to use. Just use it in pairs.

## Pseudo BNF

As mentioned in the "Williams' Notation" section BNF (Backus-Naur Form) gives us an easy grammatical ordering descriptive notation. Arcane to the un-initiated pseudo-BNF is simply a relaxation of the rules of BNF. <identifier> declares a descriptive name. A literal symbol is shown by itself and things enclosed in square brackets are optional. And a subscripted logical-and is often used to indicate a space: ∧.The pseudo-BNF of [<name_of_day_of_week>∧]<month>∧<day>,∧<year> should be familiar as a description of March 25, 2009 or optionally Wednesday March 25, 2009.

Pseudo-BNF is used throughout this document.

1 Slash is the character "/". /" "/ More properly known as a virgule. It is NOT the reverse-virgule or "\". /" "/ This is another lunacy perpetrated by MS bigots. A / is a slash and a \ is a back-slash.

## Notes And Abbreviations

KNR --- “The c Programming Language” reference, by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice Hall.

Wikipedia --- http://www.wikipedia.com encyclopedia

ISO --- The Internal Standards Organization

## Overview

**`CTime`** is a Java equivalent of most of the c programming language functionality found in the strftime() method. In general the rules are governed first by **`strftime`** in KNR. Basic functionality can be discerned from the KNR reference. However there are some additions and changes outlined by ISO 8601. This manual assumes the reader is familiar and comfortable with the c programming language statements **`printf`**, **`sprintf`**, **`strftime()`** and their use.

## KNR Inclusions

In the c programming language the statement “**strftime**” consists of the following general syntax: **`strftime ("<format_place_holder>...", <variable_identifier>...);`** /" "**...**/ Where **...** means zero or more occurrences. In general a **<format_place_holder>** has the following syntax: **"*%type*"** **`CTime`** accepts and processes the types: **`a`**, **`A`**, **`b`**, **`B`**, **`c`**, **`d`**, **`H`**, **`I`**, **`j`**, **`m`**, **`M`**, **`p`**, **`S`**, **`U`**, **`w`**, **`W`**, **`x`**, **`X`**, **`y`**, **`Y`** and **`z`**.

## ISO Inclusions

**`CTime`** follows the ISO description of **`strftime`** (see: ISO definition of format strings for strftime()). These include additions such as **`%G`**, **`%V`** and **`%n`**, **`%t`** and others. Specifically the format strings are:

| Format String | Description |
|---|---|
| **`%a`** | Replaced by the locale's abbreviated weekday name. [ tm_wday] |
| **`%A`** | Replaced by the locale's full weekday name. [ tm_wday] |
| **`%b`** | Replaced by the locale's abbreviated month name. [ tm_mon] |
| **`%B`** | Replaced by the locale's full month name. [ tm_mon] |
| **`%c`** | Replaced by the locale's appropriate date and time representation. (See the Base Definitions volume of IEEE Std 1003.1-2001, <time.h>.) |
| **`%C`** | Replaced by the year divided by 100 and truncated to an integer, as a decimal number [00,99]. [ tm_year] |

| Format String | Description |
|---|---|
| %d | Replaced by the day of the month as a decimal number [01,31]. [ tm_mday] |
| %D | Equivalent to %m / %d / %y. [ tm_mon, tm_mday, tm_year] |
| %e | Replaced by the day of the month as a decimal number [1,31]; a single digit is preceded by a space. [ tm_mday] |
| %F | Equivalent to %Y - %m - %d (the ISO 8601:2000 standard date format). [ tm_year, tm_mon, tm_mday] |
| %g | Replaced by the last 2 digits of the week-based year (see below) as a decimal number [00,99]. [ tm_year, tm_wday, tm_yday] |
| %G | Replaced by the week-based year (see below) as a decimal number (for example, 1977). [ tm_year, tm_wday, tm_yday] |
| %h | Equivalent to %b [ tm_mon] |
| %H | Replaced by the hour (24-hour clock) as a decimal number [00,23]. [ tm_hour] |
| %I | Replaced by the hour (12-hour clock) as a decimal number [01,12]. [ tm_hour] |
| %j | Replaced by the day of the year as a decimal number [001,366]. [ tm_yday] |
| %m | Replaced by the month as a decimal number [01,12]. [ tm_mon] |
| %M | Replaced by the minute as a decimal number [00,59]. [ tm_min] |
| %n | Replaced by a <newline>. |
| %p | Replaced by the locale's equivalent of either a.m. or p.m. [ tm_hour] |
| %r | Replaced by the time in a.m. and p.m. notation; [CX] in the POSIX locale this shall be equivalent to %I : %M : %S %p. [ tm_hour, tm_min, tm_sec] |
| %R | Replaced by the time in 24-hour notation ( %H : %M ). [ tm_hour, tm_min] |
| %S | Replaced by the second as a decimal number [00,60]. [ tm_sec] |
| %t | Replaced by a <tab>. |
| %T | Replaced by the time ( %H : %M : %S ). [ tm_hour, tm_min, tm_sec] |
| %u | Replaced by the weekday as a decimal number [1,7], with 1 representing Monday. [ tm_wday] |

| Format String | Description |
|---|---|
| %U | Replaced by the week number of the year as a decimal number [00,53]. The first Sunday of January is the first day of week 1; days in the new year before this are in week 0. [ tm_year, tm_wday, tm_yday] |
| %V | Replaced by the week number of the year (Monday as the first day of the week) as a decimal number [01,53]. If the week containing 1 January has four or more days in the new year, then it is considered week 1. Otherwise, it is the last week of the previous year, and the next week is week 1. Both January 4th and the first Thursday of January are always in week 1. [ tm_year, tm_wday, tm_yday] |
| %w | Replaced by the weekday as a decimal number [0,6], with 0 representing Sunday. [ tm_wday] |
| %W | Replaced by the week number of the year as a decimal number [00,53]. The first Monday of January is the first day of week 1; days in the new year before this are in week 0. [ tm_year, tm_wday, tm_yday] |
| %x | Replaced by the locale's appropriate date representation. (See the Base Definitions volume of IEEE Std 1003.1-2001, <time.h>.) |
| %X | Replaced by the locale's appropriate time representation. (See the Base Definitions volume of IEEE Std 1003.1-2001, <time.h>.) |
| %y | Replaced by the last two digits of the year as a decimal number [00,99]. [ tm_year] |
| %Y | Replaced by the year as a decimal number (for example, 1997). [ tm_year] |
| %z | Replaced by the offset from UTC in the ISO 8601:2000 standard format ( +hhmm or -hhmm ), or by no characters if no timezone is determinable. For example, "-0430" means 4 hours 30 minutes behind UTC (west of Greenwich). [CX] If tm_isdst is zero, the standard time offset is used. If tm_isdst is greater than zero, the daylight savings time offset is used. If tm_isdst is negative, no characters are returned. [ tm_isdst] |
| %Z | Replaced by the timezone name or abbreviation, or by no bytes if no timezone information exists. [ tm_isdst] |
| %% | Replaced by a %. |

## ISO Time Standards

Two surprising things to North American citizens will be that the week begins on Monday and the order of the date display is <year>-<month>-<day>. The date ordering just makes sense since a date

kept in this order is always sorted. Using the traditional ordering of <month> <day>, <year> makes about as much sense as displaying time as <minute> <second> <hours>; that is it doesn't make any sense.

ISO 8601 suggests that the date should be written as <year>–<month>–<day> or as <year><day_of_year>. German users may be use to the dot-notation separation of dates as <year>.<month>.<day> but at the end of a sentence the trailing period could be confusing. DIN 5008 changed this notation in 1995 to reflect the ISO 8601 standard.

For the North American user you should be aware that IEEE date standard reflects ISO 8601.

ISO 8601 suggests that time be written as <hour>:<minutes>:<seconds>[.<fractions_of_a_second>].

ISO 8601 suggests further that if the date and time are to be written together on the same line they should be separated by a latin capital letter T and without the separators, as in **`19951231T235959`**.

| Format String | Example Output |
|---|---|
| `%Y-%m-%d` | `03/24/09` |
| `%Y-%j` | `2009-0` |
| `%G-W%V-%u` | `2009-W12-3` |
| `%H:%M:%S` | `11:59:59 PM` |
| `%Y%m%dT%H%M%S` | `20090324T113648` |
| `%Y%jT%H%M%S` | `2009084T113648` |

For a more complete reference please see ISO 8601.

# Proprietary Inclusions

## *TimeFormatString Class*

To allow the **`%t`** type to work consistently with **`Ctime`** an additional class was designed. This class associates a format string with a time in milliseconds. There are “setter” methods for both the format string and the millisecond value to allow one to use several time values applied to a single format string or several format strings applied to a single time value. The format string and time are initially set from the constructor or you can use a default constructor to set the format string to the default value of **`%c`** and the time to the time at the instant of creation of the object (basically today's date). There are “getter” methods for both the format string and the time value. The constructor has the form:

```
public TimeFormatString() throws NullPointerException
public TimeFormatString ( String aStrformatString, long aLongValue )
       throws NullPointerException
```

The two parameters are:

1. aStrformatString – a String value containing the strftime format string as described in KNR.

2. ALongValue --- a long time value representing the time in milliseconds since (positive values) or before (negative values) 1970 January 1.

## *Methods:*

**`TimeFormatString()`**
Creates an instance of a **`TimeFormatString`** with the default format string of **`%c`** and a default millisecond value of the time of object creation.

**`TimeFormatString(String, long)`**
Creates an instance of a **`TimeFormatString`** with the user's values for the format string and the time in milliseconds.

**`static String version()`**
Returns the current version string for **`TimeFormatString.`**

**`String getFormatString()`**
Returns the current format string (which cannot be null).

**`long getTimeInMilliseconds()`**
Returns the current millisecond offset being used.

**`void setFormatStringTo(String)`**
Sets the formatString to the one provided by the user. Cannot be null.

**`void setTimeInMilliseconds(long)`**
Sets the current millisecond offset being used to the one provided by the user. Based on Java base date.

## *Overview:*

There are 24 hours in a day, 60 minutes in an hour and 60 seconds in a minute and 1,000 milliseconds in a second. This means that there are 24 * 60 * 60 * 1000 milliseconds in a day or 86,400,000 milliseconds per day.

Keep in mind that Java long values are 64-bits long.

The following code should print the current date:

```
  public boolean testTodayTime() {
    try {
        System.out.println ( "testTodayTime starts" );
        //                   J  F   M   A   M    J    J    A    S    O    N    D
        int[] days2Month = { 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334 };
        CTime ct = new Ctime();
        int currentYear  = Integer.valueOf ( ct.yearInDecimalWithCentury() );

        int currentMonth = Integer.valueOf ( ct.monthAsANumber() );
        int currentDay   = Integer.valueOf ( ct.dayOfTheMonthAsNumber() );

        // The following formula should calculate the current date:
        long totalDaysSinceEpoch1970 =
          Long.valueOf ( Math.round ( Math.floor ( ( currentYear - 1970 ) * 365.25 ) ) );
        totalDaysSinceEpoch1970 += days2Month [ currentMonth - 1 ] + currentDay + 1;
        totalDaysSinceEpoch1970 *= 86400000L;
        if ( CTime.isLeapYear() ) {
```

```
            if ( ( 2 < currentMonth ) || ( ( 2 == currentMonth ) && ( 28 < currentDay ) ) )
                totalDaysSinceEpoch1970 += 1;
        }
        TimeFormatString tfs = new TimeFormatString ( "%c", totalDaysSinceEpoch1970 );
        System.out.println ( CTime.strftime ( tfs ) );
        // or perhaps
        Object[] timeValue = new Object [ 1 ];
        timeValue [ 0 ] = (Object) tfs;
        System.out.println ( Csprintf.sprintf ( "Today is: %t", timeValue ) );
    } catch ( CsprintfExceptions csfE ) {
        csfE.printStackTrace ();
        return false; // error return
    } catch ( NullPointerException npE ) {
        npE.printStackTrace ();
        return false; // error return
    }
    System.out.println ( "testTodayTime complete" );
    return true;
  }
```

Of course, so too will either of the **`System.out.println()`** statements that follow:

```
        try {
            System.out.println (
                    CTime.strftime ( "%c", Calendar.getInstance().getTimeInMillis() ) );
            TimeFormatString tfs = new TimeFormatString();
            System.out.println ( Csprintf.sprintf ( "%t", tfs ) );
        } catch ( NullPointerException npE ) {
            npE.printStackTrace();
            fail();
        } catch ( CsprintfExceptions csfE ) {
            csfE.printStackTrace();
            fail();
        }
```

## Types

**`CTime`** does not change the ISO **`strftime()`** described types. This means the **`CTime`** user has a richer suite of types than are described in KNR.

## Methods

**`Ctime()`**

Use to instantiate an instance of **`CTime()`**. This method establishes the local locale grouping characters and grouping size.

**`doSprintf(String, Object[])`**

Non static call used with an instantiated object. String is the c programming language equivalent format string. **`Object[]`** is an vector of object values to apply to the format string.

**`set3DigitSeparatorString(String)`**

This method allows the programmer to set a new separator string. It returns the old separator string value so you can save and restore the previous separator string.

**`set3DigitSeparatorString(char)`**

Same as above except allowing the programmer the convenience of using a single character instead of a string.

**`setFractionalSeparatorChar(String)`**
Allows the programmer to specify the fractional separator string in floating point numbers. North America typically uses a period for this character and Europe typically uses a comma.

**`setFractionalSeparatorChar(char)`**
Same as above except allowing the programmer the convenience of using a single character instead of a string.

**`sprintf(String, Object)`**
Static call where the first String is the c programming language equivalent format string. Object is a single object value to apply to the format string.

**`sprintf(String, Object, String)`**
Same as above except that the second String is the separator string to apply to this static call only.

**`sprintf(String, Object, String, int)`**
Same as above except that the int is the number of characters to group together instead of using the local locale default for this static call only.

**`sprintf(String, Object, int)`**
Static call where the first String is the c programming language equivalent format string. Object is a single object value to apply to the format string and int is the number of characters to group together instead of using the local locale default for this static call only.

**`sprintf(String, Object[])`**
Static call used with an instantiated object. String is the c programming language equivalent format string. **`Object[]`** is a vector of object values to apply to the format string.

**`sprintf(String, Object[], String, int)`**
Same as above except the second String is the separator string and the int is the number of characters to group together instead of using the local locale default for this static call only.

**`sprintf(String, Object[], String)`**
Same as above except the second String is the separator string.

**`sprintf(String, Object[], int)`**
Same as above except the int is the number of characters to group together instead of using the local locale default for this static call only.

**`setNumberOfDigitsToSeparate(int)`**
For non-static calls sets the number of characters to group together before a separator string is applied. Returns the previous value.

**`GetNumberOfDigitsToSeparate()`**
Returns the current number of digits that are grouped together before a separator string is inserted into the numeric string.

**`Get3DigitSeparatorString()`**
Returns the current grouping separator string.

**`GetFractionalSeparatorString()`**
Returns the current separator string for the fractional separator.

**`set3DigitSeparatorChar(String)`**

Sets the grouping separator string and returns the current grouping separator string.

**setFractionalSeparatorChar(String)**

Sets the fractional separator string and returns the current separator string.

## Code Examples

CTime()

```
Object[] testValues = new Object [ 3 ];
CTime csf = new CTime();
testValues [ 0 ] = 512;
testValues [ 1 ] = 511;
testValues [ 2 ] = 256;

System.out.println ( csf.doSprintf ( "0%o 0%o 0%o", testValues ) );
System.out.println ( csf.doSprintf ( "%#o %#o %#o", testValues ) );

Object[] anotherSpecialTestValue = new Object [ 1 ];
aTestValue [ 0 ] = new Short ( (short) -1 );
System.out.println ( CTime.sprintf ( "%u", aTestValue ) );

specialTestValues [ 0 ] = new Float  ( 0.0 );
specialTestValues [ 1 ] = new Double ( 0.0 );
System.out.println ( csf.doSprintf ( "zero float %f double %f", testValues ) );
```

```
Will output:
01000 0777 0400
01000 0777 0400
65535
zero float 0.000000 double 0.000000
```

doSprintf ( String, Object[] )

```
Object[] testValues = new Object [ 3 ];
CTime csf = new CTime();
testValues [ 0 ] = 512;
testValues [ 1 ] = 511;
testValues [ 2 ] = 256;

System.out.println ( csf.doSprintf ( "0%o 0%o 0%o", testValues ) );
System.out.println ( csf.doSprintf ( "%#o %#o %#o", testValues ) );

Object[] aTestValue = new Object [ 1 ];
aTestValue [ 0 ] = new Short ( (short) -1 );
System.out.println ( CTime.sprintf ( "%u", aTestValue ) );

testValues [ 0 ] = new Float  ( 0.0 );
testValues [ 1 ] = new Double ( 0.0 );
System.out.println ( csf.doSprintf ( "zero float %f double %f", testValues ) );
```

(output shown on following page)

Will output:

```
01000 0777 0400
01000 0777 0400
65535
zero float 0.000000 double 0.000000
```

set3DigitSeparatorString ( String )

```
CTime csf = new CTime();
String oldcsf.set3DigitSeparatorString ( "<*>"  );
System.out.println ( csf.sprintf ( "%,Lu", 1234567890 ) );
```

Will output (for Babylon 5 fans):

```
1<*>234<*>567<*>890
```

set3DigitSeparatorString ( char )

setFractionalSeparatorChar ( String )

setFractionalSeparatorChar ( char )

sprintf ( String, Object )

```
System.out.println ( CTime.sprintf ( "%,Lu", 1234567890L  ) );
System.out.println ( CTime.sprintf ( "%,Li", -1234567890L ) );
System.out.println ( CTime.sprintf ( "%10g", new Double ( -0.156400000007 ) ) );
System.out.println ( CTime.sprintf ( "%10g", new Double (  0.156400000007 ) ) );
System.out.println ( CTime.sprintf ( "%+10g", new Double ( 0.156400000007 ) ) );

System.out.println ( CTime.sprintf ( "%12.2e", new Double  ( -0.15640000000007 ) ) );
System.out.println ( CTime.sprintf ( "%12.2E", new Double  ( -0.15640000000007 ) ) );
System.out.println ( CTime.sprintf ( "%12.2e", new Double  (  0.15640000000007 ) ) );
System.out.println ( CTime.sprintf ( "%12.2E", new Double  (  0.15640000000007 ) ) );
System.out.println ( CTime.sprintf ( "%12.3e", new Double  ( -0.15640000000007 ) ) );
System.out.println ( CTime.sprintf ( "%12.3E", new Double  ( -0.15640000000007 ) ) );
System.out.println ( CTime.sprintf ( "%12.3e", new Double  (  0.15640000000007 ) ) );
System.out.println ( CTime.sprintf ( "%12.3E", new Double  (  0.15640000000007 ) ) );
System.out.println ( CTime.sprintf ( "%12.3e", new Integer (  1                ) ) );
System.out.println ( CTime.sprintf ( "%12.2e", new Double  (  0.0              ) ) );
System.out.println ( CTime.sprintf ( "%12.3E", new Double  (  0.0              ) ) );
        // Now force a plus sign
System.out.println ( CTime.sprintf ( "%+12.2e", new Double (  0.15640000000007 ) ) );
System.out.println ( CTime.sprintf ( "%+12.2E", new Double (  0.15640000000007 ) ) );
System.out.println ( CTime.sprintf ( "%+12.3e", new Double (  0.15640000000007 ) ) );
System.out.println ( CTime.sprintf ( "%+12.3E", new Double (  0.15640000000007 ) ) );
System.out.println ( CTime.sprintf ( "%+12.2e", new Double (  0.0              ) ) );
System.out.println ( CTime.sprintf ( "%+12.3E", new Double (  0.0              ) ) );
System.out.println ( "% can stand alone Yes they can!"
                , sprintf ( "%% can stand alone %s", new String ( "Yes they can!") ) );
```

Will output:

```
1,234,567,890
-1,234,567,890
   -0.1564
    0.1564
   +0.1564
```

```
   -1.56e-01
   -1.56E-01
    1.56e-01
    1.56E-01
  -1.564e-01
  -1.564E-01
   1.564e-01
   1.564E-01
NSF
    0.00e+00
   0.000E+00
   +1.56E-01
   +1.56e-01
  +1.564e-01
  +1.564E-01
   +0.00e+00
  +0.000E+00
% can stand alone Yes they can!%% can stand alone Yes they can!
```

Note: The `NSF` is generated by the line containing: `"%12.3e", new Integer ( 1 ). Integer` is not of base type double.

sprintf ( String, Object, String )

```
System.out.println ( CTime.sprintf ( "%#,0o", -1L, " " ) );
System.out.println ( CTime.sprintf ( "%012,o", 4269801473L, " " ) );
System.out.println ( CTime.sprintf ( "%,0o", 2814749767106561L, "." ) );
```

Will output:

```
0o1 777 777 777 777 777 777 777
037 640 000 001
120.000.000.000.000.001
```

sprintf(String, Object, String, int)

```
System.out.println ( CTime.sprintf ( "0x%0,x", -1L, " ", 4 ) );
System.out.println ( CTime.sprintf ( "%0,x", 4269801473L, " ", 4 ) );
System.out.println ( CTime.sprintf ( "%,0x", 2814749767106561L, ":", 4 ) );
```

Will output:

```
0xffff ffff ffff ffff
fe80 0001
0010:0000:0000:0001
```

sprintf(String, Object, int)

```
System.out.println ( CTime.sprintf ( "%,0x", 2814749767106561L, 4 ) );
```

Will output:

```
0010,0000,0000,0001
```

sprintf(String, Object[])

```
Object[] anObject = new Object [ 1 ];
anObject [ 0 ] = 1L;
System.out.println ( sprintf ( "Example of SIGH's sprintf: %06ld", anObject ) );
```

Will output:

```
Example of SIGH's sprintf: 000001
```

sprintf(String, Object[], String, int)

sprintf(String, Object[], String)

sprintf(String, Object[], int)

setNumberOfDigitsToSeparate(int)

```
CTime csf = new CTime();
int oldValue = csf.getNumberOfDigitsToSeparate();
csf.setNumberOfDigitsToSeparate ( 4 );
csf.set3DigitSeparatorChar ( ' ' );
System.out.println ( "Previous number of digits to separate: " + oldValue );
tempObj [ 0 ] = new Long ( -1L );
System.out.println ( csf.doSprintf ( "%#,x", tempObj ) );
tempObj [ 0 ] = new Short ( (short) 23130 );
System.out.println ( csf.doSprintf ( "%0,x", tempObj ) );
System.out.println ( "Previous number of digits to separate: " +
                     csf.setNumberOfDigitsToSeparate ( oldValue ) );
csf.setLocalLocaleForBothSeparators();
```

Will output:

```
Previous number of digits to separate: 3
0xffff ffff ffff ffff
5a5a
Previous number of digits to separate: 4
```

getNumberOfDigitsToSeparate()

```
int widget = CTime.getNumberOfDigitsToSeparate();
```

get3DigitSeparatorString()

getFractionalSeparatorString()

set3DigitSeparatorChar(String)

setFractionalSeparatorChar(String)

## C vs Java And Types

Java has the following integral types and values:

| | |
|---|---|
| `byte` | -128 to 127, inclusive |
| | 8-bits |
| `short` | -32768 to 32767, inclusive |
| | 16-bits |
| `int` | -2147483648 to 2147483647, inclusive |
| | 32-bits |
| `long` | -9223372036854775808 to 9223372036854775807, inclusive |
| | 64-bits |
| `char` | `'\u0000'` to `'\uffff'` inclusive, that is, from 0 to 65535 |
| | 16-bits |

Java has the following floating-point types and values for float and double:

| Parameter | float | float-extended-exponent | double | double-extended-exponent |
|---|---|---|---|---|
| N | 24 | 24 | 53 | 53 |
| K | 8 | ≥11 | 11 | ≥15 |
| $E_{max}$ | 127 | ≥+1023 | 1023 | ≥+16383 |
| $E_{min}$ | -126 | ≤-1022 | -1022 | ≤-16382 |
| | | | | |

If you are a c programmer you may be struck by the fact that there is no unsigned (except for char). This is correct. So **`CTime`** must “fake” an unsigned value when requested. C99 has long long types and more. This is true. Java does not. How you get your c data into a Java value via JNI is up to you. Once you do you may apply the attributes of unsigned, long and short as you wish. **`CTime`** will do its best to evaluate this according to Java types and return a string representing what was requested. There are of course caveats. The best answer is to try a short example of what you are trying to do and see what the output looks like.

# Caveats

## *%g*

The formatting string "**%g**" is unusual to say the least. /" "/ KNR indicates that **%g** differs slightly from fixed-point notation in that insignificant zeroes to the right of the decimal point are not included and that the decimal point is not included on whole numbers. All of this changes if you use the alternate form flag of "**#**". /" "/

KNR also indicates that whether **%f** equivalent or the **%e** equivalent type is used should be determined by the esthetics of the display. **CTime** does a fairly reasonable job of determining the changeover point. There is no "fuzzy-logic" control of the changeover point.

# Common Misconceptions

The "**l**" (letter lowercase el) attribute modifies the format string so that the value is displayed as a long value. It does not change the value itself. /" "/ This doesn't matter in the c programming language since a long and an int are the same length (32-bits). So the following code in c would produce the following output:

```
#include <stdio.h>
#include <stdlib.h>

int main ( int argc, char** argv )
{
  long widget = -1L;
  int  wodget = -1;

  printf ( "%u %u\n", sizeof(widget), sizeof(wodget) );
  printf ( "%u %u\n", widget, wodget );
  return 0;
}
4 4
4294967295 4294967295
```

Both a long and an int are 4-bytes long (32-bits) and each has the same unsigned maximum value of 4,294,967,295.

Java is different. An int is 32-bits and a long is 64-bits long. **CTime** follows the Java values and the c programming language's philosophy of the "**l**" modifier when applied to values. /" "/

The following code yields the following output:

```
public class example0 {
    public example0() {
    }
    public static void main ( String[] argv ) {
        try {
           System.out.println ( CTime.sprintf ( "%lu", new Byte    ( (byte)  -1 ) ) );
           System.out.println ( CTime.sprintf ( "%lu", new Short   ( (short) -1 ) ) );
           System.out.println ( CTime.sprintf ( "%lu", new Integer ( (int)   -1 ) ) );
           System.out.println ( CTime.sprintf ( "%lu", new Long    ( (long)  -1 ) ) );
        } catch ( CTimeExceptions csfE ) {
```

```
            csfE.printStackTrace();
        }
    }
}

255
65535
4294967295
18446744073709551615
```

The long modifier should prefix the u flag not suffix the u flag. For example "**`%ul`**" as a format string will produce <value>l where as "**`%lu`**" will produce <value_as_unsigned_long>. /" " < >/

The best thing to do is to TRY the format string and see what you get as output. If you disagree with the results then see what the gcc compiler or g++ compiler output is. You may be surprised. If you are convinced your display is correct then see the Support chapter and send it in for an evaluation.

## Exceptions

**`CTime`** throws the following exceptions:

- **`CTimeExceptions`**
  - Consisting of the following possible exception characteristics:
    - UNKNOWN_EXCEPTION
    - INCORRECT_FORMAT_STRING
    - INCORRECT_VALUE_FOR_FORMAT_STRING
    - INCORRECT_RETURN_VALUE_TYPE
    - INCORRECT_VALUES_FOR_ASTERISK_FORMAT
- ArrayOutOfBoundsException
- NullPointerException

If you encounter one of these please send the stack trace to: info@s-i-g-h.com along with the items listed in the Support chapter.

## Errors

If **`CTime`** cannot parse the format string or if the value is not represented in the base type of the format string **`CTime`** returns **`NSF`**. **`NSF`** stands for "No Such Formatting". Since numbers are converted to strings and the strings are then manipulated there is a possibility that **`CTime`** could throw a Java ArrayOutOfBoundsException; please see the Support chapter if you experience this.

## Support

If you believe that **`CTime`** is displaying an incorrect value send the following to info@s-i-g-h.com:

- The actual numeric value(s) you used with **`CTime`**
- The format string value you used with **`CTime`**
- The actual code line of the call to **`CTime`** (just one or two lines please we don't need or want the entire project).
- What you believe **`CTime`** should have displayed as the result of the formatting string.

You should get a response within 48 hours.

## Recourse

Software Industry & General Hardware attempts to provide the most bug-free software it can, no project is completely bug free or feature complete. We will listen to and collect suggestions but implementation of the those suggestions is solely at the discretion of Software Industry & General Hardware.

This software is supplied “AS IS”. You use this software at your own risk.

Software Industry & General Hardware is not responsible for perceived or actual loss of data, time nor anything else whether to you or your customers/users.

There is no recourse.

The commercial version is governed by a different set of rules.