

```

#pragma once

class SCDatetime;

#define SCDATETIME_SUPPORT_CHRONO 1

#ifdef SCDATETIME_SUPPORT_CHRONO
#include <chrono>
#endif
#include <cstdint> // std::int64_t
#include <math.h> // ceil, floor, fabs
#include <string.h> // strcmp
#include <time.h> // time_t

// Set 1 to enable warning of usage the SCDatetime `deprecated` functions
#define SHOW_WARNING_IF_DEPRECATED 0

#ifdef SHOW_WARNING_IF_DEPRECATED
# define DEPRECATED(MESSAGE) __declspec(deprecated(MESSAGE))

// Disable warning within SCDatetime.h file because we want to use these deprecated functions internally
# pragma warning(push)
# pragma warning(disable: 4996)
#else
# define DEPRECATED(MESSAGE)
#endif

// A date in the number of days since EPOCH.
typedef int t_SCDate;

// A time of day in the number of seconds since midnight.
typedef int t_SCTime;

enum MonthEnum
{
    JANUARY = 1
    , FEBRUARY = 2
    , MARCH = 3
    , APRIL = 4
    , MAY = 5
    , JUNE = 6
    , JULY = 7
    , AUGUST = 8
    , SEPTEMBER = 9
    , OCTOBER = 10
    , NOVEMBER = 11
    , DECEMBER = 12
};

static const char* FULL_NAME_FOR_MONTH[13] = {"", "January", "February", "March", "April", "May", "June", "July",
"August", "September", "October", "November", "December"};
static const char* SHORT_NAME_FOR_MONTH[13] = {"", "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep",
"Oct", "Nov", "Dec"};
static const int DAYS_IN_MONTH[13] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
static const int DAYS_IN_YEAR_AT_MONTH_END[13] = {0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
static const char FUTURES_CODE_FOR_MONTH[13] = {'\0', 'F', 'G', 'H', 'J', 'K', 'M', 'N', 'Q', 'U', 'V', 'X', 'Z'};

enum DayOfWeekEnum
{
    DAY_OF_WEEK_NOT_SET = -1
    , SUNDAY = 0
    , MONDAY = 1
    , TUESDAY = 2
    , WEDNESDAY = 3
    , THURSDAY = 4
    , FRIDAY = 5
    , SATURDAY = 6
}

```

```

};

enum DayOfWeekMondayBasedEnum
{ DAY_OF_WEEK_MONDAY_BASED_NOT_SET = -1
, MONDAY_MONDAY_BASED = 0
, TUESDAY_MONDAY_BASED = 1
, WEDNESDAY_MONDAY_BASED = 2
, THURSDAY_MONDAY_BASED = 3
, FRIDAY_MONDAY_BASED = 4
, SATURDAY_MONDAY_BASED = 5
, SUNDAY_MONDAY_BASED = 6
};

const double DATETIME_UNSET_DOUBLE = 0;
const int64_t DATETIME_UNSET = 0;

static const char* FULL_NAME_FOR_DAY_OF_WEEK[7] = {"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday",
"Friday", "Saturday"};
static const char* SHORT_NAME_FOR_DAY_OF_WEEK[7] = {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"};
static const char* TWO_CHAR_NAME_FOR_DAY_OF_WEEK[7] = {"Su", "Mo", "Tu", "We", "Th", "Fr", "Sa"};

static const int32_t DAYS_IN_YEAR = 365; // Approximate

static const int MONTHS_PER_YEAR = 12;
static const int DAYS_PER_WEEK = 7; // number of days in a week
static const int HOURS_PER_DAY = 24; // number of hours in a day
static const int MINUTES_PER_HOUR = 60; // number of minutes in an hour
static const int SECONDS_PER_MINUTE = 60; // number of seconds in a minute
static const int MILLISECONDS_PER_SECOND = 1000; // number of milliseconds in a second
static const std::int64_t MICROSECONDS_PER_MILLISECOND = 1000; // number of microseconds in a millisecond
static const int NANOSECONDS_PER_MICROSECOND = 1000; // number of nanoseconds in a microsecond

static const int MINUTES_PER_DAY = MINUTES_PER_HOUR*HOURS_PER_DAY; // number of minutes in a day = 1
440
static const int SECONDS_PER_DAY = SECONDS_PER_MINUTE*MINUTES_PER_DAY; // number of seconds in a
day = 86 400
static const int SECONDS_PER_HOUR = SECONDS_PER_MINUTE*MINUTES_PER_HOUR; // number of seconds in
an hour = 3 600
static const int MILLISECONDS_PER_DAY = MILLISECONDS_PER_SECOND*SECONDS_PER_DAY; // number of
milliseconds in a day = 86 400 000
static const int MILLISECONDS_PER_HOUR = MILLISECONDS_PER_SECOND*SECONDS_PER_HOUR; // number of
milliseconds in an hour = 3 600 000
static const int MILLISECONDS_PER_MINUTE = MILLISECONDS_PER_SECOND*SECONDS_PER_MINUTE; //
number of milliseconds in a minute = 60 000
static const std::int64_t MICROSECONDS_PER_DAY =
MICROSECONDS_PER_MILLISECOND*MILLISECONDS_PER_DAY; // number of microseconds in a day = 86 400 000
000
static const std::int64_t MICROSECONDS_PER_HOUR =
MICROSECONDS_PER_MILLISECOND*MILLISECONDS_PER_HOUR; // number of microseconds in an hour = 3 600
000 000
static const std::int64_t MICROSECONDS_PER_MINUTE =
MICROSECONDS_PER_MILLISECOND*MILLISECONDS_PER_MINUTE; // number of microseconds in a minute = 60
000 000
static const std::int64_t MICROSECONDS_PER_SECOND =
MICROSECONDS_PER_MILLISECOND*MILLISECONDS_PER_SECOND; // number of microseconds in a second = 1
000 000
static const std::int64_t NANOSECONDS_PER_DAY =
NANOSECONDS_PER_MICROSECOND*MICROSECONDS_PER_DAY; // number of nanoseconds in a day = 86 400
000 000 000
static const std::int64_t NANOSECONDS_PER_SECOND =
NANOSECONDS_PER_MICROSECOND*MICROSECONDS_PER_SECOND; // number of nanoseconds in a second =
1 000 000 000
static const std::int64_t NANOSECONDS_PER_MILLISECOND =
NANOSECONDS_PER_MICROSECOND*MICROSECONDS_PER_MILLISECOND; // number of nanoseconds in a
millisecond = 1 000 000

```

```

static const int EPOCH_YEAR = 1899;
static const int EPOCH_MONTH = 12;
static const int EPOCH_DAY = 30;

static const int SCDATETIME_UNIX_EPOCH_DATE = 25569; // 1970-01-01
static const double SCDATETIME_UNIX_EPOCH = 25569.0; // 1970-01-01 00:00:00

#ifdef SCDATETIME_SUPPORT_CHRONO
// This is the SCDateTime epoch (1899-12-30 00:00:00) as a system_clock
// time_point.
//
// The from_time_t method takes a number of seconds since the Unix epoch,
// which is 1970-01-01 00:00:00. The difference between the two epochs is
// 25569 days, and there are 86400 seconds per day. The product of these two
// values would overflow a 32-bit int, so the ll literal suffix is used to
// prevent that from happening. Since SCDateTime epoch is before Unix epoch,
// the offset from the Unix epoch is negative.
static const auto SCDATETIME_EPOCH_SYSTEM_CLOCK
    = std::chrono::system_clock::from_time_t(-25569ll * 86400ll);
#endif

/*=====*/
inline int DaysSince1Jan0AtYearBeginning(const int Year)
{
    return Year * 365 + Year / 4 - Year / 100 + Year / 400;
}

static const int DAYS_BEFORE_UNIX_EPOCH = DaysSince1Jan0AtYearBeginning(1970) + 1; // 1970-01-01
static const int DAYS_BEFORE_SC_EPOCH = DaysSince1Jan0AtYearBeginning(1900) - 1; // 1899-12-30

int64_t SCDateTimeDoubleToSCDateTime(const double DateTime);

/*=====*/
DEPRECATED("The IS_UNSET is deprecated. Use portable SCDateTime::IsUnset() instead.")
inline bool IS_UNSET(const double DateTime)
{
    return DateTime == DATETIME_UNSET_DOUBLE;
}

/*=====*/
DEPRECATED("The IS_UNSET is deprecated. Use portable SCDateTime::IsUnset() instead.")
inline bool IS_UNSET(const int64_t DateTime)
{
    return DateTime == DATETIME_UNSET;
}

/*=====*/
inline int64_t ADJUST(const int64_t Value, const int64_t Epsilon)
{
    if (Value < 0)
        return Value - Epsilon;
    else
        return Value + Epsilon;
}

/*=====*/
inline double ADJUST(const double Value, const double Epsilon = 0.5)
{
    if (Value < 0)
        return Value - Epsilon;
    else
        return Value + Epsilon;
}

/*=====*/

```

```

DEPRECATED("The ROUND_US is private and deprecated.")
inline double ROUND_US(const double DateTime)
{
    return ADJUST(DateTime, 1.0 / MICROSECONDS_PER_DAY / 2);
}

/*=====*/
DEPRECATED("The ADJUST_ROUND_SECOND is private and deprecated.")
inline int64_t ADJUST_ROUND_SECOND(int64_t DateTime)
{
    return ADJUST(DateTime, MICROSECONDS_PER_SECOND / 2);
}

/*=====
Add 500 microseconds to the given DateTime value.
-----*/
DEPRECATED("The ADJUST_ROUND_MS is private and deprecated.")
inline int64_t ADJUST_ROUND_MS(int64_t DateTime)
{
    return ADJUST(DateTime, MICROSECONDS_PER_MILLISECOND / 2);
}

/*=====
Gets the date part of a SCDatetime. Returns the number of days since
1899-12-30.
-----*/
DEPRECATED("The DATE_PART is deprecated. Use portable SCDatetime::GetDate() instead.")
inline int DATE_PART(const int64_t DateTime)
{
    return static_cast<int>(ADJUST_ROUND_SECOND(DateTime) / MICROSECONDS_PER_DAY);
}

/*=====*/
DEPRECATED("Usage of DATE_PART is deprecated. Use portable SCDatetime::GetDate() instead.")
inline int DATE_PART(double DateTimeDouble)
{
    const int64_t DateTime = SCDatetimeDoubleToSCDateTime(DateTimeDouble);

    return DATE_PART(DateTime);
}

/*=====
Returns only the time part (ignoring the date) of the given DateTime
value as the number of seconds since midnight. The return value will be
in the range [0..86399]. The given value is rounded to the nearest
second.
-----*/
DEPRECATED("Usage of TIME_PART is deprecated. Use portable SCDatetime::GetTime() instead.")
inline int TIME_PART(const int64_t DateTime)
{
    int64_t DateTimeAdjusted = ADJUST_ROUND_SECOND(DateTime);
    DateTimeAdjusted %= MICROSECONDS_PER_DAY; // remove the date part
    int64_t TimeInSeconds = DateTimeAdjusted / MICROSECONDS_PER_SECOND;

    if (TimeInSeconds >= SECONDS_PER_DAY)
        TimeInSeconds = 0;

    return static_cast<int>(TimeInSeconds);
}

/*=====*/
DEPRECATED("Usage of TIME_PART is deprecated. Use portable SCDatetime::GetTime() instead.")
inline int TIME_PART(double DateTimeDouble)
{
    const int64_t DateTime = SCDatetimeDoubleToSCDateTime(DateTimeDouble);

```

```

    return TIME_PART(DateTime);
}

/*=====
Gets the date part of a SCDatetime with millisecond rounding. Returns
the number of days since EPOCH.
-----*/
DEPRECATED("Usage of DATE_PART_MS is deprecated. Use portable SCDatetime::GetDateMS() instead.")
inline int DATE_PART_MS(const int64_t DateTime)
{
    return static_cast<int>(DateTime / MICROSECONDS_PER_DAY);
}

/*=====
DEPRECATED("Usage of DATE_PART_MS is deprecated. Use portable SCDatetime::GetDateMS() instead.")
inline int DATE_PART_MS(double DateTimeDouble)
{
    const int64_t DateTime = SCDatetimeDoubleToSCDateTime(DateTimeDouble);

    return DATE_PART_MS(DateTime);
}

/*=====
Returns only the time part (ignoring the date) of the given DateTime
value as the number of seconds since midnight. The return value will be
in the range 0 to 86399. The microsecond component is discarded so that the
value is truncated to the containing second.
-----*/
DEPRECATED("Usage of TIME_PART_MS is deprecated. Use portable SCDatetime::GetTime() instead.")
inline int TIME_PART_MS(const int64_t DateTime)
{
    int64_t TimeOnly = DateTime;
    TimeOnly %= MICROSECONDS_PER_DAY; // remove the date part
    int64_t TimeInSeconds = TimeOnly / MICROSECONDS_PER_SECOND;

    if (TimeInSeconds >= SECONDS_PER_DAY)
        TimeInSeconds = 0;

    return static_cast<int>(TimeInSeconds);
}

/*=====
DEPRECATED("Usage of TIME_PART_MS is deprecated. Use portable SCDatetime::GetTime() instead.")
inline int TIME_PART_MS(double DateTimeDouble)
{
    const int64_t DateTime = SCDatetimeDoubleToSCDateTime(DateTimeDouble);

    return TIME_PART_MS(DateTime);
}

/*=====
Returns only the seconds part of the given DateTime, ignoring any
date part or time part that is 1 minute or greater. The return value
will be in the range [0..59]. The given value is rounded to the nearest
second.
-----*/
DEPRECATED("The SECONDS_PART is deprecated. Use portable SCDatetime::GetSecond() instead.")
inline int SECONDS_PART(const int64_t DateTime)
{
    // Round to the nearest second
    const int64_t DateTimeAdjusted = ADJUST_ROUND_SECOND(DateTime);
    int SecondsPart = (DateTimeAdjusted / MICROSECONDS_PER_SECOND) % SECONDS_PER_MINUTE;

    if (DateTime < 0)
        SecondsPart = -SecondsPart;
}

```

```

    return SecondsPart;
}

/*=====*/
DEPRECATED("The SECONDS_PART is deprecated. Use portable SCDateTime::GetSecond() instead.")
inline int SECONDS_PART(double DateTimeDouble)
{
    const int64_t DateTime = SCDateTimeDoubleToSCDateTime(DateTimeDouble);

    return SECONDS_PART(DateTime);
}

/*=====
Returns only the milliseconds part of the given DateTime, ignoring any
date part or time part that is 1 second or greater. The return value
will be in the range [0..999].
=====*/
DEPRECATED("The MS_PART is deprecated. Use portable SCDateTime::GetMillisecond() instead.")
inline int MS_PART(const int64_t DateTime)
{
    const int MillisecondsPart = (DateTime / MICROSECONDS_PER_MILLISECOND) %
MILLISECONDS_PER_SECOND;

    return MillisecondsPart;
}

/*=====*/
DEPRECATED("The MS_PART is deprecated. Use portable SCDateTime::GetMillisecond() instead.")
inline int MS_PART(double DateTimeDouble)
{
    const int64_t DateTime = SCDateTimeDoubleToSCDateTime(DateTimeDouble);

    return MS_PART(DateTime);
}

/*=====
Combines a date (days since 1899-12-30) and time (seconds since midnight)
into a SCDateTime/OLE Date Time value.
=====*/
DEPRECATED("The COMBINE_DATE_TIME is deprecated. Use SCDateTime::SCDateTime(int Date, int
TimeInSeconds) constructor instead.")
inline double COMBINE_DATE_TIME(int Date, int Time)
{
    return (Date) + Time / static_cast<double>(SECONDS_PER_DAY);
}

/*=====*/
DEPRECATED("The COMBINE_DATE_TIME is deprecated. Use SCDateTime::SCDateTime(int Date, int
TimeInSeconds) constructor instead.")
inline int64_t COMBINE_DATE_TIME_US(int Date, int Time)
{
    return (Date)* MICROSECONDS_PER_DAY + Time * MICROSECONDS_PER_SECOND;
}

/*=====*/
DEPRECATED("The DAY_OF_WEEK is deprecated. Use SCDateTime::GetDayOfWeek() instead.")
inline DayOfWeekEnum DAY_OF_WEEK(int Date)
{
    Date -= 1;
    // Special handling for zero and negative values.
    if (Date < 0)
        Date = Date % DAYS_PER_WEEK + DAYS_PER_WEEK;

    return static_cast<DayOfWeekEnum>(Date % DAYS_PER_WEEK);
}

```

```

}

/*=====*/
DEPRECATED("The DAY_OF_WEEK is deprecated. Use SCDatetime::GetDayOfWeek() instead.")
inline DayOfWeekEnum DAY_OF_WEEK(double DateTime)
{
    return DAY_OF_WEEK(DATE_PART(DateTime));
}

/*=====*/
DEPRECATED("The DAY_OF_WEEK_MONDAY_BASED is deprecated. Use
SCDateTime::GetDayOfWeekMondayBased() instead.")
inline DayOfWeekMondayBasedEnum DAY_OF_WEEK_MONDAY_BASED(int Date)
{
    Date -= 2;
    if (Date < 0)
        Date = DAYS_PER_WEEK + Date % DAYS_PER_WEEK ;

    return static_cast<DayOfWeekMondayBasedEnum>(Date % DAYS_PER_WEEK);
}

/*=====*/
inline const char* GetFullNameForMonth(int Month)
{
    if (Month < JANUARY || Month > DECEMBER)
        return FULL_NAME_FOR_MONTH[0];

    return FULL_NAME_FOR_MONTH[Month];
}

/*=====*/
inline const char* GetShortNameForMonth(int Month)
{
    if (Month < JANUARY || Month > DECEMBER)
        return SHORT_NAME_FOR_MONTH[0];

    return SHORT_NAME_FOR_MONTH[Month];
}

/*=====*/
inline int GetMonthFromShortName(const char* ShortName)
{
    if (ShortName == NULL)
        return 0;

    for (int Month = JANUARY; Month <= DECEMBER; ++Month)
    {
        #if _MSC_VER >= 1400
        if (_stricmp(ShortName, SHORT_NAME_FOR_MONTH[Month]) == 0)
        #elif __linux__
        if (strcasecmp(ShortName, SHORT_NAME_FOR_MONTH[Month]) == 0)
        #else
        if (stricmp(ShortName, SHORT_NAME_FOR_MONTH[Month]) == 0)
        #endif
            return Month;
    }

    return 0;
}

/*=====
Returns the month (JANUARY-DECEMBER) for the given MonthCode. Returns 0
if the given MonthCode does not match any of the futures codes for months.
=====*/
inline int GetMonthFromFuturesCode(char MonthCode)
{

```

```

// Make uppercase
if (MonthCode >= 'a' && MonthCode <= 'z')
    MonthCode += 'A' - 'a';

for (int Month = JANUARY; Month <= DECEMBER; ++Month)
{
    if (MonthCode == FUTURES_CODE_FOR_MONTH[Month])
        return Month;
}

return 0;
}

/*=====*/
inline bool IsValidFuturesMonthCode(char MonthCode)
{
    return (GetMonthFromFuturesCode(MonthCode) != 0);
}

/*=====*/
inline const char* GetFullNameForDayOfWeek(int DayOfWeek)
{
    if (DayOfWeek < SUNDAY || DayOfWeek > SATURDAY)
        return "";

    return FULL_NAME_FOR_DAY_OF_WEEK[DayOfWeek];
}

/*=====*/
inline const char* GetShortNameForDayOfWeek(int DayOfWeek)
{
    if (DayOfWeek < SUNDAY || DayOfWeek > SATURDAY)
        return "";

    return SHORT_NAME_FOR_DAY_OF_WEEK[DayOfWeek];
}

/*=====*/
inline const char* GetTwoCharNameForDayOfWeek(int DayOfWeek)
{
    if (DayOfWeek < SUNDAY || DayOfWeek > SATURDAY)
        return "";

    return TWO_CHAR_NAME_FOR_DAY_OF_WEEK[DayOfWeek];
}

/*=====*/
inline DayOfWeekEnum GetDayOfWeekFromInt(int DayOfWeek)
{
    if (DayOfWeek < SUNDAY || DayOfWeek > SATURDAY)
        return DAY_OF_WEEK_NOT_SET;

    return static_cast<DayOfWeekEnum>(DayOfWeek);
}

/*=====*/
inline DayOfWeekEnum GetDayOfWeekFromFullName(const char* FullName)
{
    if (FullName == NULL)
        return DAY_OF_WEEK_NOT_SET;

    for (int DayOfWeek = SUNDAY; DayOfWeek <= SATURDAY; ++DayOfWeek)
    {
        #if _MSC_VER >= 1400
        if (_stricmp(FullName, FULL_NAME_FOR_DAY_OF_WEEK[DayOfWeek]) == 0)

```



```

    #elif __linux__
    if (strcasecmp(FullName, FULL_NAME_FOR_DAY_OF_WEEK[DayOfWeek]) == 0)
    #else
    if (strcmp(FullName, FULL_NAME_FOR_DAY_OF_WEEK[DayOfWeek]) == 0)
    #endif
        return static_cast<DayOfWeekEnum>(DayOfWeek);
}

return DAY_OF_WEEK_NOT_SET;
}

/*=====*/
inline DayOfWeekEnum GetDayOfWeekFromShortName(const char* ShortName)
{
    if (ShortName == NULL)
        return DAY_OF_WEEK_NOT_SET;

    for (int DayOfWeek = SUNDAY; DayOfWeek <= SATURDAY; ++DayOfWeek)
    {
        #if _MSC_VER >= 1400
        if (_strcmp(ShortName, SHORT_NAME_FOR_DAY_OF_WEEK[DayOfWeek]) == 0)
        #elif __linux__
        if (strcasecmp(ShortName, SHORT_NAME_FOR_DAY_OF_WEEK[DayOfWeek]) == 0)
        #else
        if (strcmp(ShortName, SHORT_NAME_FOR_DAY_OF_WEEK[DayOfWeek]) == 0)
        #endif
            return static_cast<DayOfWeekEnum>(DayOfWeek);
    }

    return DAY_OF_WEEK_NOT_SET;
}

/*=====*/
DEPRECATED("The IsSaturday is deprecated. Use SCDatetime::IsSaturday() instead.")
inline bool IsSaturday(int Date)
{
    return (DAY_OF_WEEK(Date) == SATURDAY);
}

/*=====*/
DEPRECATED("The IsSaturday is deprecated. Use SCDatetime::IsSaturday() instead.")
inline bool IsSaturday(double DateTime)
{
    return (DAY_OF_WEEK(DateTime) == SATURDAY);
}

/*=====*/
DEPRECATED("The IsSunday is deprecated. Use SCDatetime::IsSunday() instead.")
inline bool IsSunday(int Date)
{
    return (DAY_OF_WEEK(Date) == SUNDAY);
}

/*=====*/
DEPRECATED("The IsSunday is deprecated. Use SCDatetime::IsSunday() instead.")
inline bool IsSunday(double DateTime)
{
    return (DAY_OF_WEEK(DateTime) == SUNDAY);
}

/*=====*/
DEPRECATED("The IsWeekend is deprecated. Use SCDatetime::IsWeekend() instead.")
inline bool IsWeekend(int Date)
{
    const int DayOfWeek = DAY_OF_WEEK(Date);

```

```

    return (DayOfWeek == SUNDAY || DayOfWeek == SATURDAY);
}

/*=====*/
DEPRECATED("The IsWeekend is deprecated. Use SCDatetime::IsWeekend() instead.")
inline bool IsWeekend(double DateTime)
{
    const int DayOfWeek = DAY_OF_WEEK(DateTime);

    return (DayOfWeek == SUNDAY || DayOfWeek == SATURDAY);
}

/*=====*/
inline bool IsWeekend(int Date, bool UseSaturdayData)
{
    const int DayOfWeek = DAY_OF_WEEK(Date);

    return (DayOfWeek == SUNDAY || (!UseSaturdayData && DayOfWeek == SATURDAY));
}

/*=====*/
inline bool IsWeekend(double DateTime, bool UseSaturdayData)
{
    const int DayOfWeek = DAY_OF_WEEK(DateTime);

    return (DayOfWeek == SUNDAY || (!UseSaturdayData && DayOfWeek == SATURDAY));
}

/*=====*/
inline bool IsBusinessDay(int Date)
{
    return !IsWeekend(Date);
}

/*=====*/
inline bool IsLeapYear(int Year)
{
    return ((Year & 3) == 0) && ((Year % 100) != 0 || (Year % 400) == 0);
}

/*=====*/
inline int GetDaysInMonth(int Month, int Year)
{
    if (Month < JANUARY || Month > DECEMBER)
        return 0;

    if (Month == FEBRUARY && IsLeapYear(Year))
        return DAYS_IN_MONTH[Month] + 1;
    else
        return DAYS_IN_MONTH[Month];
}

/*=====
This function returns the number of weekdays within the given date span,
including both the FirstDate and the LastDate. Neither Saturdays nor
Sundays are counted.
=====*/
inline int DaysInDateSpanNotIncludingWeekends(int FirstDate, int LastDate)
{
    if (FirstDate > LastDate)
    {
        // Swap the dates
        const int EarlierDate = LastDate;
        LastDate = FirstDate;
    }
}

```

```

    FirstDate = EarlierDate;
}

const int SpanDifference = LastDate - FirstDate;
const int FirstDayOfWeek = DAY_OF_WEEK(FirstDate);
const int SundayBasedSpan = SpanDifference + FirstDayOfWeek;

int NumSaturdays;
if (SundayBasedSpan - SATURDAY < 0)
    NumSaturdays = 0;
else
    NumSaturdays = (SundayBasedSpan - SATURDAY) / DAYS_PER_WEEK + 1;

int NumSundays = SundayBasedSpan / DAYS_PER_WEEK;
if (FirstDayOfWeek == SUNDAY)
    ++NumSundays;

return SpanDifference + 1 - NumSaturdays - NumSundays;
}

/*=====
SCDateTime values are the same as Excel DateTime values. These are
stored as a floating point value (double) representing the number of days
since 1899-12-30 00:00:00. (Time is represented as a fraction of a day.)
Unix-style timestamps are stored as integer values representing the
number of seconds since 1970-01-01 00:00:00. These two functions are
used to convert from one to the other.
-----*/
DEPRECATED("The SCDateTimeDoubleToUNIXTime is deprecated. Use SCDateTime::ToUNIXTime() instead.")
inline time_t SCDateTimeDoubleToUNIXTime(double DateTimeDouble)
{
    if (IS_UNSET(DateTimeDouble))
        return 0;

    return (time_t)ADJUST((DateTimeDouble - SCDATETIME_UNIX_EPOCH) * SECONDS_PER_DAY);
}

/*=====
DEPRECATED("The SCDateTimeToFloatUNIXTime is deprecated. Use SCDateTime::ToFloatUNIXTime() instead.")
inline double SCDateTimeToFloatUNIXTime(double DateTimeDouble)
{
    if (IS_UNSET(DateTimeDouble))
        return DATETIME_UNSET_DOUBLE;

    return ((DateTimeDouble - SCDATETIME_UNIX_EPOCH) * SECONDS_PER_DAY);
}

/*=====
DEPRECATED("The SCDateTimeDoubleToSCDateTime is private and deprecated")
inline int64_t SCDateTimeDoubleToSCDateTime(const double DateTime)
{
    if (IS_UNSET(DateTime))
        return DATETIME_UNSET;

    return static_cast<int64_t>(ADJUST(DateTime * MICROSECONDS_PER_DAY));
}

/*=====
inline double SCUNIXTimeToSCDateTime(const int64_t SCUnixTime)
{
    return double(SCUnixTime) / MICROSECONDS_PER_DAY;
}

/*=====
// UNIXTime is in seconds.
DEPRECATED("The UNIXTimeToSCDateTime is deprecated. Use SCDateTime::SetUNIXTime() instead.")
inline double UNIXTimeToSCDateTime(time_t UnixTime)

```

```

{
    if (UnixTime == 0)
        return DATETIME_UNSET_DOUBLE;

    return (UnixTime) / static_cast<double>(SECONDS_PER_DAY) + SCDATETIME_UNIX_EPOCH;
}

/*=====*/
DEPRECATED("The UNIXTimeToSCDateTimeUNIX is deprecated. Use SCDateTime::SetUNIXTime() instead.")
inline int64_t UNIXTimeToSCDateTimeUNIX(time_t UnixTime)
{
    if (UnixTime == 0)
        return DATETIME_UNSET;

    return UnixTime * MICROSECONDS_PER_SECOND + SCDATETIME_UNIX_EPOCH_DATE *
MICROSECONDS_PER_DAY;
}

/*=====*/
DEPRECATED("The UNIXTimeWithMillisecondsToSCDateTime is deprecated. Use
SCDateTime::SetUNIXTimeWithMilliseconds() instead.")
inline double UNIXTimeWithMillisecondsToSCDateTime(double UnixTimeWithMilliseconds)
{
    if (UnixTimeWithMilliseconds == 0.0)
        return DATETIME_UNSET_DOUBLE;

    return UnixTimeWithMilliseconds / SECONDS_PER_DAY + SCDATETIME_UNIX_EPOCH;
}

/*=====*/
DEPRECATED("The UNIXTimeWithMillisecondsToSCDateTimeUNIX is deprecated. Use
SCDateTime::SetUNIXTimeWithMilliseconds() instead.")
inline int64_t UNIXTimeWithMillisecondsToSCDateTimeUNIX(double UnixTimeWithMilliseconds)
{
    if (UnixTimeWithMilliseconds == 0.0)
        return DATETIME_UNSET;

    return static_cast<int64_t>(ADJUST(UnixTimeWithMilliseconds * MICROSECONDS_PER_SECOND)) +
SCDATETIME_UNIX_EPOCH_DATE * MICROSECONDS_PER_DAY;
}

/*=====*/
inline int64_t HMS_MS_DATETIME(int Hour, int Minute, int Second, int Millisecond)
{
    return Hour * MICROSECONDS_PER_HOUR
        + Minute * MICROSECONDS_PER_MINUTE
        + Second * MICROSECONDS_PER_SECOND
        + Millisecond * MICROSECONDS_PER_MILLISECOND;
}

/*=====
This function expects one based values for Month and Day.
=====*/
inline int64_t YMDHMS_MS_DATETIME(int Year, int Month, int Day, int Hour, int Minute, int Second, int Millisecond)
{
    // Validate year and month (ignore day of week)
    if (Year > 9999 || Year < -9999 || Month < 1 || Month > 12)
        return DATETIME_UNSET;

    // Check for leap year and set the number of days in the month
    const bool bLeapYear = IsLeapYear(Year);

    const int nDaysInMonth
        = DAYS_IN_YEAR_AT_MONTH_END[Month] - DAYS_IN_YEAR_AT_MONTH_END[Month - 1]
        + ((bLeapYear && Day == 29 && Month == 2) ? 1 : 0);

```

```

// Finish validating the date
if ( Day < 1 || Day > nDaysInMonth
)
{
    return DATETIME_UNSET;
}

//It is a valid date; make Jan 1, 1AD be 1
int nDate = DaysSince1Jan0AtYearBeginning(Year)
    + DAYS_IN_YEAR_AT_MONTH_END[Month - 1]
    + Day;

// If leap year and it's before March, subtract 1:
if (Month <= 2 && bLeapYear)
    --nDate;

// Offset so that 12/30/1899 is 0
nDate -= DAYS_BEFORE_SC_EPOCH;

const int64_t Time = HMS_MS_DATETIME(Hour, Minute, Second, Millisecond);

return nDate * MICROSECONDS_PER_DAY + ((nDate >= 0) ? Time : -Time);
}

/*=====*/
DEPRECATED("The YMDHMS_DATETIME is deprecated. Use SCDatetime::GetDataYMD() instead.")
inline int64_t YMDHMS_DATETIME(int Year, int Month, int Day, int Hour, int Minute, int Second)
{
    return YMDHMS_MS_DATETIME(Year, Month, Day, Hour, Minute, Second, 0);
}

/*=====
This function is thread safe.
-----*/
inline void DATETIME_TO_YMDHMS_MS(int64_t DateTime, int& Year, int& Month, int& Day, int& Hour, int& Minute,
int& Second, int& MillisecondOrMicrosecond, const bool GetMicroseconds = false)
{
    Year = Month = Day = Hour = Minute = Second = 0;

    int n400Years;    // Number of 400 year increments since 1/1/0
    int n400Century;  // Century within 400 year block (0,1,2 or 3)
    int n4Years;      // Number of 4 year increments since 1/1/0
    int n4Day;         // Day within 4 year block
                    // (0 is 1/1/yr1, 1460 is 12/31/yr4)
    int n4Yr;          // Year within 4 year block (0,1,2 or 3)
    int bLeap4 = 1;    // TRUE if 4 year block includes leap year

    int64_t TempDataTime = DateTime; // temporary serial date

    if (DateTime != DATETIME_UNSET)
    {
        // If resolving for second fractions
        if (MillisecondOrMicrosecond != -1)
        {
            // DateTime is rounded to microsecond by default
            if (!GetMicroseconds)
            {
                // Round to the millisecond.
                TempDataTime = ADJUST_ROUND_MS(DateTime);
            }
        }
    }
    else
    {
        // Round to the second

```

```

    TempDateTime = ADJUST_ROUND_SECOND(DateTime);
}
}

MillisecondOrMicrosecond = 0;

// Number of days from 1970-01-01
const int nAllDays = static_cast<int>(TempDateTime / MICROSECONDS_PER_DAY);

int nDaysAbsolute = nAllDays + DAYS_BEFORE_SC_EPOCH; // Add days from 1/1/0 to 12/30/1899

TempDateTime = abs(TempDateTime);
int64_t Time = TempDateTime % MICROSECONDS_PER_DAY;

const int NumberOfDaysin400Years = DaysSince1Jan0AtYearBeginning(400);

// Leap years every 4 yrs except centuries not multiples of 400.
n400Years = nDaysAbsolute / NumberOfDaysin400Years;

// Set nDaysAbsolute to day within 400-year block
nDaysAbsolute %= NumberOfDaysin400Years;

// -1 because first century has extra day
n400Century = ((nDaysAbsolute - 1) / 36524);

// Non-leap century
if (n400Century != 0)
{
    // Set nDaysAbsolute to day within century
    nDaysAbsolute = (nDaysAbsolute - 1) % 36524;

    // +1 because 1st 4 year increment has 1460 days
    n4Years = ((nDaysAbsolute + 1) / 1461);

    if (n4Years != 0)
        n4Day = ((nDaysAbsolute + 1) % 1461);
    else
    {
        bLeap4 = 0;
        n4Day = nDaysAbsolute;
    }
}
else
{
    // Leap century - not special case!
    n4Years = nDaysAbsolute / 1461;
    n4Day = nDaysAbsolute % 1461;
}

if (bLeap4)
{
    // -1 because first year has 366 days
    n4Yr = (n4Day - 1) / 365;

    if (n4Yr != 0)
        n4Day = (n4Day - 1) % 365;
}
else
{
    n4Yr = n4Day / 365;
    n4Day %= 365;
}

// n4Day is now 0-based day of year. Save 1-based day of year, year number
Year = n400Years * 400 + n400Century * 100 + n4Years * 4 + n4Yr;

```

```

do
{
    // Handle leap year: before, on, and after Feb. 29.
    if (n4Yr == 0 && bLeap4)
    {
        // Leap Year
        if (n4Day == 59)
        {
            /* Feb. 29 */
            Month = 2;
            Day = 29;

            break;
        }

        // Pretend it's not a leap year for month/day comp.
        if (n4Day >= 60)
            --n4Day;
    }

    // Make n4DaY a 1-based day of non-leap year and compute
    // month/day for everything but Feb. 29.
    ++n4Day;

    // Month number always >= n/32, so save some loop time */
    for (Month = (n4Day >> 5) + 1;
        n4Day > DAYS_IN_YEAR_AT_MONTH_END[Month]; Month++);

    Day = (n4Day - DAYS_IN_YEAR_AT_MONTH_END[Month - 1]);
} while (false);

if (Time != 0)
{
    Hour = static_cast<int>(Time / MICROSECONDS_PER_HOUR);
    Time -= Hour * MICROSECONDS_PER_HOUR;
    Minute = static_cast<int>(Time / MICROSECONDS_PER_MINUTE);
    Time -= Minute * MICROSECONDS_PER_MINUTE;
    Second = static_cast<int>(Time / MICROSECONDS_PER_SECOND);
    Time -= Second * MICROSECONDS_PER_SECOND;

    if (GetMicroseconds)
        MillisecondOrMicrosecond = static_cast<int>(Time);
    else
        MillisecondOrMicrosecond = static_cast<int>(Time / MICROSECONDS_PER_MILLISECOND);
}
}

/*=====
This function is thread safe.
-----*/
DEPRECATED("The DATETIME_TO_YMDHMS is deprecated. Use SCDatetime::GetDateTimeYMDHMS() instead.")
inline void DATETIME_TO_YMDHMS(int64_t DateTime, int& Year, int& Month, int& Day, int& Hour, int& Minute, int&
Second)
{
    int Millisecond = -1;
    DATETIME_TO_YMDHMS_MS(DateTime, Year, Month, Day, Hour, Minute, Second, Millisecond);
}

DEPRECATED("The DATETIME_TO_YMDHMS is deprecated. Use SCDatetime::GetDateTimeYMDHMS() instead.")
inline void DATETIME_TO_YMDHMS(double DateTime, int& Year, int& Month, int& Day, int& Hour, int& Minute, int&
Second)
{
    DATETIME_TO_YMDHMS(SCDateTimeDoubleToSCDateTime(DateTime), Year, Month, Day, Hour, Minute, Second);
}

```

```

/*=====*/
DEPRECATED("The YMD_DATE is deprecated. Use SCDateTime::SCDateTime(int Year, int Month, int Day, int
Minute, int Second).GetDate() instead.")
inline int YMD_DATE(int Year, int Month, int Day)
{
    return static_cast<int>(YMDHMS_DATETIME(Year, Month, Day, 0, 0, 0) / MICROSECONDS_PER_DAY);
}

/*=====*/
DEPRECATED("The DATE_TO_YMD is deprecated. Use SCDateTime::GetDateYMD() instead.")
inline void DATE_TO_YMD(int Date, int& Year, int& Month, int& Day)
{
    int Hour, Minute, Second;
    DATETIME_TO_YMDHMS(Date * MICROSECONDS_PER_DAY, Year, Month, Day, Hour, Minute, Second);
}

/*=====
Converts hours, minutes, seconds to an integer time. This is seconds
since midnight.
-----*/
DEPRECATED("The HMS_TIME is deprecated. Use SCDateTime(int Hour, int Minute, int Second, int
Millisecond).GetTime() instead.")
inline int HMS_TIME(int Hour, int Minute, int Second)
{
    return (Hour * SECONDS_PER_HOUR + Minute * SECONDS_PER_MINUTE + Second);
}

/*=====*/
inline int HMS_TIME_WITH_VALIDATION(int Hour, int Minute, int Second)
{
    if (Hour < 0 || Hour > 23 || Minute < 0 || Minute > 59 || Second < 0 || Second > 59)
        return 0;

    return HMS_TIME(Hour, Minute, Second);
}

/*=====*/
inline int HOUR_OF_TIME(int Time)
{
    return ((Time / SECONDS_PER_HOUR) % HOURS_PER_DAY);
}

/*=====*/
inline int MINUTE_OF_TIME(int Time)
{
    return ((Time / SECONDS_PER_MINUTE) % MINUTES_PER_HOUR);
}

/*=====*/
inline int SECOND_OF_TIME(int Time)
{
    return (Time % SECONDS_PER_MINUTE);
}

/*=====*/
DEPRECATED("The TIME_TO_HMS is deprecated. Use SCDateTime::GetTimeHMS() instead.")
inline void TIME_TO_HMS(int Time, int& Hour, int& Minute, int& Second)
{
    Hour = HOUR_OF_TIME(Time);
    Minute = MINUTE_OF_TIME(Time);
    Second = SECOND_OF_TIME(Time);
}

/******

```



```
// SCDatetime
```

The SCDatetime object wraps a single 8-byte integer value representing a complete date and time in form of a number of microseconds elapsed from the SC epoch, 30 December 1899 00:00:00.

Comparison operators on this class are designed to compare date-time values (including doubles) within half a second. For best results, at least one of the date-time values should be as close to the exact second as possible, meaning avoid any milliseconds.

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

```
class SCDatetime
{
    private:
        int64_t m_dt; // microseconds

    public:
        SCDatetime();
        #ifdef SCDATETIME_SUPPORT_CHRONO
        template<typename t_Rep, typename t_Period>
            SCDatetime(const std::chrono::duration<t_Rep, t_Period> Duration);
        SCDatetime(const std::chrono::system_clock::time_point TimePoint);
        #endif
        SCDatetime(double DateTime);
        SCDatetime(const SCDatetime& DateTime);
        SCDatetime(int Date, int TimeInSeconds);
        SCDatetime(int Hour, int Minute, int Second, int Millisecond);
        SCDatetime(int Year, int Month, int Day, int Hour, int Minute, int Second);
        SCDatetime(int Year, int Month, int Day, int Hour, int Minute, int Second, int Millisecond);

        void ValidateAsCorrectDateTime();

        const SCDatetime& operator = (double dt);
        const SCDatetime& operator = (const SCDatetime& DateTime);
        const SCDatetime& operator += (double dt);
        const SCDatetime& operator += (const SCDatetime& DateTime);
        const SCDatetime& operator -= (double dt);
        const SCDatetime& operator -= (const SCDatetime& DateTime);

        bool operator == (int Date) const;
        bool operator != (int Date) const;
        bool operator < (int Date) const;
        bool operator <= (int Date) const;
        bool operator > (int Date) const;
        bool operator >= (int Date) const;

        bool operator == (double dt) const;
        bool operator != (double dt) const;
        bool operator < (double dt) const;
        bool operator <= (double dt) const;
        bool operator > (double dt) const;
        bool operator >= (double dt) const;

        bool operator == (const SCDatetime& DateTime) const;
        bool operator != (const SCDatetime& DateTime) const;
        bool operator < (const SCDatetime& DateTime) const;
        bool operator <= (const SCDatetime& DateTime) const;
        bool operator > (const SCDatetime& DateTime) const;
        bool operator >= (const SCDatetime& DateTime) const;

        void Clear();
        void SetToMaximum();
        bool IsMaximum() const;
}
```

```

bool IsUnset() const;
bool IsDateSet() const;
bool IsTimeAtMidnight() const;
void ClearDate();
void SetAsNegative();
int GetDate() const;
int GetDateMS() const;
int GetDateUS() const;
SCDateTime GetDateAsSCDateTime() const;
int GetDays() const;

int GetTime() const;
int GetTimeInSeconds() const;
int GetTimeInMilliseconds() const;
int GetTimeInSecondsWithoutMilliseconds() const;

SCDateTime GetTimeAsSCDateTime() const;
SCDateTime GetTimeAsSCDateTimeMS() const;
double GetTimeAsDouble() const;

SCDateTime GetDateTimeRoundedToSeconds() const;

void GetDateYMD(int& Year, int& Month, int& Day) const;
void GetTimeHMS(int& Hour, int& Minute, int& Second) const;
void GetDateTimeYMDHMS(int& Year, int& Month, int& Day, int& Hour, int& Minute, int& Second) const;
void GetDateTimeYMDHMS_MS(int& Year, int& Month, int& Day, int& Hour, int& Minute, int& Second, int&
Millisecond) const;
void GetDateTimeYMDHMS_US(int& Year, int& Month, int& Day, int& Hour, int& Minute, int& Second, int&
Microsecond) const;
int GetYear() const;
int GetMonth() const;
int GetDay() const;
int GetHour() const;
int GetMinute() const;
int GetSecond() const;
int GetMicrosecond() const;
int GetMillisecond() const;
int GetDayOfWeek() const;
int GetDayOfWeekMondayBased() const;
SCDateTime& SetToPriorSundayForDateTime(const SCDateTime& OriginalDateTime);

// Note: These truncate to the containing unit.
std::int64_t AsMicrosecondsSinceBaseDate() const;
std::int64_t AsMillisecondsSinceBaseDate() const;
std::int64_t AsSecondsSinceBaseDate() const;

// Note: These round to the nearest unit.
std::int64_t GetMillisecondsSinceBaseDate() const;
std::int64_t GetSecondsSinceBaseDate() const;
int GetMinutesSinceBaseDate() const;
int GetHoursSinceBaseDate() const;
int GetDaysSinceBaseDate() const;
int GetWeeksSinceBaseDate() const;
int GetYearsSinceBaseDate() const;

double GetFloatMillisecondsSinceBaseDate() const;
double GetFloatSecondsSinceBaseDate() const;
double GetFloatMinutesSinceBaseDate() const;
double GetFloatHoursSinceBaseDate() const;
double GetFloatDaysSinceBaseDate() const;
double GetFloatWeeksSinceBaseDate() const;
double GetFloatYearsSinceBaseDate() const;

int GetDateDifferenceInDays(const SCDateTime& DateTime) const;

```

```

bool IsNegative() const;

SCDateTime GetAbsoluteValue() const;
double GetAsDouble() const;

bool IsWeekend(const bool UseSaturdayData = false) const;
bool IsSaturday() const;
bool IsSunday() const;
bool IsMonday() const;
bool IsGivenTimeWithinGivenTimeRange(const SCDateTime& DateTimeToCompareTo, const SCDateTime&
TimeWithin);

int GetBusinessDaysBetweenDates(const SCDateTime& FutureDate) const;
bool IsSameDate(const SCDateTime& DateTime) const;

bool IsSameTimeToHour(const SCDateTime& DateTime) const;
bool IsSameTimeToMinute(const SCDateTime& DateTime) const;
bool IsSameTimeToSecond(const SCDateTime& DateTime) const;
bool IsSameTimeToMillisecond(const SCDateTime& DateTime) const;

void RoundDateTimeDownToMinute();
void RoundDateTimeDownToSecond();
void RoundDateTimeDownToMillisecond();
void RoundToNearestSecond();
void RoundToNearestMillisecond();
SCDateTime& SetDateTimeAsDouble(double DateTime);
SCDateTime& SetDateTime(int Date, int Time);
SCDateTime& SetDate(int Date);
SCDateTime& SetDate(const SCDateTime& Date);
SCDateTime& SetTime(int Time);
SCDateTime& SetTimeFromSCDateTime(const SCDateTime& Time);
SCDateTime& SetDateYMD(int Year, int Month, int Day);
SCDateTime& SetTimeHMS(int Hour, int Minute, int Second);
SCDateTime& SetTimeHMS_MS(int Hour, int Minute, int Second, int Millisecond);
SCDateTime& SetTimeHMS_US(int Hour, int Minute, int Second, int Microsecond);
SCDateTime& SetDateTimeYMDHMS(int Year, int Month, int Day, int Hour, int Minute, int Second);
SCDateTime& SetDateTimeYMDHMS_MS(int Year, int Month, int Day, int Hour, int Minute, int Second, int
Millisecond);
SCDateTime& SetDateTimeYMDHMS_US(int Year, int Month, int Day, int Hour, int Minute, int Second, int
Microsecond);
SCDateTime& SetFromUNIXTimeInSeconds(time_t UNIXTime);
SCDateTime& SetUNIXTimeWithMilliseconds(double UNIXTime);

SCDateTime& AddNanoseconds(const std::uint64_t Nanoseconds);
SCDateTime& AddMicroseconds(int64_t Microseconds);
SCDateTime& SubtractMicroseconds(int Microseconds);
SCDateTime& AddMilliseconds(int Milliseconds);
SCDateTime& SubtractMilliseconds(int Milliseconds);
SCDateTime& AddSeconds(int Seconds);
SCDateTime& SubtractSeconds(int Seconds);
SCDateTime& AddMinutes(int Minutes);
SCDateTime& SubtractMinutes(int Minutes);
SCDateTime& AddHours(int Hours);
SCDateTime& SubtractHours(int Hours);

SCDateTime& AddDays(int Days);
SCDateTime& SubtractDays(int Days);
SCDateTime& AddYears(int Years);
SCDateTime& SubtractYears(int Years);
SCDateTime& SetDayToLastDayInMonth();
SCDateTime& MultiplyTime(float Multiplier);

time_t ToUNIXTime() const;
double ToFloatUNIXTime() const;
int64_t ToUNIXTimeInMilliseconds() const;

```

```

int64_t ToUNIXTimeInMicroseconds() const;
double ToUNIXTimeWithMillisecondsFraction() const;

void SetFromUNIXTimeMilliseconds(int64_t UNIXTimeMilliseconds);
void SetFromUNIXTimeMicroseconds(int64_t UNIXTimeMicroseconds);
void SetFromUnixTimeInDays(const int Days);
void SetFromUNIXTimeMillisecondsFraction(double UNIXTimeMillisecondsFraction);

static SCDateTime GetMinimumDate();
static SCDateTime GetMaximumDate();

static inline SCDateTime YEARS(const int Years)
{
    SCDateTime DT;
    DT.GetInternalDateTime() = static_cast<int64_t>(ROUND_US(Years * 365.25 * MICROSECONDS_PER_DAY));

    return DT;
}

static inline SCDateTime WEEKS(const int Weeks)
{
    return DAYS(Weeks * DAYS_PER_WEEK);
}

static inline SCDateTime DAYS(const int Days)
{
    SCDateTime DT;
    DT.GetInternalDateTime() = Days * MICROSECONDS_PER_DAY;

    return DT;
}

static inline SCDateTime HOURS(const int Hours)
{
    SCDateTime DT;
    DT.GetInternalDateTime() = Hours * MICROSECONDS_PER_HOUR;

    return DT;
}

static inline SCDateTime MINUTES(const int Minutes)
{
    SCDateTime DT;
    DT.GetInternalDateTime() = Minutes * MICROSECONDS_PER_MINUTE;

    return DT;
}

static inline SCDateTime SECONDS(const int Seconds)
{
    SCDateTime DT;
    DT.GetInternalDateTime() = Seconds * MICROSECONDS_PER_SECOND;

    return DT;
}

static inline SCDateTime SECONDS_Fraction(const double Seconds)
{
    SCDateTime DT;
    DT.GetInternalDateTime() = static_cast<int64_t>(ROUND_US(Seconds * MICROSECONDS_PER_SECOND));

    return DT;
}

static inline SCDateTime MILLISECONDS(const int Milliseconds)

```

```

{
    SCDatetime DT;
    DT.GetInternalDateTime() = Milliseconds * MICROSECONDS_PER_MILLISECOND;

    return DT;
}

static inline SCDatetime MICROSECONDS(const int64_t Microseconds)
{
    SCDatetime DT;
    DT.GetInternalDateTime() = Microseconds;

    return DT;
}

static inline SCDatetime NANOSECONDS(const int64_t Nanoseconds)
{
    SCDatetime DT;
    DT.GetInternalDateTime() = Nanoseconds / 1000;

    return DT;
}

protected:

    inline int64_t& GetInternalDateTime()
    {
        return m_dt;
    }

public:
    inline int64_t GetInternalDateTime() const
    {
        return m_dt;
    }

    inline void SetInternalDateTime(int64_t InternalDateTimeValue)
    {
        m_dt = InternalDateTimeValue;
    }
};

typedef SCDatetime SCDatetimeMS;

/*****
// c_SCDatetime

/*=====*/
inline SCDatetime::SCDatetime()
    : m_dt(DATETIME_UNSET)
{
}

#ifdef SCDATETIME_SUPPORT_CHRONO
/*=====
    This constructor supports converting from any std::chrono::duration.
    -----*/
template<typename t_Rep, typename t_Period>
inline SCDatetime::SCDatetime
( const std::chrono::duration<t_Rep, t_Period> Duration
)
    : m_dt(std::chrono::duration_cast<std::chrono::microseconds>(Duration).count())
{
}

```

```

/*=====*/
inline SCDateTime::SCDateTime
( const std::chrono::system_clock::time_point TimePoint
)
    : SCDateTime(TimePoint - SCDATETIME_EPOCH_SYSTEM_CLOCK)
{
}
#endif

/*=====*/
inline SCDateTime::SCDateTime(double DateTime)
    : m_dt(SCDateTimeDoubleToSCDateTime(DateTime))
{
}

/*=====*/
inline SCDateTime::SCDateTime(const SCDateTime& DateTime)
    : m_dt(DateTime.m_dt)
{
}

/*=====*/
inline SCDateTime::SCDateTime(int Date, int TimeInSeconds)
    : m_dt(COMBINE_DATE_TIME_US(Date, TimeInSeconds))
{
}

/*=====*/
inline SCDateTime::SCDateTime(int Year, int Month, int Day, int Hour, int Minute, int Second)
    : m_dt(YMDHMS_DATETIME(Year, Month, Day, Hour, Minute, Second))
{
}

/*=====*/
inline SCDateTime::SCDateTime(int Year, int Month, int Day, int Hour, int Minute, int Second, int Millisecond)
    : m_dt(YMDHMS_MS_DATETIME(Year, Month, Day, Hour, Minute, Second, Millisecond))
{
}

/*=====*/
inline SCDateTime::SCDateTime(int Hour, int Minute, int Second, int Millisecond)
    : m_dt(DATETIME_UNSET)
{
    SetTimeHMS_MS(Hour, Minute, Second, Millisecond);
}

/*=====*/
inline void SCDateTime::ValidateAsCorrectDateTime()
{
    if (IsUnset())
        return;

    const char UpperByte = (m_dt >> (sizeof(m_dt) - 1) * 8) & 0xFF;

    // Upper byte between 0x3E and 0x41 means positive double values of DateTime between 1899 and 9999 years
    // Upper byte between 0xBD and 0xC1 means negative double values of DateTime between -9999 and 1899 years
    if ((0x3E <= UpperByte && UpperByte <= 0x41) || (-67 <= UpperByte && UpperByte <= -63))
    {
        *this = SCDateTime(*reinterpret_cast<double*>(&m_dt));
        //Due to the fact that microseconds cannot be precisely represented with SCDateTime as a double, we need to
        remove them. Otherwise, we will have erroneous microsecond values showing up.
        RoundToNearestMillisecond();
    }
}

```

```

/*=====*/
inline const SCDateTime& SCDateTime::operator = (double dt)
{
    *this = SCDateTime(dt);
    return *this;
}

/*=====*/
inline const SCDateTime& SCDateTime::operator = (const SCDateTime& DateTime)
{
    m_dt = DateTime.m_dt;
    return *this;
}

/*=====*/
inline const SCDateTime& SCDateTime::operator += (double dt)
{
    *this += SCDateTime(dt);
    return *this;
}

/*=====*/
inline const SCDateTime& SCDateTime::operator += (const SCDateTime& DateTime)
{
    m_dt += DateTime.m_dt;
    return *this;
}

/*=====*/
inline const SCDateTime& SCDateTime::operator -= (double dt)
{
    *this -= SCDateTime(dt);
    return *this;
}

/*=====*/
inline const SCDateTime& SCDateTime::operator -= (const SCDateTime& DateTime)
{
    m_dt -= DateTime.m_dt;
    return *this;
}

/*=====*/
inline bool SCDateTime::operator == (int Date) const
{
    return *this == SCDateTime(Date, 0);
}

/*=====*/
inline bool SCDateTime::operator != (int Date) const
{
    return *this != SCDateTime(Date, 0);
}

/*=====*/
inline bool SCDateTime::operator < (int Date) const
{
    return *this < SCDateTime(Date, 0);
}

/*=====*/
inline bool SCDateTime::operator <= (int Date) const
{
    return *this <= SCDateTime(Date, 0);
}

```

```

}

/*=====*/
inline bool SCDatetime::operator > (int Date) const
{
    return *this > SCDatetime(Date, 0);
}

/*=====*/
inline bool SCDatetime::operator >= (int Date) const
{
    return *this >= SCDatetime(Date, 0);
}

/*=====*/
inline bool SCDatetime::operator == (double dt) const
{
    return *this == SCDatetime(dt);
}

/*=====*/
inline bool SCDatetime::operator != (double dt) const
{
    return *this != SCDatetime(dt);
}

/*=====*/
inline bool SCDatetime::operator < (double dt) const
{
    return *this < SCDatetime(dt);
}

/*=====*/
inline bool SCDatetime::operator <= (double dt) const
{
    return *this <= SCDatetime(dt);
}

/*=====*/
inline bool SCDatetime::operator > (double dt) const
{
    return *this > SCDatetime(dt);
}

/*=====*/
inline bool SCDatetime::operator >= (double dt) const
{
    return *this >= SCDatetime(dt);
}

/*=====*/
inline bool SCDatetime::operator == (const SCDatetime& DateTime) const
{
    return GetInternalDateTime() == DateTime.GetInternalDateTime();
}

/*=====*/
inline bool SCDatetime::operator != (const SCDatetime& DateTime) const
{
    return !this->operator ==(DateTime);
}

/*=====*/
inline bool SCDatetime::operator < (const SCDatetime& DateTime) const
{

```



```

    return GetInternalDateTime() < DateTime.GetInternalDateTime();
}

/*=====*/
inline bool SCDatetime::operator <= (const SCDatetime& DateTime) const
{
    return !this->operator >(DateTime);
}

/*=====*/
inline bool SCDatetime::operator > (const SCDatetime& DateTime) const
{
    return GetInternalDateTime() > DateTime.GetInternalDateTime();
}

/*=====*/
inline bool SCDatetime::operator >= (const SCDatetime& DateTime) const
{
    return !this->operator <(DateTime);
}

/*=====*/
inline void SCDatetime::Clear()
{
    m_dt = DATETIME_UNSET;
}

/*=====*/
inline void SCDatetime::SetToMaximum()
{
    *this = GetMaximumDate();
}

/*=====*/
inline bool SCDatetime::IsMaximum() const
{
    return GetInternalDateTime() == GetMaximumDate().GetInternalDateTime();
}

/*=====*/
inline bool SCDatetime::IsUnset() const
{
    return m_dt == DATETIME_UNSET;
}

/*=====
Returns true only if the date is not at or before the base date of
1900-01-01.
-----*/
inline bool SCDatetime::IsDateSet() const
{
    return (GetDate() > 0);
}

/*=====*/
inline bool SCDatetime::IsTimeAtMidnight() const
{
    return GetTimeInSeconds() == 0;
}

/*=====*/
inline void SCDatetime::ClearDate()
{
    m_dt %= MICROSECONDS_PER_DAY; // drop date part to keep time part
}

/*=====*/

```

```

inline void SCDatetime::SetAsNegative()
{
    if (m_dt <= 0)
        return;

    m_dt = -m_dt;
}

/*=====*/
inline int SCDatetime::GetDate() const
{
    // Always get the date by rounding only to the nearest microsecond.
    return DATE_PART_MS(GetInternalDateTime());
}

/*=====*/
inline int SCDatetime::GetDateMS() const
{
    return DATE_PART_MS(GetInternalDateTime());
}

/*=====
Returns the date and ignores the time. There is no rounding. If the
time is at the last microsecond of the date, then the same date is
returned.
-----*/
inline int SCDatetime::GetDateUS() const
{
    return static_cast<int>(GetInternalDateTime() / MICROSECONDS_PER_DAY);
}

/*=====*/
inline SCDatetime SCDatetime::GetDateAsSCDateTime() const
{
    return SCDatetime(GetDate(), 0);
}

/*=====*/
inline int SCDatetime::GetDays() const
{
    return GetDate();
}

/*=====*/
inline int SCDatetime::GetTime() const
{
    return TIME_PART_MS(GetInternalDateTime());
}

/*=====*/
inline int SCDatetime::GetTimeInSeconds() const
{
    return GetTime();
}

/*=====*/
inline int SCDatetime::GetTimeInMilliseconds() const
{
    int64_t TimeInMilliseconds = 0;
    int64_t DateTime = ADJUST_ROUND_MS(m_dt);

    DateTime %= MICROSECONDS_PER_DAY; // remove the date part

    TimeInMilliseconds = DateTime / MICROSECONDS_PER_MILLISECOND;

    if (TimeInMilliseconds >= MILLISECONDS_PER_DAY)

```

```

        TimeInMilliseconds = 0;

    return static_cast<int>(TimeInMilliseconds);
}

/*=====*/
inline int SCDateTime::GetTimeInSecondsWithoutMilliseconds() const
{
    return TIME_PART_MS(GetInternalDateTime());
}

/*=====*/
inline SCDateTime SCDateTime::GetTimeAsSCDateTime() const
{
    SCDateTime DateTime = GetAbsoluteValue();
    DateTime.m_dt %= MICROSECONDS_PER_DAY; // remove date part
    return DateTime;
}

/*=====*/
inline SCDateTime SCDateTime::GetTimeAsSCDateTimeMS() const
{
    return GetTimeAsSCDateTime();
}

/*=====*/
inline double SCDateTime::GetTimeAsDouble() const
{
    return GetTimeAsSCDateTime().GetInternalDateTime() / static_cast<double>(MICROSECONDS_PER_DAY);
}

/*=====*/
inline SCDateTime SCDateTime::GetDateTimeRoundedToSeconds() const
{
    SCDateTime RoundedDateTime(*this);

    RoundedDateTime.m_dt = ADJUST_ROUND_SECOND(RoundedDateTime.m_dt);
    RoundedDateTime.m_dt -= RoundedDateTime.m_dt % MICROSECONDS_PER_SECOND;

    return RoundedDateTime;
}

/*=====*/
inline void SCDateTime::GetDateYMD(int& Year, int& Month, int& Day) const
{
    DATE_TO_YMD(GetDate(), Year, Month, Day);
}

/*=====*/
inline void SCDateTime::GetTimeHMS(int& Hour, int& Minute, int& Second) const
{
    TIME_TO_HMS(GetTime(), Hour, Minute, Second);
}

/*=====*/
inline void SCDateTime::GetDateTimeYMDHMS(int& Year, int& Month, int& Day, int& Hour, int& Minute, int& Second)
const
{
    int Millisecond = 0;
    DATETIME_TO_YMDHMS_MS(m_dt, Year, Month, Day, Hour, Minute, Second, Millisecond);
}

/*=====*/
inline void SCDateTime::GetDateTimeYMDHMS_MS(int& Year, int& Month, int& Day, int& Hour, int& Minute, int&
Second, int& Millisecond) const

```

```

{
    DATETIME_TO_YMDHMS_MS(m_dt, Year, Month, Day, Hour, Minute, Second, Millisecond);
}

/*=====*/
inline void SCDatetime::GetDateTimeYMDHMS_US(int& Year, int& Month, int& Day, int& Hour, int& Minute, int&
Second, int& Microsecond) const
{
    DATETIME_TO_YMDHMS_MS(m_dt, Year, Month, Day, Hour, Minute, Second, Microsecond, true);
}

/*=====*/
inline int SCDatetime::GetYear() const
{
    int Year, Month, Day;
    GetDateYMD(Year, Month, Day);
    return Year;
}

/*=====*/
inline int SCDatetime::GetMonth() const
{
    int Year, Month, Day;
    GetDateYMD(Year, Month, Day);
    return Month;
}

/*=====*/
inline int SCDatetime::GetDay() const
{
    int Year, Month, Day;
    GetDateYMD(Year, Month, Day);
    return Day;
}

/*=====*/
inline int SCDatetime::GetHour() const
{
    return HOUR_OF_TIME(GetTime());
}

/*=====*/
inline int SCDatetime::GetMinute() const
{
    return MINUTE_OF_TIME(GetTime());
}

/*=====*/
inline int SCDatetime::GetSecond() const
{
    return SECOND_OF_TIME(GetTime());
}

/*=====*/
inline int SCDatetime::GetMicrosecond() const
{
    return static_cast<int>(GetInternalDateTime() % MICROSECONDS_PER_MILLISECOND);
}

/*=====*/
inline int SCDatetime::GetMillisecond() const
{
    return MS_PART(GetInternalDateTime());
}

```

```

/*=====*/
// Returns DayOfWeekEnum
inline int SCDatetime::GetDayOfWeek() const
{
    return DAY_OF_WEEK(GetDate());
}

/*=====*/
inline int SCDatetime::GetDayOfWeekMondayBased() const
{
    return DAY_OF_WEEK_MONDAY_BASED(GetDate());
}

/*=====*/
inline SCDatetime& SCDatetime::SetToPriorSundayForDateTime(const SCDatetime& OriginalDateTime)
{
    m_dt = OriginalDateTime.GetInternalDateTime();
    SetTime(0);
    SubtractDays(OriginalDateTime.GetDayOfWeek());
    return *this;
}

/*=====*/
inline std::int64_t SCDatetime::AsMicrosecondsSinceBaseDate() const
{
    return m_dt;
}

/*=====*/
inline std::int64_t SCDatetime::AsMillisecondsSinceBaseDate() const
{
    return m_dt / MICROSECONDS_PER_MILLISECOND;
}

/*=====*/
inline std::int64_t SCDatetime::AsSecondsSinceBaseDate() const
{
    return m_dt / MICROSECONDS_PER_SECOND;
}

/*=====*/
inline std::int64_t SCDatetime::GetMillisecondsSinceBaseDate() const
{
    return static_cast<int64_t>(ADJUST(GetFloatMillisecondsSinceBaseDate()));
}

/*=====*/
inline std::int64_t SCDatetime::GetSecondsSinceBaseDate() const
{
    return static_cast<int64_t>(ADJUST(GetFloatSecondsSinceBaseDate()));
}

/*=====*/
inline int SCDatetime::GetMinutesSinceBaseDate() const
{
    return static_cast<int>(ADJUST(GetFloatMinutesSinceBaseDate()));
}

/*=====*/
inline int SCDatetime::GetHoursSinceBaseDate() const
{
    return static_cast<int>(ADJUST(GetFloatHoursSinceBaseDate()));
}

/*=====*/

```

```

inline int SCDatetime::GetDaysSinceBaseDate() const
{
    return static_cast<int>(ADJUST(GetFloatDaysSinceBaseDate()));
}

/*=====*/
inline int SCDatetime::GetWeeksSinceBaseDate() const
{
    return static_cast<int>(ADJUST(GetFloatWeeksSinceBaseDate()));
}

/*=====*/
inline int SCDatetime::GetYearsSinceBaseDate() const
{
    return static_cast<int>(ADJUST(GetFloatYearsSinceBaseDate()));
}

/*=====*/
inline double SCDatetime::GetFloatMillisecondsSinceBaseDate() const
{
    return static_cast<double>(GetInternalDateTime()) / MICROSECONDS_PER_MILLISECOND;
}

/*=====*/
inline double SCDatetime::GetFloatSecondsSinceBaseDate() const
{
    return static_cast<double>(GetInternalDateTime()) / MICROSECONDS_PER_SECOND;
}

/*=====*/
inline double SCDatetime::GetFloatMinutesSinceBaseDate() const
{
    return static_cast<double>(GetInternalDateTime()) / MICROSECONDS_PER_MINUTE;
}

/*=====*/
inline double SCDatetime::GetFloatHoursSinceBaseDate() const
{
    return static_cast<double>(GetInternalDateTime()) / MICROSECONDS_PER_HOUR;
}

/*=====*/
inline double SCDatetime::GetFloatDaysSinceBaseDate() const
{
    return static_cast<double>(GetInternalDateTime()) / MICROSECONDS_PER_DAY;
}

/*=====*/
inline double SCDatetime::GetFloatWeeksSinceBaseDate() const
{
    return static_cast<double>(GetInternalDateTime()) / MICROSECONDS_PER_DAY / DAYS_PER_WEEK;
}

/*=====*/
inline double SCDatetime::GetFloatYearsSinceBaseDate() const
{
    return static_cast<double>(GetInternalDateTime()) / MICROSECONDS_PER_DAY / DAYS_IN_YEAR;
}

/*=====*/
inline int SCDatetime::GetDateDifferenceInDays(const SCDatetime& DateTime) const
{
    return GetDate() - DateTime.GetDate();
}

```

```

/*=====*/
inline bool SCDatetime::IsNegative() const
{
    return m_dt < 0;
}

/*=====*/
inline SCDatetime SCDatetime::GetAbsoluteValue() const
{
    SCDatetime DateTime = *this;
    DateTime.m_dt = abs(DateTime.m_dt);
    return DateTime;
}

/*=====*/
inline double SCDatetime::GetAsDouble() const
{
    return SCUNIXTimeToSCdatetime(m_dt);
}

/*=====*/
// When UseSaturdayData is true, then Saturday data is not considered part of the weekend.
inline bool SCDatetime::IsWeekend(const bool UseSaturdayData) const
{
    const int DayOfWeek = GetDayOfWeek();

    return (!UseSaturdayData && DayOfWeek == SATURDAY) || DayOfWeek == SUNDAY;
}

/*=====*/
inline bool SCDatetime::IsSaturday() const
{
    return (GetDayOfWeek() == SATURDAY);
}

/*=====*/
inline bool SCDatetime::IsSunday() const
{
    return (GetDayOfWeek() == SUNDAY);
}

/*=====*/
inline bool SCDatetime::IsMonday() const
{
    return (GetDayOfWeek() == MONDAY);
}

/*=====*/
inline bool SCDatetime::IsGivenTimeWithinGivenTimeRange(const SCDatetime& DateTimeToCompareTo, const SCDatetime& TimeWithin)
{
    return abs(GetInternalDateTime() - DateTimeToCompareTo.GetInternalDateTime()) <= TimeWithin.GetInternalDateTime();
}

/*=====*/
inline int SCDatetime::GetBusinessDaysBetweenDates(const SCDatetime& FutureDate) const
{
    if (*this >= FutureDate)
        return 0;

    int NumberOfBusinessDays = 0;
    const int StartDate = GetDate();
    const int EndDate = FutureDate.GetDate();

```

```

for (int DateValue = StartDate; DateValue < EndDate; DateValue++)
{
    const SCDateTime BusinessDayCheck(DateValue, 0);
    if (BusinessDayCheck.IsWeekend())
        continue;

    NumberOfBusinessDays++;
}

return NumberOfBusinessDays;
}

/*=====*/
inline bool SCDateTime::IsSameDate(const SCDateTime& DateTime) const
{
    return GetDate() == DateTime.GetDate();
}

/*=====
Returns true only if the given DateTime is at exactly the same date and
hour as this SCDateTime. Any minutes, seconds, or smaller components are
excluded from the comparison.
=====*/
inline bool SCDateTime::IsSameTimeToHour(const SCDateTime& DateTime) const
{
    const int64_t ThisHour = GetInternalDateTime() / MICROSECONDS_PER_HOUR;
    const int64_t ThatHour = DateTime.GetInternalDateTime() / MICROSECONDS_PER_HOUR;

    return (ThatHour == ThisHour);
}

/*=====
Returns true only if the given DateTime is at exactly the same date, hour,
and minute as this SCDateTime. Any seconds, milliseconds, or smaller
components are excluded from the comparison.
=====*/
inline bool SCDateTime::IsSameTimeToMinute(const SCDateTime& DateTime) const
{
    const int64_t ThisMinute = GetInternalDateTime() / MICROSECONDS_PER_MINUTE;
    const int64_t ThatMinute = DateTime.GetInternalDateTime() / MICROSECONDS_PER_MINUTE;

    return (ThatMinute == ThisMinute);
}

/*=====
Returns true only if the given DateTime is at exactly the same date, hour,
minute, and second as this SCDateTime. Any milliseconds, or smaller
components are excluded from the comparison.
=====*/
inline bool SCDateTime::IsSameTimeToSecond(const SCDateTime& DateTime) const
{
    const int64_t ThisSecond = GetInternalDateTime() / MICROSECONDS_PER_SECOND;
    const int64_t ThatSecond = DateTime.GetInternalDateTime() / MICROSECONDS_PER_SECOND;

    return (ThatSecond == ThisSecond);
}

/*=====
Returns true only if the given DateTime is at exactly the same date, hour,
minute, second, and millisecond as this SCDateTime. Any smaller
components are excluded from the comparison.
=====*/
inline bool SCDateTime::IsSameTimeToMillisecond(const SCDateTime& DateTime) const
{
    const int64_t ThisMillisecond = GetInternalDateTime() / MICROSECONDS_PER_MILLISECOND;

```



```

    const int64_t ThatMillisecond = DateTime.GetInternalDateTime() / MICROSECONDS_PER_MILLISECOND;

    return (ThatMillisecond == ThisMillisecond);
}

/*=====*/
inline void SCDatetime::RoundDateTimeDownToMinute()
{
    m_dt -= m_dt % MICROSECONDS_PER_MINUTE;
}

/*=====*/
inline void SCDatetime::RoundDateTimeDownToSecond()
{
    m_dt -= m_dt % MICROSECONDS_PER_SECOND;
}

/*=====*/
inline void SCDatetime::RoundDateTimeDownToMillisecond()
{
    m_dt -= m_dt % MICROSECONDS_PER_MILLISECOND;
}

/*=====*/
inline void SCDatetime::RoundToNearestSecond()
{
    m_dt = ADJUST_ROUND_SECOND(GetInternalDateTime());
    m_dt -= m_dt % MICROSECONDS_PER_SECOND;
}

/*=====*/
inline void SCDatetime::RoundToNearestMillisecond()
{
    m_dt = ADJUST_ROUND_MS(GetInternalDateTime());
    m_dt -= m_dt % MICROSECONDS_PER_MILLISECOND;
}

/*=====*/
inline SCDatetime& SCDatetime::SetDateTimeAsDouble(double DateTime)
{
    m_dt = SCDatetimeDoubleToSCDateTime(DateTime);
    return *this;
}

/*=====*/
inline SCDatetime& SCDatetime::SetDateTime(int Date, int Time)
{
    m_dt = Date * MICROSECONDS_PER_DAY + Time * MICROSECONDS_PER_SECOND;
    return *this;
}

/*=====*/
inline SCDatetime& SCDatetime::SetDate(int Date)
{
    m_dt %= MICROSECONDS_PER_DAY; // drop date part to keep time part
    m_dt += Date * MICROSECONDS_PER_DAY; // add date part
    return *this;
}

/*=====*/
inline SCDatetime& SCDatetime::SetDate(const SCDatetime& Date)
{
    m_dt %= MICROSECONDS_PER_DAY; // drop date part to keep time part
    m_dt += Date.m_dt - Date.m_dt % MICROSECONDS_PER_DAY; // add date part
}

```

```

    return *this;
}

/*=====*/
inline SCDatetime& SCDatetime::SetTime(int Time)
{
    m_dt += Time * MICROSECONDS_PER_SECOND - m_dt % MICROSECONDS_PER_DAY;
    return *this;
}

/*=====*/
inline SCDatetime& SCDatetime::SetTimeFromSCDatetime(const SCDatetime& Time)
{
    m_dt += Time.m_dt % MICROSECONDS_PER_DAY - m_dt % MICROSECONDS_PER_DAY;
    return *this;
}

/*=====*/
inline SCDatetime& SCDatetime::SetDateYMD(int Year, int Month, int Day)
{
    return SetDate(YMD_DATE(Year, Month, Day));
}

/*=====*/
inline SCDatetime& SCDatetime::SetTimeHMS(int Hour, int Minute, int Second)
{
    return SetTime(HMS_TIME(Hour, Minute, Second));
}

/*=====*/
inline SCDatetime& SCDatetime::SetTimeHMS_MS(int Hour, int Minute, int Second, int Millisecond)
{
    SetTimeHMS(Hour, Minute, Second);
    m_dt += (Millisecond * MICROSECONDS_PER_MILLISECOND);
    return *this;
}

/*=====*/
inline SCDatetime& SCDatetime::SetTimeHMS_US(int Hour, int Minute, int Second, int Microsecond)
{
    SetTimeHMS(Hour, Minute, Second);
    AddMicroseconds(Microsecond);

    return *this;
}

/*=====*/
inline SCDatetime& SCDatetime::SetDateTimeYMDHMS(int Year, int Month, int Day, int Hour, int Minute, int Second)
{
    m_dt = YMDHMS_DATETIME(Year, Month, Day, Hour, Minute, Second);
    return *this;
}

/*=====*/
inline SCDatetime& SCDatetime::SetDateTimeYMDHMS_MS(int Year, int Month, int Day, int Hour, int Minute, int
Second, int Millisecond)
{
    m_dt = YMDHMS_MS_DATETIME(Year, Month, Day, Hour, Minute, Second, Millisecond);
    return *this;
}

/*=====*/
inline SCDatetime& SCDatetime::SetDateTimeYMDHMS_US(int Year, int Month, int Day, int Hour, int Minute, int
Second, int Microsecond)
{

```

```

        m_dt = YMDHMS_DATETIME(Year, Month, Day, Hour, Minute, Second);
        AddMicroseconds(Microsecond);

    return *this;
}

/*=====*/
inline SCDateTime& SCDateTime::SetFromUNIXTimeInSeconds(time_t UNIXTime)
{
    m_dt = UNIXTimeToSCDateTimeUNIX(UNIXTime);
    return *this;
}

/*=====*/
inline SCDateTime& SCDateTime::SetUNIXTimeWithMilliseconds(double UNIXTimeWithMilliseconds)
{
    m_dt = UNIXTimeWithMillisecondsToSCDateTimeUNIX(UNIXTimeWithMilliseconds);
    return *this;
}

/*=====*/
inline SCDateTime& SCDateTime::AddMicroseconds(int64_t Microseconds)
{
    m_dt += Microseconds;

    return *this;
}

/*=====*/
inline SCDateTime& SCDateTime::SubtractMicroseconds(int Microseconds)
{
    m_dt -= Microseconds;

    return *this;
}

/*=====*/
inline SCDateTime& SCDateTime::AddMilliseconds(int Milliseconds)
{
    m_dt += Milliseconds * MICROSECONDS_PER_MILLISECOND;
    return *this;
}

/*=====*/
inline SCDateTime& SCDateTime::SubtractMilliseconds(int Milliseconds)
{
    m_dt -= Milliseconds * MICROSECONDS_PER_MILLISECOND;
    return *this;
}

/*=====*/
inline SCDateTime& SCDateTime::AddSeconds(int Seconds)
{
    m_dt += Seconds * MICROSECONDS_PER_SECOND;
    return *this;
}

/*=====*/
inline SCDateTime& SCDateTime::SubtractSeconds(int Seconds)
{
    m_dt -= Seconds * MICROSECONDS_PER_SECOND;
    return *this;
}

/*=====*/

```

```

inline SCDatetime& SCDatetime::AddMinutes(int Minutes)
{
    m_dt += Minutes * MICROSECONDS_PER_MINUTE;

    return *this;
}
/*=====*/
inline SCDatetime& SCDatetime::SubtractMinutes(int Minutes)
{
    m_dt -= Minutes * MICROSECONDS_PER_MINUTE;
    return *this;
}
/*=====*/
inline SCDatetime& SCDatetime::AddHours(int Hours)
{
    m_dt += Hours * MICROSECONDS_PER_HOUR;
    return *this;
}
/*=====*/
inline SCDatetime& SCDatetime::SubtractHours(int Hours)
{
    m_dt -= Hours * MICROSECONDS_PER_HOUR;
    return *this;
}
/*=====*/
inline SCDatetime& SCDatetime::AddDays(int Days)
{
    m_dt += Days * MICROSECONDS_PER_DAY;
    return *this;
}
/*=====*/
inline SCDatetime& SCDatetime::SubtractDays(int Days)
{
    m_dt -= Days * MICROSECONDS_PER_DAY;
    return *this;
}
/*=====*/
inline SCDatetime& SCDatetime::AddYears(int Years)
{
    int CurrentYear = 0;
    int CurrentMonth = 0;
    int CurrentDay = 0;
    int CurrentHour = 0;
    int CurrentMinute = 0;
    int CurrentSecond = 0;
    int CurrentMicrosecond = 0;

    GetDateTimeYMDHMS_US(CurrentYear, CurrentMonth, CurrentDay, CurrentHour, CurrentMinute, CurrentSecond,
CurrentMicrosecond);

    CurrentYear += Years;

    SetDateTimeYMDHMS_US(CurrentYear, CurrentMonth, CurrentDay, CurrentHour, CurrentMinute, CurrentSecond,
CurrentMicrosecond);

    return *this;
}
/*=====*/
inline SCDatetime& SCDatetime::SubtractYears(int Years)
{
    return AddYears(-Years);
}

```

```

}

/*=====*/
inline SCDatetime& SCDatetime::SetDayToLastDayInMonth()
{
    int Year, Month, Day;
    GetDateYMD(Year, Month, Day);

    Day = GetDaysInMonth(Month, Year);

    SetDateYMD(Year, Month, Day);
    return *this;
}

/*=====*/
inline SCDatetime& SCDatetime::MultiplyTime(float Multiplier)
{
    m_dt = static_cast<int64_t>(ADJUST(m_dt * static_cast<double>(Multiplier)));
    return *this;
}

/*=====*/
inline time_t SCDatetime::ToUNIXTime() const
{
    if (IsUnset())
        return 0;

    return (time_t)ADJUST(ToFloatUNIXTime());
}

/*=====*/
inline double SCDatetime::ToFloatUNIXTime() const
{
    if (IsUnset())
        return 0.0;

    return (m_dt - SCDATETIME_UNIX_EPOCH_DATE * MICROSECONDS_PER_DAY) / static_cast<double>(MICROSECONDS_PER_SECOND);
}

/*=====*/
inline int64_t SCDatetime::ToUNIXTimeInMilliseconds() const
{
    if (IsUnset())
        return 0;

    return static_cast<int64_t>(ADJUST((m_dt - SCDATETIME_UNIX_EPOCH_DATE * MICROSECONDS_PER_DAY) / static_cast<double>(MICROSECONDS_PER_MILLISECOND)));
}

/*=====*/
inline int64_t SCDatetime::ToUNIXTimeInMicroseconds() const
{
    if (IsUnset())
        return DATETIME_UNSET;

    return m_dt - SCDATETIME_UNIX_EPOCH_DATE * MICROSECONDS_PER_DAY;
}

/*=====*/
inline double SCDatetime::ToUNIXTimeWithMillisecondsFraction() const
{
    if (IsUnset())
        return DATETIME_UNSET_DOUBLE;
}

```

```

    return ToUNIXTimeInMilliseconds() / static_cast<double>(MILLISECONDS_PER_SECOND);
}

/*=====*/
inline void SCDatetime::SetFromUNIXTimeMilliseconds(int64_t UNIXTimeMilliseconds)
{
    if (IS_UNSET(UNIXTimeMilliseconds))
        m_dt = DATETIME_UNSET;
    else
        m_dt = UNIXTimeMilliseconds * MICROSECONDS_PER_MILLISECOND + SCDATETIME_UNIX_EPOCH_DATE *
MICROSECONDS_PER_DAY;
}

/*=====*/
inline void SCDatetime::SetFromUNIXTimeMicroseconds(int64_t UNIXTimeMicroseconds)
{
    if (IS_UNSET(UNIXTimeMicroseconds))
        m_dt = DATETIME_UNSET;
    else
        m_dt = UNIXTimeMicroseconds + SCDATETIME_UNIX_EPOCH_DATE * MICROSECONDS_PER_DAY;
}

/*=====*/
inline void SCDatetime::SetFromUnixTimeInDays(const int Days)
{
    if (Days != 0)
        m_dt = Days * MICROSECONDS_PER_DAY + SCDATETIME_UNIX_EPOCH_DATE *
MICROSECONDS_PER_DAY;
    else
        m_dt = 0;
}

/*=====*/
inline void SCDatetime::SetFromUNIXTimeMillisecondsFraction(double UNIXTimeMillisecondsFraction)
{
    if (UNIXTimeMillisecondsFraction == 0)
    {
        m_dt = DATETIME_UNSET;
        return;
    }

    m_dt = static_cast<int64_t>(ADJUST(UNIXTimeMillisecondsFraction * MILLISECONDS_PER_SECOND)) *
MICROSECONDS_PER_MILLISECOND + SCDATETIME_UNIX_EPOCH_DATE * MICROSECONDS_PER_DAY;
}

/*=====*/
//Static
inline SCDatetime SCDatetime::GetMinimumDate()
{
    return -0x7FFFFFFFFFFFFFFF / MICROSECONDS_PER_DAY;
}

/*=====*/
//Static
inline SCDatetime SCDatetime::GetMaximumDate()
{
    return 0x7FFFFFFFFFFFFFFF / MICROSECONDS_PER_DAY;
}

/*=====*/
inline std::int64_t SCDatetimeToUNIXTimeRemoveMilliseconds(SCDatetime DateTime)
{
    DateTime.RoundDateTimeDownToSecond();
    return DateTime.ToUNIXTimeInMicroseconds() / MICROSECONDS_PER_SECOND;
}

```

```

/*=====*/
inline SCDatetime operator + (const SCDatetime& DateTimeA, const SCDatetime& DateTimeB)
{
    SCDatetime Result = DateTimeA;
    Result += DateTimeB;

    return Result;
}

/*=====*/
inline SCDatetime operator - (const SCDatetime& DateTimeA, const SCDatetime& DateTimeB)
{
    SCDatetime Result = DateTimeA;
    Result -= DateTimeB;

    return Result;
}

/*=====*/
// Integer comparison operators

/*=====*/
inline bool operator == (int IntDateTime, const SCDatetime& DateTime)
{
    return SCDatetime(IntDateTime, 0) == DateTime;
}

/*=====*/
inline bool operator != (int IntDateTime, const SCDatetime& DateTime)
{
    return SCDatetime(IntDateTime, 0) != DateTime;
}

/*=====*/
inline bool operator < (int IntDateTime, const SCDatetime& DateTime)
{
    return SCDatetime(IntDateTime, 0) < DateTime;
}

/*=====*/
inline bool operator <= (int IntDateTime, const SCDatetime& DateTime)
{
    return SCDatetime(IntDateTime, 0) <= DateTime;
}

/*=====*/
inline bool operator > (int IntDateTime, const SCDatetime& DateTime)
{
    return SCDatetime(IntDateTime, 0) > DateTime;
}

/*=====*/
inline bool operator >= (int IntDateTime, const SCDatetime& DateTime)
{
    return SCDatetime(IntDateTime, 0) >= DateTime;
}

/*=====*/
// Double comparison operators

/*=====*/
inline bool operator == (double dt, const SCDatetime& DateTime)
{
    return SCDatetime(dt) == DateTime;
}

```

```

}

/*=====*/
inline bool operator != (double dt, const SCDatetime& DateTime)
{
    return SCDatetime(dt) != DateTime;
}

/*=====*/
inline bool operator < (double dt, const SCDatetime& DateTime)
{
    return SCDatetime(dt) < DateTime;
}

/*=====*/
inline bool operator <= (double dt, const SCDatetime& DateTime)
{
    return SCDatetime(dt) <= DateTime;
}

/*=====*/
inline bool operator > (double dt, const SCDatetime& DateTime)
{
    return SCDatetime(dt) > DateTime;
}

/*=====*/
inline bool operator >= (double dt, const SCDatetime& DateTime)
{
    return SCDatetime(dt) >= DateTime;
}

/*=====*/
inline SCDatetime GetNextMillisecondForSameSecondDateTime(const SCDatetime& PreviousDateTime, SCDatetime
NextDateTime)
{
    //Purpose: We need to make a new record timestamp unique by using milliseconds.
    const SCDatetime PreviousDateTimeCopy(PreviousDateTime);

    if (PreviousDateTimeCopy.IsSameTimeToSecond(NextDateTime)) //Timestamp to the second is the same.
    {
        const int Milliseconds = PreviousDateTimeCopy.GetMillisecond();

        if (Milliseconds >= MILLISECONDS_PER_SECOND - 1)
            NextDateTime = PreviousDateTimeCopy;
        else
            NextDateTime = PreviousDateTimeCopy + SCDatetime::MILLISECONDS(1);
    }

    return NextDateTime;
}

/*=====*/
inline SCDatetime GetNextMicrosecondForSameMillisecondDateTime_AllowSkippedMicroseconds(const SCDatetime
PreviousDateTime, SCDatetime NextDateTime)
{
    //Purpose: We need to make a new record timestamp unique by using microseconds.

    if (PreviousDateTime.IsSameTimeToMillisecond(NextDateTime) //Timestamp to the millisecond is the same.
        && NextDateTime <= PreviousDateTime) // Skip this if NextDateTime is greater than PreviousDateTime
    {
        const int Microseconds = PreviousDateTime.GetMicrosecond();

        if (Microseconds >= MICROSECONDS_PER_MILLISECOND - 1)
            NextDateTime = PreviousDateTime;
    }
}

```



```

    else
        NextDateTime = PreviousDateTime + SCDatetime::MICROSECONDS(1);
    }

    return NextDateTime;
}

/*=====*/
// Purpose: We need to make a new record timestamp unique by using microseconds.
inline SCDatetime GetNextMicrosecondForSameMillisecondDateTime(const SCDatetime& PreviousDateTime,
SCDatetime NextDateTime)
{
    if (PreviousDateTime.IsSameTimeToMillisecond(NextDateTime)) //Timestamp to the millisecond is the same.
    {
        NextDateTime = PreviousDateTime;

        const int Microseconds = PreviousDateTime.GetMicrosecond();
        if (Microseconds < MICROSECONDS_PER_MILLISECOND - 1)
            NextDateTime.AddMicroseconds(1);
    }

    return NextDateTime;
}

```

```

/*****
// SCDatetimeSpan

```

The SCDatetimeSpan object wraps a int64_t value representing a time span between two SCDatetime objects.

```

-----*/
class SCDatetimeSpan
{
public:
    SCDatetime m_TimeSpan;

    SCDatetimeSpan SetTimeSpan(const SCDatetime& FirstDateTime, const SCDatetime& SecondDateTime)
    {
        m_TimeSpan = FirstDateTime;
        m_TimeSpan -= SecondDateTime;

        return *this;
    }

    SCDatetime GetTimeSpan() const
    {
        return m_TimeSpan;
    }

    SCDatetimeSpan& operator = (double TimeSpan)
    {
        m_TimeSpan = TimeSpan;

        return *this;
    }
};

```

```

#ifdef SHOW_WARNING_IF_DEPRECATED
// Re enable warning outside of the SCDatetime.h file because we want to see warning `deprecated`
#pragma warning(pop)
#endif

```