Common Calendar Binary Format

Common Calendar Timestamp System

Brooks Harris Version 3 2024-04-23

Table of Contents

Notation

"YMDhms" is shorthand for year-month-day hour:minute:second representation.

ISO 8601 representation is supplemented with suffixes (UTC) and (TAI), for example

1970-01-01 00:00:10 (TAI) = 1970-01-01T00:00:00 (UTC).

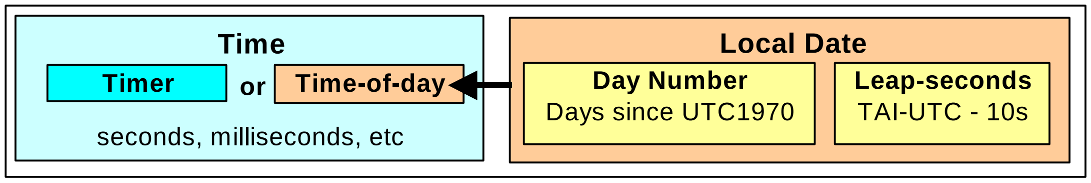
"UTC1970" is shorthand for 1970-01-01 00:00:10 (TAI) = 1970-01-01T00:00:00 (UTC).

1 Introduction

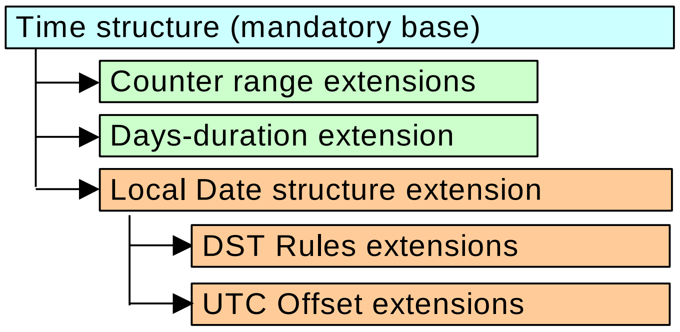
Common Calendar Binary Format (CBF) provides a compact binary data format to facilitate fast transfer, efficient interchange, and economical storage. The CBF design is intended for binary systems, protocols, and languages, such as embedded systems, time dissemination, and c/c++ , while offering compatible expression on other platforms and formats such as Java and XML.

CBF acts in concert with Common Calendar Character Format (CCF) to provide comprehensive description of local date and time with symmetrical conversion between the binary and character based YMDhms representations.

The CBF binary timestamp is a variable length compound counter with associated metadata. It is made up of the mandatory Time structure base structure that provides the anchor for any configuration of CBF, and optional extensions, including the important Local Date structure.



The variable size design provides a single syntax to construct binary timestamps to accommodate use as a simple 24 hour timer (using only the mandatory Time structure), or a full local date and time-of-day timestamp (with the Local Date extension). This flexibility minimizes the total size of the timestamp for each purpose.



The mandatory Time structure contains a 35-bit unsigned counter and sign bit (signed 36-bit). The range may be increased to 48, 64, or 80 bits using the counter extensions. It can represent resolutions from seconds to picoseconds depending on the rate indicator. Used by itself it is a simple 24-hour timer.

The Local Date structure extension is added to provide full local date and time-of-day representation. Its calendar date uses the TAI-UTC API date data structure which includes the day-number-since-UTC1970 and TAI-UTC (leap-seconds). Its time zone metadata uses elements derived from the Common Calendar Time Zone API.

Additional leap-second metadata provides sufficient information for interpretation by a simple receiving application that has no leap-second or local time information available, and facilitates conversion to the CCF YMDhms representation.

When Local Date is used the values of the base Time structure are interpreted as time-of-day. Together, the compound counter, made up of the time-of-day from Time, and the day-number and leap-seconds from Local Date, represent the local date and time as an absolute seconds-since-UTC1970 value. For example, at a seconds resolution (rate), the formula is:

```
seconds-since-UTC1970=(DayNumber*86400)+(leap-seconds)+(seconds-since-
midnight)
```

The design, with its compound data elements, lends advantage to reading and interpreting a Common Calendar timestamp. The emitting application has the responsibility to accurately populate the timestamp parameters while it has access to leap-second and time zone information. Important aspects of the emitter's processing are transferred to the timestamp data and metadata and this simplifies the receiver's role. The receiver is relieved from performing some computationally intensive processing and there is no need for access to leap-second and local time information.

Common Calendar specifies a policy of recording the local date and time as known to the emitting system in all cases. This brings clarity and specificity to the meaning of the data and consistency and simplicity to generating and interpreting Common Calendar timestamp values.

An outline of CBF features:

- **Time structure** - mandatory fixed size 64-bit structure containing:
  - clock rate enumeration (supports seconds, milliseconds, nanoseconds, etc)
  - 36-bit counter extensible to 48, 64, and 80 bits
  - flag to indicate the presence of Days-duration extension
  - flag to indicate the presence of Local Date structure extension

- **Local Date structure** – optional fixed size 15 bytes, 120-bit, local date structure containing:
  - The CCT origin is UTC1970 (1970-01-01 00:00:10 (TAI) = 1970-01-01T00:00:00 (UTC))
  - CCT always records local time as known to the emitting application.
  - Ordinal day counter with leap-seconds for approximately -22967 to 22967 year range
  - UTC-Offset of local time
  - IANA Time Zone Database time zone
  - DST bias, change day, change time-of-day
  - UTC-Offset shifts and shift time-of-day
  - Time-of-day (TOD) Count Mode defines the linkage between the Time and Local Date:
    - TOD_LEAPSECOND_UTC_UTC - Leap-seconds are introduced simultaneous with UTC on local timescales labeled as xx:59:60 as per UTC specification
    - TOD_LEAPSECOND_UTC_NTP - Leap-seconds are introduced simultaneous with UTC on local timescales labeled as xx:59:59 ("freeze") as per NTP specification
    - TOD_LEAPSECOND_UTC_POSIX - Leap-seconds are introduced simultaneous with UTC on local timescales labeled as xx:00:00 ("roll over and reset") as per POSIX specification
    - TOD_LEAPSECOND_MIDNIGHT - Leap-seconds are introduced "at midnight" on each Local Timescale labeled as 23:59:60 as per UTC specification
    - TOD_86400S_DAY_DATE – Date and time-of-day data are treated as 86400-second-days (Leap-seconds are unavailable or unknown).
    - TOD_NONE - time not related to date

- **Geostamp**

The Common Calendar Timestamp (CCT) specification has been extended to include geographic coordinates to create a Geostamp. The Geostamp specification was developed in collaboration with Son Voba of Sync-n-Scale to support "tractability provenance". A Geostamp consists of geographic coordinates and a CCT timestamp.

For an overview of Common Calendar in general please see
Common Calendar Introduction and Scope

2 Scope

This interoperability standard specifies a variable length binary data structure with mandatory and optional components and metadata to fully represent deterministic time points and time intervals including UTC accurate local date and time.

3 Normative References

Common Calendar Date and Time Terms and Definitions

Common Calendar TAI-UTC API

Common Calendar YMDhms API

Common Calendar Local Timescales

Common Calendar Time Zone API

Common Calendar Binary Format

Common Calendar Character Format

Common Calendar Geostamp

National Marine Electronics Association
NMEA 0183 Interface Standard
GGA Global Positioning System Fix Data. Time,
Position and fix related data for a GPS receiver $GPGGA
https://www.nmea.org/nmea-0183.html

4 Common Calendar Binary Format

Common Calendar Binary Format (CBF) specifies a compact variable length binary data structure containing time, date (if applicable), and metadata. It includes sufficient information to construct a corresponding Common Calendar Character Format (CCF).

A CBF and its corresponding CCF format can have one of six meanings:

- time point with UTC accurate local date and time-of-day
- time point with UTC accurate local date with time portion having no relation to the date
- time point less than 24 hours with no relation to date (< 86400 seconds)
- time point 24 hours or greater with no relation to date (>= 86400 seconds)
- time interval less than 24 hours (< 86400 seconds)
- time interval 24 hours or greater (>= 86400 seconds)

In the case of representation of a time point of UTC accurate local date and time-of-day a CBF and its corresponding CCF shall contain the local date, time-of-day, and local metadata as known to the emitting system when generated in accordance with the rules and guidelines set out in Common Calendar Local Timescales.

A CBF shall not represent an incomplete, ambiguous, or non-deterministic time-point or interval. Any incomplete data or metadata for the intended purpose shall be regarded as an error and the CBF as a whole shall be regarded as malformed. Applications should take appropriate action to protect the system and users from incomplete or erroneous data.

4.1 CBF Components

A CBF is constructed from the mandatory Time data structure CBFTime_st with optional extensions. The components are assembled to support the required functionality. Each component is described in detail with sections below. The order of construction of optional components is shown in sections CBF Construction and Assembly.

4.1.1 Time - CBFTime_st

The CBFTime_st structure is mandatory and provides the fixed-size (64-bit) anchor of the variable length format. It contains a 35-bit unsigned counter and a sign bit (a 36-bit signed value) together with rate enumeration, counter size indicator, and extension flags.

The range of the CBFTime_st 35-bit counter may be extended to 48, 64, or 80 bits by addition of the optional CBFCounterHigh16_st and/or CBFCounterHigh32_st extensions to accommodate 24 hours at rates higher than milliseconds.

The presence of one or both of the range extensions is indicated by CBFTime_st::m_eCounterSize holding a CBFCounterSize_et enumeration. See CBFCounterSize_et. The extensions are treated as the high-words and the CBFTime_st::m_ulCounterLow32 counter as the low-word.

CCT supports several rates, or resolutions. These are indicated by enumerations. See CBFRate_et. Each requires a counter of sufficient size to accommodate 24 hours (86400 seconds) plus one leap-second when applicable (86401 seconds).

```
typedef struct CBFTime_st            // 8 bytes, 64-bit
{
unsigned char m_eRateEnumeration:8;  // Enumerated clock rates
```

```
                                     // See CBFRate_et
unsigned char m_bLocalDateExt:1;     // CBFLocalDate_st extension
                                     // is present
                                     // See CBFLocalDate_st
unsigned char m_b24HourPeriodExt:1;  // CBF24HourPeriod_st extension
                                     // is present
unsigned char m_eCounterSize:2;      // Enumerated 35/48/64-bit size
unsigned char m_bCounterSign:1;      // 1 = counter value is negative
                                     // See COUNTERSIZE_et
unsigned char m_iCounterHigh3:3;     // MSBs of 35-bit counter.
                                     // Unused when (set to zero) when
                                     // 16-bit or 32-bit high extensions
                                      // are active for 48-bit or 64-bit
                                      // counters
unsigned long m_ulCounterLow32;       // 32 bit unsigned counter
                                      // treated as low word if counter
                                      // size extension present
unsigned char m_bIsInterval:1;        // CBF is an interval
unsigned char m_reserved7:7;          // Reserved for media
unsigned char m_reserved8:8;          // Reserved for media
} CBFTime_st;
```

4.1.1.1 Rate enumeration - CBFRate_et

CBF supports several rates, or resolutions. These are indicated by enumerations. See CBFRate_et. Each requires a counter of sufficient size to accommodate 25 hours (86400 seconds) plus one leap-second when applicable (86401 seconds).

```
typedef enum v
{
 CLOCK_UNKNOWN=0,
 CLOCK_0  , // 1/1 second
 CLOCK_1  , // 1/10 tenths of second
 CLOCK_2  , // 1/100 hundredths of second
 CLOCK_3  , // 1/1000 Millisecond
 CLOCK_6  , // 1/1000000 Microsecond
 CLOCK_9  , // 1/1000000000 Nanosecond
 CLOCK_12 , // 1/1000000000000 Picosecond
 CLOCK_15 , // 1/1000000000000000 Femtosecond
 CLOCK_18 , // 1/1000000000000000000 Attosecond
 CLOCK_21 , // 1/1000000000000000000000 Zeptosecond
 CLOCK_24 , // 1/1000000000000000000000000 Yoctosecond
 CLOCK_44 , // 1/10tothe44th Planck time
} CBFRate_et;
```

4.1.1.2 Counter Size enumeration - CBFCounterSize_et

The range of the CBFTime_st 36-bit counter may be extended to 48, 64, or 80 bits by addition of the optional CBFCounterHigh16_st and/or CBFCounterHigh32_st extensions to accommodate 24 hours at rates higher than milliseconds.

The presence of one or both of the range extensions is indicated by CBFTime_st::m_eCounterSize holding a CBFCounterSize_et enumeration. See CBFCounterSize_et. The extensions are treated as the high-words and the CBFTime_st 32-bit counter as the low-word.

```
typedef enum CBFCounterSize_et     // m_eCounterSize:2
{
COUNTERSIZE_32 = 0, // 32-bit counter contained in base object
COUNTERSIZE_48,     // 48-bit counter
                    // 16-bit counter extension present,
                    // CBFTime_st::m_lCounterLow32,
COUNTERSIZE_64,     // 64-bit counter
                    // 32-bit counter extension present,
                    // TBOCounter32_st::m_ulCounterHigh32 and
                    // CBFTime_st::m_lCounterLow32,
COUNTERSIZE_80        // 80-bit counter
                    // 16-bit counter extension present,
                    // 32-bit counter extension present,
                    // TBOCounter16_st::m_unCounterHigh16 and
                    // TBOCounter32_st::m_ulCounterHigh32 and
                    // CBFTime_st::m_lCounterLow32,
} CBFCounterSize_et;
```

The counter size must accommodate 24 hours (86400 seconds) plus one leap-second if applicable (86401 seconds) of the rate, or resolution. The required sizes have been calculated, for examples, as:

Microseconds requires 37 bits -
86401 * 1000000 = 86401000000, 2^37 = 137438953472 MAX, unused 51037953472

Nanoseconds requires 47 bits -
86401 * 1000000000 = 86401000000000, 2^47 = 140737488355328 MAX, unused 54336488355328

Picoseconds requires 57-bits -
86401 * 1000000000000 = 86401000000000000, 2^57 = 144115188075856000 MAX, unused 57714188075855900

The rates supported, and the required counter sizes for each are:

```
enum        rate                resolution        counter size
CBFRate_et                                       CBFCounterSize_et
CLOCK_0     1/1 second                             COUNTERSIZE_32
CLOCK_1     1/10 tenths                            COUNTERSIZE_32
CLOCK_2     1/100 hundredths                       COUNTERSIZE_32
CLOCK_3     1/1000 Millisecond                     COUNTERSIZE_32
CLOCK_6     1/1000000 Microsecond                  COUNTERSIZE_48
CLOCK_9     1/1000000000 Nanosecond                COUNTERSIZE_48
CLOCK_12    1/1000000000000 Picosecond             COUNTERSIZE_64
CLOCK_15    1/1000000000000000 Femtosecond         COUNTERSIZE_80
CLOCK_18    1/1000000000000000000 Attosecond       COUNTERSIZE_80
```

(current code implementation supports up to picoseconds due to platform 64-bit limit)
(Zeptosecond would require 87 bits and Yoctosecond 97 bits, not currently supported)

If rate is <= CLOCK_3 (milliseconds) the CBFTime_st::m_ulCounterLow32 forms the entire counter and no counter size extension is required..

If rate is CLOCK_6 (microseconds) or CLOCK_9 (nanoseconds) CBFCounter16_st shall be added and CBFCounter16_st::m_CounterHigh16 is treated as high word and CBFTime_st::m_ulCounterLow32 as the low word of the compound 48-bit counter.

If rate is CLOCK_12 (picoseconds) CBFCounter16_st and CBFCounter32_st shall be added. The CBFCounter32_st::m_CounterHigh32 is treated as high word, CBFCounter16_st::m_CounterHigh16 as a "mid word", and CBFTime_st::m_ulCounterLow32 as the low word of the compound 80-bit counter.

See CCbf::SetTimeCounter(..) for bit-shift examples from each CBF counter size to uint64_t (unsigned 64-bit integer type).

See CCbf::GetTimeSeconds(..) for bit-shift examples to set CBF compound counter values from seconds and fractions of seconds.

4.1.2 Counter Size - CBFCounterHigh16_st, CBFCounterHigh32_st

The CBFTime_st::m_ulCounterLow32 counter can be extended to 48-bits by adding CBFCounter16_st::m_CounterHigh16, or 64-bits by adding CBFCounter32_st::m_CounterHigh32, or 80-bits by adding both. The presence of these extensions are indicated by CBFTime_st::m_eCounterSize enumeration. See CBFCounterSize_et.

If rate is <= CLOCK_3 (milliseconds) the CBFTime_st::m_ulCounterLow32 forms the entire counter.

If rate is CLOCK_6 (microseconds) or CLOCK_9 (nanoseconds) CBFCounter16_st is added and CBFCounter16_st::m_CounterHigh16 is treated as high word and CBFTime_st::m_ulCounterLow32 as the low word of the compound 48-bit counter.

If rate is CLOCK_12 (picoseconds) CBFCounter16_st and CBFCounter32_st are added and CBFCounter32_st::m_CounterHigh32 is treated as high word, CBFCounter16_st::m_CounterHigh16 as a "mid word", and CBFTime_st::m_ulCounterLow32 as the low word of the compound 80-bit counter.

See CCbf::SetTimeCounter(..) for bit-shift examples from each CBF counter size to uint64_t (unsigned 64-bit integer type).

See CCbf::GetTimeSeconds(..) for bit-shift examples to set CBF compound counter values from seconds and fractions of seconds.

The corresponding CCF encoding of rate is contained in the CCF Time Element. See CCF.h, Time Element.

```
typedef struct CBFCounterHigh16_st // 16 bit counter extension
{
  unsigned short m_unCounterHigh16;
} CBFCounterHigh16_st;

typedef struct CBFCounterHigh32_st  // 32 bit counter extension
{
  unsigned long m_ulCounterHigh32;
} CBFCounterHigh32_st;
```

4.1.3 24 Hour Period data structure - CBF24HourPeriod_st

CBF24HourPeriod_st extension is used to increase the range of the CBFTime_st counter equal to or greater than 24 hours (>= 86400 seconds).

The name "24 hour period" is used to avoid the word "day" in this context because only UTC, with its occasional 86401 second leap-second days, represents accurate calendar dates. The 24-hour Period Extension indicates a count of fixed-length 86400 second periods, not UTC days.

If the CBF24HourPeriod_st extension is present the CBFTime_st counter value shall be interpreted as a zero-base 86400 second period within the CBF24HourPeriod_st count.

The CBF24HourPeriod_st::m_ul24HourPeriods member is 21 bits thus matching the range of the 21-bit DateTaiUtc_st::m_ui1970DayNumber member of DateTaiUtc_st used by TAI-UTC API. See TaiUtcApi.h - DateTaiUtc_st.

The corresponding CCF element is 24-hour Period Element. See CCF.h, 24-hour Period Element.

CBF24HourPeriod_st can be used to extend the range of a time-point or interval.

If CBFTime_st::m_bIsInterval == false the CBF represents a time-point and the CBF24HourPeriod_st::m_ul24HourPeriods value holds a zero-based count of 24 hour periods.

The corresponding CCF 24-hour Period Element encoding used as a time-point is E delimiter - (E)vent time-point >= 24 hours

If CBFTime_st::m_bIsInterval == true the CBF represents an interval and the CBF24HourPeriod_st::m_ul24HourPeriods value holds a zero-based count of 24 hour periods.

The corresponding CCF 24-hour Period Element encoding used as an interval is P delimiter - (P)eriod interval >= 24 hours

Use of the CBFLocalDate_st is prohibited if CBFTime_st::m_bIsInterval == true because it is impossible to represent an accurate duration from a specific date and time without the full values and local time parameters to represent both the initial and final time points of that interval. For that case consider using two full CBF date-time local representations and calculate the duration between those two time points. A CBF can be used to represent that difference.

```
                                    // 32 bit 24 hour (86400
typedef struct CBF24HourPeriod_st   // second) period extension
{
```

```
unsigned long m_ul24HourPeriods:21; // count of 86400-second-
                                    // periods from arbitrary
                                    // zero origin
                                    // 2^21 = 2097152 MAX
unsigned long m_ulReserved:11;
} CBF24HourPeriod_st;
```

4.1.4 Local Date - CBFLocalDate_st

The optional CBFLocalDate_st structure represents the local calendar date together with sufficient metadata to fully describe local date and time-of-day in accordance with the rules and guidelines set out in Common Calendar Local Timescales. The CBFDstBias_st extension adds Daylight Saving bias if applicable. The CBFDstTransDay_st signals a DST change during the day if applicable. The CBFUtcShift_st extension adds support of UTC-offset shifts if applicable. The presence of CBFLocalDate_st is indicated by CBFTime_st::m_bLocalDateExt.

If the CBFLocalDate_st extension is present the CBFTime_st counter value represents time-of-day (24 hours plus one leap-second if applicable) on the calendar date indicated by the values of CBFLocalDate_st. DST information shall be provided by the CBFDstBias_st and CBFDstTransDay_st extensions if applicable.

The values and metadata of CBFTime_st, CBFLocalDate_st and CBFDstBias_st (if present) shall *always* represent the emitting system's local time. There is no need for a choice between "local time" and "GMT time", because all CBF and CCF timestamps represent local time. One of those local timescales is "Etc/UTC" which can be used if needed.

The local time information is obtained from the IANI Time Zone Database (Tz Database, TzDb) through the Time Zone API, including the time zone name and UTC offset, and DST rules if applicable. The version of the Tz Database source files is recorded to make accurate forensic analysis possible in cases where the Tz Database data may have changed since a timestamp was written. See TzDatabaseApi.h

The calendar date value itself, CBFLocalDate_st::m_DateTaiUtc_st, includes a zero-based count of days-since-UTC1970 together with the positive and negative leap-second values. This is the same DateTaiUtc_st data type used by TaiUtcApi, see TaiUtcApi.h.

Excerpt from TaiUtcApi.h

```
typedef struct DateTaiUtc_st // 7 bytes, 52-bit
{
signed long m_l1970DayNumber:24;  // signed zero-based 86400-second days
                                  // since 1970-01-01T00:00:00 (UTC)
                                  // ((2^24)/2)-1 = 8388607 MAX
                                  // 8388607 / 365.24 = ~22967.38 years
                                  // (2^24)/2)* -1 = -8388608 MIN
                                  // -8388608 / 365.24 = ~-22967.38 years
signed long m_lLeapsecsNegLow:8;  // Negative Leap-seconds low word
signed short m_nLeapsecsNegHigh:7; // Negative leap-seconds high word
                                   // Negative TAI-UTC minus initial 10s
                                   // ((2^15)/2)* -1 = -16384 MIN
signed short m_nLeapsecsHigh:9;   // Positive leap-seconds high word
signed char m_nLeapsecsLow:6;     // Positive leap-seconds low word
                                  // TAI-UTC minus initial 10s
                                  // ((2^15)/2)-1 = 16383 MAX
signed char m_nReserved:2;
} DateTaiUtc_st;
```

The origin (epoch) of the date and TAI-UTC values shall be:

441762480s TAI = 1970-01-01 00:00:10(TAI) = 1970-01-01T00:00:00(UTC) = MJD 40587. (called UTC1970). Thus seconds-since-UTC1970 = (1970DayNumber * 86400s) + positive leap-seconds + negative leap-seconds.

Time zone and UTC offset metadata are included according to the Time Zone API. See TzDatabaseApi.h

The optional CBFDstBias_st structure shall be appended to provide required DST metadata if applicable. See CBFLocalDate_st::m_bDSTExt and CBFDstBias_st.

The optional CBFDstTransDay_st structure shall be appended to provide required DST changes if applicable. See CBFLocalDate_st:: m_bDstTransDayExt and CBFDstTransDay.

The optional CBFUtcShift_st structure shall be appended to provide required UTC-Offset metadata if applicable. See CBFLocalDate_st::m_bUtcShiftExt and CBFUtcShift.

The corresponding CCF element is Date Element. See CCF.h, Date Element.

The presence of a CBFLocalDate_st is indicated by
`CBFTime_st:: m_bLocalDateExt:1`

```
typedef struct CBFLocalDate_st   // 16 bytes, 128-bit
{
DateTaiUtc_st m_DateTaiUtc_st;   // UTC1970 Day number
                                 // and Leapsecs values
                                 // See TaiUtcApi.h - 7 bytes, 52-bit
TZDTimeZoneID_st m_TZDTimeZoneID_st; // tz datebase zone id
                                     // see TzDatabaseApi.h - 4 bytes, 32-bit
signed long m_lUTCOffset:17;    // UTC Offset in seconds
                                // 17 bits signed = min -65536 / 3600 =
                                // -18.20444444 hrs
                                // 17 bits signed = max  65535 / 3600 =
                                // 18.20416667 hrs
unsigned long m_eTODMode:3;     // CBFTodCountMode_et
                                // See CBFTodCountMode_et
unsigned long m_eDstMode:2;     // DST count mode enum
                                // See CBFDstCountMode_et
unsigned long m_bDstBiasExt:1;  // CBFDstBias_st present
                                // following this
                                // TBOLocalTime_st struct
unsigned long m_bDstTransDayExt:1; // Dst Transition Day_st extension
                                   // see CBFDstTransDay_st
unsigned long m_bUtcShiftExt:1;  // CBFUtcShift_st present
                                 // see CBFUtcShift_st
unsigned char m_bIsLeapSecond:1;         // this local second is a leap-second
unsigned char m_bIsLeapSecondDay:1;      // today local is a leap-second day
unsigned char m_bIsLeapSecondNegative:1; // leap-second is negative
unsigned char m_lReserved:4;
} CBFLocalDate_st;
```

4.1.4.1 Time Zone Identity

TzDb encodes the time zone identity as strings, such as “America/New_York”, “Europe/Moscow”.
CCT encodes these as index numbers, derived from The TzDb source files.

```
typedef struct TZDTimeZoneID_st // 4 bytes, 32 bits
{
unsigned short m_unZoneIdx:10;       // tz datebase zone index
                         // 2^10 = 1024 MAX
// same variables as TZDDataRelease_st but local here for byte alignment
unsigned short m_unTzDataReleaseLetter:5; // 26 release letters a-z
                         // 2^5 = 32 MAX
unsigned short m_bCBFLocationExt:1;     // CBFLocation_st present
unsigned short m_unTzDataReleaseYear:12; // UTC1970 zero based year number
                         // 1970 + 3465 = year 5435
                         // 2^12 = 4096 MAX
unsigned short m_bCBFAbbrExt:1;      // CCbf::m_aCBFChar_st16PosixAbbr[16];
unsigned short m_bCBFAbbrChangeExt:1;   // CCbf::m_aCBFChar_st16PosixAbbr[16];
unsigned short m_unReserved:2;
} TZDTimeZoneID_st;
```

See Common Calendar Time Zone API

4.1.4.2 Posix Abbreviated Name

Tz Database assigns "time zone abbreviations" to the Posix TZ environment variable.

For example, in time zone "America/New_York", when DST is not in effect,
the TZ environment variable provided by Tz Database is "EST", and when DST is in effect, “EDT”.

Charaters are encoded in a variable length character string, CBFChar_st

```
typedef struct CBFChar_st
{
unsigned char m_Char:7;
unsigned char m_Next:1;
} CBFChar_st;
```

Each character is limited to ASCII values 0 - 126.
CBFChar_st.m_Char:7 holds the character.
CBFChar_st.m_Next:1 flags if another character follows.
  if CBFChar_st.m_Next == 1, another character follows
  if CBFChar_st.m_Next == 0, no more characters

CCT carries these encodings in class CCbf as lower-case characters in a 16 byte CBFChar_st array; see CCbf::m_aCBFChar_st16Abbr;

Only the populated CBFChar_st bytes of this array are written to the CBF binary format.

For example, for the TZ environment variable "EST"
three CBFChar_st bytes are written:
CBFChar_st.m_Char = 'e';
CBFChar_st.m_Next = 1; // have another char
CBFChar_st.m_Char = 's';
CBFChar_st.m_Next = 1; // have another char
CBFChar_st.m_Char = 't';
CBFChar_st.m_Next = 0; // last char

See CBF.h CBFChar_s
See CCbf::SetCBFChar_stStringLowerCase()
See CCbf::GetCBFChar_stString()

4.1.4.3 Posix Abbreviation Name Change

Some Tz Database transitions occur only because the Posix TZ environment variable changes. This is a rare situation, but supported by the presence CBFAbbrChange_st as flagged by CBFLocalDate_st::TZDTimeZoneID_st:m_bCBFAbbrChangeExt.

Only the populated CBFChar_st bytes of these arrays are written to the CBF binary format.

```
typedef struct CBFAbbrChange_st   // 37 bytes, 296-bit
{
unsigned long m_ulAbbrChangeTime:17; // Abbr change time-of-day
                       // 86400 = 24 hours
                       // 17 bits unsigned max 131071 / 3600 = 36.40861111 hrs
CBFChar_st m_aCBFChar_st16Abbr_Before[16];
CBFChar_st m_aCBFChar_st16Abbr_After[16];

unsigned char m_lReserved:6;
} CBFAbbrChange_st;
```

The CBF member is `CBFAbbrChange_st;`
The presence of the Posix Abbreviation Name Change is indicated by
`CBFLocalDate_st::TZDTimeZoneID_st:m_bCBFAbbrChangeExt`

4.1.4.4 Time-of-Day Count Mode - CBFTodCountMode_et

CCT provides means to construct the YMDhms counting sequence through Common Calendar Character Format (CCF) to support either: A) the widely used common practice of introducing leap-seconds simultaneous with UTC or B) an alternate scheme introducing the leap-second at the end of the local day (rolling leap-second).

CBFTodCountMode_et instructs how the CBF binary representation is to be converted to the CCF YMDhms counting sequence representation. Options include support for UTC, POSIX, or NTP YMDhms sequences.

See Common Calendar Local Timescales.

```
typedef enum CBFTodCountMode_et
{
TOD_NONE = 0,             // Not Time-of-day
                          // Time has no relation to Date,
                          // Time has zero or "arbitrary" epoch
                          // CCF char indicator "a"
TOD_LEAPSECOND_MIDNIGHT,  // Leap-seconds introduced at
                          // midnight on local timescales
                          // (Rolling leap-second)
                          // CCF char indicator "m" (midnight)
TOD_LEAPSECOND_UTC_UTC,   // Leap-seconds introduced
                          // simultaneous with UTC on
                          // local timescales
                          // Leap-second label
                          // 23:59:60
                          // CCF char indicator "u" (utc)
TOD_LEAPSECOND_UTC_NTP,   // Leap-seconds introduced
                          // simultaneous with UTC on
                          // local timescales
                          // Leap-second label
                          // 59:59:59 ("freeze")
                          // CCF char indicator "n" (ntp)
TOD_LEAPSECOND_UTC_POSIX,// Leap-seconds introduced
                          // simultaneous with UTC on
                          // local timescales
                          // Leap-second label
                          // 00:00:00 ("roll over and reset")
                          // CCF char indicator "p" (posix)
TOD_24HOUR_DAY_DATE,      // 86400-second-days of calendar
                          // (Leap-seconds unknown or unavailable)
                          // CCF char indicator "g" (gregorian)
TOD_NA                    // not set or logic error (default)
} CBFTodCountMode_et;
```

4.1.4.5 UTC Offset Shift - CBFUtcShift_st

Many time zones have shifted their UTC-offset independent of DST shifts. Tz Database calls this "STDOFF" for "standard time offset". When this occurs CBFUtcShift_st provides metadata to represent these UTC-offset shifts.

The presence of a CBFUtcShift_st is indicated by `CBFLocalDate_st::m_bUtcShiftExt:1`

```
typedef struct CBFUtcShift_st     // 5 bytes, 40-bit
{
```

```
unsigned short m_unUtcShiftTimeLow:16; // Utc offset shift time-of-day in
seconds
signed short m_nUtcShiftLow:16;      // Utc offset shift in seconds

unsigned char m_eUtcShiftDay:2;      // Utc offset shift day enum
                                     // See TzDatabaseApi.h TZDUtcShiftDay_et
signed char m_nUtcShiftHigh:2;       // Utc offset shift in seconds
                                     // 18 bit min -131072 / 3600 = -
36.40888889 hr
                                     // 18 bit max  131071 / 3600 =
36.40861111 hr
unsigned char m_unUtcShiftTimeHigh:1;  // Utc offset shift time-of-day in
                                       // seconds
                                       // 86400 = 24 hours
                                       // 17 bits unsigned max 131071 / 3600 =
                                       // 36.40861111 hrs
unsigned char m_unReserved3:3;

} CBFUtcShift_st;
```

4.1.5 Daylight Saving Time (DST)

Where Daylight Saving is observed the optional CBFDstBias_st struct extension shall be added. The presence of CBFDstBias_st is indicated by CBFLocalDate_st:: m_bDstBiasExt:1.

4.1.5.1 Daylight Saving - CBFDstCountMode_et

Daylight Saving offsets may be applied in two modes:

“Conventional” where the DST shift occurs at some mid time-of-day

“Uninterrupted” where the DST shift at midnight

```
typedef enum CBFDstCountMode_et
{
DSTCOUNTMODE_NOTAPPLICABLE = 0,
DSTCOUNTMODE_CONVENTIONAL, // conventional count mode
DSTCOUNTMODE_UNINTERRUPTED // uninterrupted count mode
} CBFDstCountMode_et;
```

4.1.5.2 Daylight Saving - CBFDstBias_st

If a DST shift is in effect its value, called “DstBias”, is held by CBFDstBias_st:: m_nDstBias

Presence is signaled by CBFLocalDate_st:: m_bDstBiasExt:1

```
typedef struct CBFDstBias_st  // 2 bytes, 16-bit
{
signed short m_nDstBias;     // DST bias in effect
                             // min -32768 / 3600 = -9.102222222 hr
                             // max  32767 / 3600 = 9.101944444 hr
} CBFDstBias_st;
```

4.1.5.3 Daylight Saving - CBFDstTransDay_st

If a DST shift is to occur during this day the CBFDstTransDay_st shall appear.

Presence is signaled by CBFLocalDate_st:: m_bDstTransDayExt:1

```
typedef struct CBFDstTransDay_st  // 5 bytes, 40-bit
{
```

```
unsigned short m_unDstTransTimeLow:16;// DST transition time of day
                       // 17 bit unsigned max 131071 = 36.40861111 hr
signed short m_nDstBiasChangeLow:16;  // DST bias transition value
                       // 18 bit min -131072 / 3600 = -36.40888889 hr
                       // 18 bit max  131071 / 3600 =  36.40861111 hr
signed char m_nDstBiasChangeHigh:2;
unsigned char m_unDstTransTimeHigh:1;
unsigned char m_lReserved5:5;
} CBFDstTransDay_st;
```

Enumerated values used by CBFDst_st variables are defined by Tz Database API. See TzDatabaseApi.h

```
typedef enum TZDDstChangeDay_et
typedef enum TZDDstBias_et
```

4.2 Construction of CBF

CBFTime_st structure is mandatory and the anchor of the binary image. Optional counter extensions may be used to increase its counter range for any configuration.

The state of the CBFTime_st::m_bIsInterval flag indicates if the CBF represents a time-point (m_bIsInterval == false) or a time interval (m_bIsInterval == true).

4.2.1 As Time-point:

If m_bIsInterval == false a CBFTime_st represents a time-point.

Used without the optional CBFLocalDate_st extension, the values of a CBFTime_st represent a time point within a 24 hour period (86400 seconds) with no reference to any date or other timescale. It represents a zero-based count of time units from an origin marked as "zero". It is a timestamp of a simple timer, like a "game clock" or "stop-watch".

Combined with the optional CBFLocalDate_st extension the CBF represents a leap-second accurate local date and time. The CBFTime_st counter value represents time-of-day (24 hours plus one leap-second if applicable) on the calendar date indicated by the CBFLocalDate_st member values. The presence of CBFLocalDate_st is indicated by the CBFTime_st::m_bLocalDateExt flag.

If DST is observed in the time zone, the CBFDst_st extension shall be present, providing the necessary DST information. The presence of CBFDst_st is indicated by the CBFLocalDate_st::m_bDSTExt flag.

4.2.1.1 As UTC accurate local time-point timestamp :

CBFTime_st – mandatory
CBFTime_st::m_bIsInterval == false
CBFCounterHigh16_st and/or CBFCounterHigh32_st optional
CBFLocalDate_st – local date extension
CBFLocalDate_st::m_eTODMode == one of CBFTodCountMode_et:

- TOD_LEAPSECOND_UTC_UTC - Leap-seconds introduced simultaneous with UTC on local timescales, leap-second label 23:59:60, CCF TOD Count Mode char indicator "u" (utc)
- TOD_LEAPSECOND_UTC_NTP - Leap-seconds introduced simultaneous with UTC on local timescales, leap-second label 59:59:59 ("freeze"), CCF TOD Count Mode char indicator "n" (ntp)
- TOD_LEAPSECOND_UTC_POSIX - Leap-seconds introduced simultaneous with UTC on local timescales, leap-second label 00:00:00 ("roll over and reset"), CCF TOD Count Mode char indicator "p" (posix)
- TOD_LEAPSECOND_MIDNIGHT - Leap-seconds introduced at midnight on local timescales, Leap-second label 23:59:60, CCF TOD Count Mode char indicator "m" (midnight)

CBFDst_st – Daylight Saving Time metadata extension (if applicable)

4.2.1.2 As UTC accurate local date with time portion having no relation to the date:

CBFTime_st – mandatory
CBFTime_st::m_bIsInterval == false
CBFCounterHigh16_st and/or CBFCounterHigh32_st optional
CBFLocalDate_st – local date extension

CBFDstBia_st – Daylight Saving Time metadata extension (if applicable)
CBFDstTransDay_st - Daylight Saving Change Day
CBFLocalDate_st::m_eTODMode == CBFTodCountMode_et ==
  TOD_NONE - Not Time-of-day, Time has no relation to Date, Time has zero or "arbitrary" epoch
  CCF TOD Count Mode char indicator "a"

4.2.1.3 As time-point timestamp < 86400 seconds:

CBFTime_st – mandatory
CBFTime_st::m_bIsInterval == false
CBFCounterHigh16_st and/or CBFCounterHigh32_st optional

4.2.1.4 As time-point timestamp >= 86400 seconds:

CBFTime_st – mandatory
CBFTime_st::m_bIsInterval == false
CBFCounterHigh16_st and/or CBFCounterHigh32_st optional
CBF24HourPeriod_st – 24 hour period counter extension

4.2.2 As Time Interval

If CBFTime_st::m_bIsInterval == true the CBF represents an interval, rather than a time-point.

Used without the optional CBF24HourPeriod_st extension the CBF represents an interval less than 24 hours (< 86400 seconds).

If optional CBF24HourPeriod_st extension is present the CBF represents an interval equal to or greater than 24 hours (>= 86400 seconds). The CBF24HourPeriod_st::m_ul24HourPeriods value represents a count of 24 hour periods (1 == 86400 seconds) and the CBFTime_st counter value represents a count of 86400 seconds within that 24 hour period. The CBF24HourPeriod_st shall be present if the state of CBFTime_st::m_bIsInterval == true and the total counter value exceeds 86400.

Note intervals >= 86400 seconds are not accurate UTC date and time because Leap-seconds are not accounted for; intervals are made up of 24 hour (86400 second) periods, not UTC accurate days.

4.2.2.1 As time interval < 86400 seconds:

CBFTime_st – mandatory
CBFTime_st::m_bIsInterval == true
CBFCounterHigh16_st and/or CBFCounterHigh32_st optional

4.2.2.2 As time interval >= 86400 seconds:

CBFTime_st – mandatory
CBFTime_st::m_bIsInterval == true
CBFCounterHigh16_st and/or CBFCounterHigh32_st optional
CBF24HourPeriod_st – 24 hour period counter extension

4.2.3 Assembly Order

A CBF shall be formed with the mandatory CBFTime_st structure with optional components concatenated in the following order:

- To form a time-point timestamp < 86400 seconds:

```
CBFTime_st - mandatory and CBFTime_st::m_bIsInterval == false
  CBFCounterHigh16_st and/or CBFCounterHigh32_st optional
  extensions to form 48, 64, or 80 bit counter
```

- To form a time-point timestamp >= 86400 seconds:

```
CBFTime_st - mandatory and CBFTime_st::m_bIsInterval == false
  CBFCounterHigh16_st and/or CBFCounterHigh32_st optional
  extensions to form 48, 64, or 80 bit counter
CBF24HourPeriod_st - optional 24 hour period counter
```

- To form a UTC accurate local time-point timestamp:

```
CBFTime_st - mandatory and CBFTime_st::m_bIsInterval == false
  CBFCounterHigh16_st and/or CBFCounterHigh32_st optional
  extensions to form 48, 64, or 80 bit counter
CBFLocalDate_st - optional local date extension
CBFDst_st - optional Daylight Savings Time metadata extension if applicable
CBFUtcShift_st - optional UTC-offset shift metadata extension if applicable
```

```
Variable length Posix TZ environment string of CBFChar_st
                in array CCbf::m_aCBFChar_st16Abbr[]
```

*Interval and LocalDate are exclusive*

- To form a interval < 86400 seconds:

```
CBFTime_st - mandatory and CBFTime_st::m_bIsInterval == true
  CBFCounterHigh16_st and/or CBFCounterHigh32_st optional
  extensions to form 48, 64, or 80 bit counter
```

- To form an interval >= 86400 seconds:

```
CBFTime_st - mandatory and CBFTime_st::m_bIsInterval == true
  CBFCounterHigh16_st and/or CBFCounterHigh32_st optional
  extensions to form 48, 64, or 80 bit counter
CBF24HourPeriod_st - optional day duration counter
```

CBF assembly order in pseudocode -

```
CBFTime_st - mandatory primary component

if(m_CBFTime_st.m_eCounterSize == COUNTERSIZE_48)
  append CBFCounterHigh16_st
if(m_CBFTime_st.m_eCounterSize == COUNTERSIZE_64)
  append CBFCounterHigh32_st
if(m_CBFTime_st.m_eCounterSize == COUNTERSIZE_80)
  append CBFCounterHigh16_st
  append CBFCounterHigh32_st

// Interval and LocalDate are exclusive
if(m_CBFTime_st.m_bIsInterval)
 {
 if(m_CBFTime_st.m_b24HourPeriodExt)
  append CBF24HourPeriod_st
 }
else
 {
 if(m_CBFTime_st.m_bLocalDateExt)
  append CBFLocalDate_st
 if(m_CBFLocalDate_st.m_bUtcShiftExt)
  append CBFUtcShift_st
 if(m_CBFLocalDate_st.m_bDSTExt)
  append CBFDst_st
 if(m_CBFLocalDate_st.m_TZDTimeZoneID_st.m_bCBFAbbrExt)
  append number of CBFChar_st in array m_aCBFChar_st16Abbr[]
 if(m_CBFLocalDate_st.m_TZDTimeZoneID_st.m_bCBFLocationExt)
  append CBFLocation_st
 }
```

See `CCbf::AssembleCbf(char* pcaBinayCbf, int* piLen);`

Assembled CBF size guide:

- Minimum size 8 bytes (mandatory CBFTime_st)
- Typical size with date and time may be the range of 30 - 34 bytes
- Maximum size with date, time and location could be 76 bytes

4.3 Geostamp

The Common Calendar Timestamp (CCT) specification may include geographic coordinates to create a Geostamp. The Geostamp specification was developed in collaboration with Son Voba of Sync-n-Scale to support "tractability provenance".

A Geostamp consists of geographic coordinates and a CCT timestamp. Geostamps are technically accurate, making them suitable for general and legal purposes where time recording is used for tracking and auditing and a wide range of spatial-temporal geographic information systems (4D GIS) applications in machine learning, artificial intelligence, data analytics and blockchain distributed ledgers.

Like CCT, Geostamps can be formed in either a binary or character format. The binary format supports efficient machine interoperability while the character format is human readable making their meaning accessible to those less familiar with the intricacies of timekeeping and geographic representations.

CCT carries coordinates in the form specified by National Marine Electronics Association (NMEA), NMEA 0183 Interface Standard, GPGGA., GGA Global Positioning System Fix Data. Time, Position and fix related data for a GPS receiver.

The NMEA data is translated into the CBFLocation_st structure in the CCT binary CBF format. The CBF data is reflected in the CCF character format in the Location Element field.

Geographic coordinates are carried in the CBFLocation_st structure.

The presence of the CBFLocation_st structure in the CCT timestamp is indicated by the CBFLocalDate_st::m_TZDTimeZoneID_st::m_bCBFLocationExt flag.

4.3.1 Geographic Coordinates - CBFLocation_st

The CBFLocation_st struct carries coordinates in a compact form consistent with data as specified by the NMEA 0183 Interface Standard, GPGGA, GGA Global Positioning System Fix Data. Time, Position and fix related data for a GPS receiver.

```
typedef struct CBFLocation_st // 14 bytes
{

unsigned long m_i21Lat_uMin:21;  // micro-minutes -100000 to 100000 range
                                 //[((2^21)/2)-1  =  1048575 MAX]
unsigned long m_i9Lat_Deg:9;     // degrees -90 to 90 range, negative is South
                                 // [((2^9) / 2) - 1 = 255 MAX]
unsigned long m_Pad1:2;

unsigned long m_i21Lng_uMin:21;  // micro-minutes -100000 to 100000 range
                                 // [((2^21)/2)-1  =  1048575 MAX]
unsigned long m_i9Lng_Deg:9;     // degrees -180 to 180 range, negative is
                                 // West [((2^9) / 2) - 1 = 255 MAX]
unsigned long m_Pad2:2;

unsigned long m_i19Alt_cm:19;    // +- 20000 meter x 100 centimeter range
                                 // [((2^19) / 2) - 1 = 262143 MAX]
unsigned long m_i7Lng_Min:7;     // minutes 0 to 60 range [((2^7) / 2) - 1 =
                                 // 63 MAX]
unsigned long m_Pad3:6;

unsigned char m_i7Lat_Min:7;     // minutes 0 to 60 range [((2^7) / 2) - 1 =
                                 // 63 MAX]
unsigned char m_bSourceIsExtern:1; // flag is external location, otherwise is
                                   // Tz Database location
unsigned char m_bIsValidLat:1;   // flag Latitude value valid
unsigned char m_bIsValidLng:1;   // flag Longitude value valid
unsigned char m_bIsValidAlt:1;   // flag Altitude value valid
unsigned char m_Pad4:5;

} CBFLocation_st;
```

See TzDatabaseAPI.h, CBFLocation_st
See CCct.h, CCct.cpp class CCct
int CCct::SetLocation(char* psLatitude, char* psLongitude, char* psAltitude);

4.4 CBF RIFF Wrapper

One or more CBFs may be contained in a CBF RIFF Wrapper.

The four-cc of the CBF RIFF shall be "CCBF"

The four-cc of each CBF chunk shall be "ccbf"

See CCTRiffCbf.h

```
#define FOURCC_CCBF "CCBF" // RIFF header
#define FOURCC_ccbf "ccbf" // CBF chunk
```

See CCTRiffCbf.h and CCTRiffCbf.cpp

The CBF RIFF Wrapper can be opened, populated by one or more assembled CCBs, and closed by

```
CCct::WriteCRiff_Ccbf();
```

The CBF RIFF Wrapper can be opened, populating one or more CCbf classes, and closed by

```
CCct::ReadCRiff_Ccbf();
```