This chapter describes a set of utility routines that you can use to operate on dates and times. You can use these routines to get and change information about the current date, time, geographic location, time zone, and units of measurement.

The routines described in this chapter return this information in a format that is best suited to the current script. As a result, you can facilitate localization of your application by using these date, time, and measurement utilities.

To understand the material in this chapter, you need to be familiar with the international resources, especially the numeric-format and long-date-format resources, and the Script Manager. These topics are described in *Inside Macintosh: Text*. In addition, the chapter "Text Utilities" in *Inside Macintosh: Text* describes how to convert date and time information into strings of text.

Many of the Date, Time, and Measurement Utilities were previously associated with other managers in the Macintosh system software, and several of these routines have been renamed. Table 4-4 on page 4-33 shows the original names and locations of the modified Date, Time, and Measurement Utilities routines.

The next section provides an introduction to the Date, Time, and Measurement Utilities.

About the Date, Time, and Measurement Utilities

You can use the Date, Time, and Measurement Utilities to manipulate the date-time information and geographic location data used by a Macintosh computer. A Macintosh computer contains a battery-operated ***clock chip*** that maintains

- the current date-time information
- the geographic location and related time-zone information

The date-time information is stored in a 4-byte value located on the clock chip.The geographic location and related time-zone information is stored in extended parameter RAM. For information on extended parameter RAM, see the chapter "Parameter RAM Utilities" in this book.

You can use the routines provided by the Date, Time, and Measurement Utilities to manipulate this information. Specifically, the Date, Time, and Measurement Utilities provide routines that you can use to

- get the current date and time
- set the current date and time, if necessary
- convert between internal date-time structures
- get and set the geographic location and time-zone information
- determine the current measurement system
- determine the number of elapsed microseconds since system startup

The following sections give an overview of these utilities.

Date and Time

A Macintosh computer contains a battery-operated clock chip that maintains the current date-time information. This date-time information is expressed, using 4 bytes, as the number of seconds elapsed since midnight, January 1, 1904. At system startup the date-time information is copied into low memory and is accessible through the system global variable `Time`. System software updates the value of the global variable `Time` each second. Doing this is faster than manipulating the clock chip directly.

The Date, Time, and Measurement Utilities provide four data structures that you can use to access date-time information. You can access date-time information through

- a ***standard date-time value*** that consists of a 32-bit long integer indicating the total number of seconds elapsed since midnight, January 1, 1904
- a ***date-time record*** that contains fields to indicate the year, month, day, hour, minute, second, and day of the week
- a ***long date-time record*** that extends the date-time record format by adding fields for era, day of the year, week of the year, and morning/evening designations (for example, A.M. and P.M.)
- a ***long date-time value*** that consists of a 64-bit integer, in SANE `comp` (computational) format, which also maintains the total number of seconds relative to midnight on January 1, 1904

To access date-time information as a date and time, you can use a date-time record or a long date-time record. A date-time record is defined by a data structure of type `DateTimeRec`

```
TYPE DateTimeRec =
RECORD
   year:       Integer;    {year, ranging from 1904 to 2040}
   month:      Integer;    {month, 1 = January and 12 = December}
   day:        Integer;    {day, from 1 to 31}
   hour:       Integer;    {hour, from 0 to 23}
   minute:     Integer;    {minute, from 0 to 59}
   second:     Integer;    {second, from 0 to 59}
   dayOfWeek:  Integer;    {day of the week, 1 = Sunday, }
                           { 7 = Saturday}
END;
```

The `year` field contains the year of the date, ranging from 1904 to 2040. The `month` field contains the month of the year, where a value of 1 equals January and 12 equals December. The `day` field contains the number of the day, ranging from day 1 to day 31. The `hour` field contains the hour, where the value of 0 equals midnight and 23 equals 11 P.M. The `minute` field contains the number of minutes, ranging from 0 to 59 minutes. The `second` field contains the number of seconds, ranging from 0 to 59 seconds. The `dayOfWeek` field specifies the name of the day; a value of 1 equals Sunday and a value of 7 equals Saturday. For additional information about the fields in a date-time record, see "The Date-Time Record" beginning on page 4-23.

Note
The date-time record can be used to hold date and time values only for a Gregorian calendar. The long date-time record, described next, can be used for a Gregorian calendar as well as other calendar systems. ◆

Because the values in a date-time record are simply a translation of the long integer containing the number of seconds since midnight, January 1, 1904, the data structure suffers the same limitation as the long integer representation: after the long integer has reached its maximum value of $FFFFFFFF, it resets to 0. Therefore, the date-time record can track dates and times only between midnight on January 1, 1904 and 6:28:15 A.M. on February 6, 2040.

For some applications, this range might be inadequate. For example, a hotel management application might need to let managers book reservations for customers who think ahead to 2050, or a history multimedia application might need to track dates in the first century B.C. If your application needs to track dates and times beyond the range supported by the date-time record, you must use a long date-time record. A long date-time record is defined by a data structure of type `LongDateRec`

```
TYPE LongDateRec =
RECORD
   CASE Integer OF
   0:
      (era:       Integer;    {era}
      year:       Integer;    {year, from 30081 B.C. to 29940 A.D}
      month:      Integer;    {month, 1 = January and }
                              { 12 = December}
      day:        Integer;    {day, from 1 to 31}
      hour:       Integer;    {hour, from 0 to 23}
      minute:     Integer;    {minute, from 0 to 59}
      second:     Integer;    {second, from 0 to 59}
      dayOfWeek:  Integer;    {day of the week, 1= Sunday, }
                              { 7 = Saturday}
      dayOfYear:  Integer;    {day of the year, 1 to 365}
      weekOfYear: Integer;    {week of the year, 1 to 52}
      pm:         Integer;    {which half of day--0 for }
                              { morning, 1 for evening}
      res1:       Integer;    {reserved}
      res2:       Integer;    {reserved}
      res3:       Integer);   {reserved}
   1:
                              {index by LongDateField}
      (list:      ARRAY [0..13] OF Integer);
   2:
      (eraAlt:    Integer;    {era}
```

```
                                {date-time record}
      oldDate:    DateTimeRec);

END;
```

You can use a long date-time record for three purposes: to access a date and time, to specify which of the fields in a long date-time record to verify, and to convert a date and time represented by a date-time record into a date and time represented by a long date-time record.

IMPORTANT

The long date-time record covers a much longer time span (30,000 B.C. to 30,000 A.D.) than the date-time record. In addition, the long date-time record allows conversions to different calendar systems, such as a lunar calendar. ▲

A long date time-record includes all of the fields available in a date-time record in addition to fields that describe the era, day of the year, week of the year, and morning /evening designations (for example, A.M. and P.M.). The `era` field contains the era: a value of 0 represents A.D., and –1 represents B.C. The `dayOfYear` field contains a number that represents a day of a year. For example, the value 300 equals the 300th day of a year. The `weekOfYear` field contains a week number. The `pm` field contains the morning or evening half of the 24-hour day cycle, where a value of 0 represents the morning (for example, A.M.) and 1 represents the evening (for example, P.M.).

The `list` field contains an array of values that indicate which of the fields in a long date-time record need to be verified.

The `eraAlt` field, which indicates the era, and the `oldDate` field, which contains a date-time record, are used only for conversion from a date-time record to a long date-time record. For additional information about the fields in the long date-time record, see "The Long Date-Time Record" beginning on page 4-26.

Note that if you specify, in either record, a value in the `month`, `day`, `hour`, `minute`, or `second` field that exceeds the maximum value allowed for that field (for example, a value larger than 23 for the `hour` field), the result is a wraparound to a future date and time when you modify the date-time format. Suppose you set the `year` field in a date-time record to a value greater than 2040, for example 2045. When you modify the date-time format, you get a value of 1909, because the value 2045 caused a wraparound to 1904 plus 5, the number of years over 2040. See "Calculating Dates" beginning on page 4-14 to see how you can use a wraparound to calculate and retrieve information about a specific date.

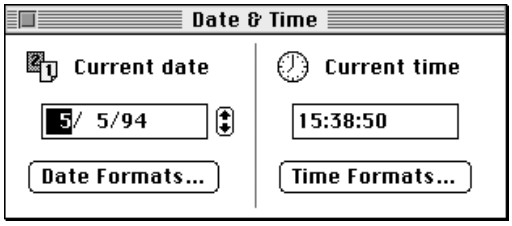
Note

To present a date and time value as a date and time text string, you need to use the Text Utilities routines. For a complete description of these routines, see *Inside Macintosh: Text*. ◆

A user can set the current date-time information by using the General Controls control panel, the Date & Time control panel, or the Alarm Clock. After the user sets the new

date and time, this new date and time is written to the clock chip, and the global variable `Time` is updated to reflect the new date and time. Figure 4-1 illustrates how a user might change the date, using the Date & Time control panel.

Figure 4-1 The Date & Time control panel



Geographic Location and Time Zone

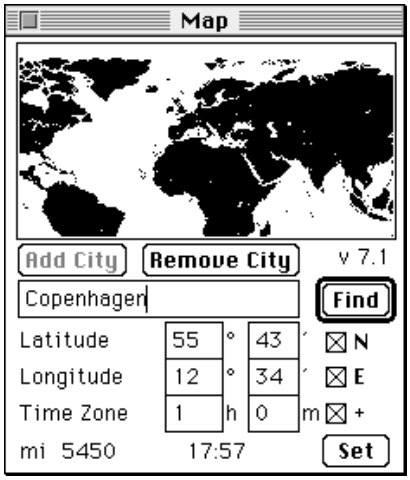
Geographic location and related time-zone information are stored in the Macintosh parameter RAM (extended parameter RAM). System software provides routines that allow you to read this information and, if necessary, make changes to it and then store the new settings in the parameter RAM (extended parameter RAM).

You can read and store values for

- latitude
- longitude
- daylight saving time (DST)
- Greenwich mean time (GMT)

The Map control panel allows the user to get geographic location and time-zone information. Figure 4-2 shows the Map control panel.

Figure 4-2 The Map control panel



The Map control panel specifies latitude and longitude, computation of Greenwich mean time for international time specification (shown as the Time Zone information), and computation of the distance and time difference between the current location (in this case, the location of the user's computer is Cupertino, California) and an arbitrary city (in this case, Copenhagen, Denmark).

See "Handling Geographic Location and Time-Zone Data" beginning on page 4-18, to see how you can use Date, Time, and Measurement Utilities routines to work with the geographic location and time-zone information.

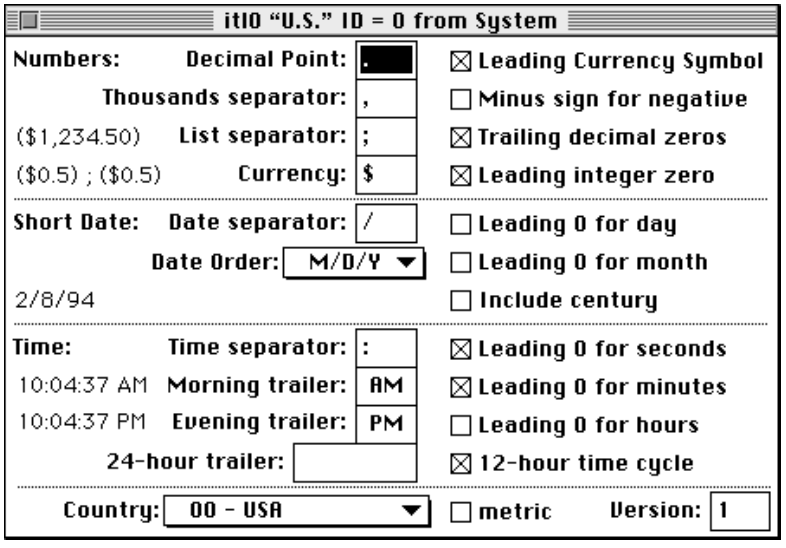
System of Measurement

The Date, Time, and Measurement Utilities provide a routine (the `IsMetric` function) that you can use to determine the type of measurement used by the current script system. The system software supports two types of measurement systems:

- the International System of Units (also called the metric system)—for example centimeters, kilometers, milligrams, degrees Celsius, and so on.
- the English system of measurement (also called the British or British imperial system)—for example, inches, miles, ounces, degrees Fahrenheit, and so on.

The measurement information is stored in the numeric-format resource (resource type `'itl0'`) of a script system. The `IsMetric` function determines whether the current script system uses the International System of Units or the English system of measurement by examining the '`itl0`' resource. Figure 4-3 depicts the window ResEdit displays for a numeric-format resource. Note that in the bottom of the figure the metric box is unchecked, indicating that the script system associated with this '`itl0`' resource uses the English system of measurement.

Figure 4-3 The numeric-format resource (resource type '`it10`')



Time Measurement

The Date, Time, and Measurement Utilities provide a routine (the `Microseconds` procedure) that you can use to measure the number of microseconds that have elapsed since system startup. The `Microseconds` procedure is not effected by any user-specified changes to the date and time information, that is, a user can modify the current date-time information without effecting the value returned by the `Microseconds` procedure.

The number of microseconds elapsed is returned in a 64-bit unsigned integer, specified by the unsigned wide record. An unsigned wide record is defined by a data structure of type `UnsignedWide.`

```
TYPE UnsignedWide =
PACKED RECORD
   hi:   LongInt;          {high-order 32 bits}
   lo:   LongInt;          {low-order 32 bits}
END;
```

Using the Date, Time, and Measurement Utilities

This section describes how to

- get the current date and time
- set the current date and time
- calculate days and dates mathematically
- convert between date-time formats
- convert to different calendar systems
- read and store geographic location and time-zone data
- determine which measurement system to use
- determine the number of elapsed microseconds

Getting the Current Date and Time

The Date, Time, and Measurement Utilities provide

- a function—`ReadDateTime`—that system software uses at system startup time to copy the current date-time information from the clock chip into low memory. This low-memory copy of the current date-time is accessible through the global variable `Time`. You application should never need to use this function.
- two procedures —`GetDateTime` and `GetTime`—that allow you to access the current date-time information stored in the global variable `Time`.

You can access the date-time information through a date-time record, representing the date and time, or you can access the date-time information through a standard date-time value, a 32-bit integer representing the number of seconds since midnight, January 1, 1904.

To obtain the current date-time information, you can use the `GetDateTime` and `GetTime` procedures. The `GetDateTime` procedure requires that you pass it a standard date-time value as a parameter. Listing 4-1 shows how you can get the current date-time information, expressed as a number of seconds. The application-defined procedure `MyCurrentDateTimeInt` returns in the long integer the number of seconds elapsed since midnight, January 1, 1904.

Listing 4-1 Getting the current date and time with the `GetDateTime` procedure

```
PROCEDURE MyCurrentDateTimeInt (VAR myStandardDateTime: LongInt);
BEGIN
   GetDateTime(myStandardDateTime);
END;
```

The `GetTime` procedure requires that you pass it a date-time record as a parameter, and it fills in the fields of this record appropriately. Listing 4-2 shows how you can get the current date-time information, expressed as a date and time. The application-defined procedure `MyCurrentDateTimeRec` returns in the fields of the date-time record the current date and time.

Listing 4-2 Getting the current date and time with the `GetTime` procedure

```
PROCEDURE MyCurrentDateTimeRec (VAR myDateTime: DateTimeRec);
BEGIN
   GetTime(myDateTime);
END;
```

If you need to access the date-time information through a long date-time value or a long date-time record, see "Converting Date-Time Formats" beginning on page 4-12 for more information about converting date-time formats.

Setting the Current Date and Time

Your application can change the current date-time information stored in both the system global variable `Time` and in the clock chip by calling either the `SetDateTime` function or the `SetTime` procedure. The `SetDateTime` function requires a 32-bit integer as a parameter. The `SetTime` procedure requires a date-time record as a parameter.

Note
If you are using formats other than a date-time value or a date-time record to access date-time information, you must first convert these formats into a standard date-time value or a date-time record before you can write the new date-time information to the clock chip. See "Converting Date-Time Formats" beginning on page 4-12 for more information about converting date-time formats. ◆

Listing 4-3 shows an application-defined function that uses the `SetDateTime` function to change the current date and time to 5:50 A.M. on April 5, 1994.

Listing 4-3 Changing the current date and time with the `SetDateTime` function

```
FUNCTION MyChangeDateTimeInt: OSErr;
VAR
   myDateTimeInt: LongInt;
   myErr:         OSErr;
BEGIN
   myDateTimeInt := $A9C6AC88;
   myErr := SetDateTime(myDateTimeInt);
END;
```

Listing 4-4 shows an application-defined procedure that uses the `SetTime` function to change the current date and time to 5:50 A.M. on April 5, 1994.

Listing 4-4 Changing the current date and time with the `SetTime` function

```
PROCEDURE MyChangeDateTimeRec;
VAR
   myDateTimeRec: DateTimeRec;
   myErr:         OSErr;
BEGIN
   WITH myDateTimeRec DO
   BEGIN
      year := 1994;
      month := 4;
      day := 5;
      hour := 5;
      minute := 50;
      second := 0;
      dayOfWeek := 3;
   END;
   SetTime(myDateTimeRec);
END;
```

IMPORTANT

Users can change the current date and time stored in both the system global variable `Time` and in the clock chip by using the General Controls control panel, Date & Time control panel, or the Alarm Clock desk accessory. In general, your application should not directly change the current date-time information. If your application does need to modify the current date-time information, it should instruct the user how to change the date and time. ▲

Converting Date-Time Formats

The Date, Time, and Measurement Utilities provide four routines—the `DateToSeconds`, `SecondsToDate`, `LongDateToSeconds`, and `LongSecondsToDate` procedures—that you can use to convert date-time formats. You can convert a date and time to a number of seconds and a number of seconds to a date and time.

Note that when you call one of these routines, system software uses the `DateToSeconds`, `SecondsToDate`, `LongDateToSeconds`, and `LongSecondsToDate` procedures provided by the current script system.

Note

The routines that convert between time formats assume that each day contains 86,400 seconds. Occasionally (approximately once each two years) astronomers add a second to either June 31 or December 31 to compensate for imperfections in the earth's rotation. If you need to compute the exact number of seconds between two points in time, you might need to take these occasional additions into account. The routines that convert between formats are designed not to provide astronomical accuracy, but merely to convert data between one data structure and another. ◆

If you use a standard date-time value or a date-time record to access date-time information, you can use the `SecondsToDate` procedure to convert a number of seconds to a date and time, and the `DateToSeconds` procedure to convert a date and time to a number of seconds. Listing 4-5 shows an application-defined procedure, `MyConvertSecondsAndDates`, that uses the `SecondsToDate` and `DateToSeconds` procedures to manipulate the date-time information. After calling the `GetDateTime` procedure, `MyConvertSecondsAndDates` calls the `SecondsToDate` procedure to convert the number of seconds (returned by the `GetDateTime` procedure) to a date and time. The `MyConvertSecondsAndDates` procedure manipulates the `year` field in the date-time record and then calls `DateToSeconds` to convert the date and time back into a number of seconds. The `SetDateTime` procedure writes the new date-time information to the clock chip.

Listing 4-5 Manipulating date-time information

```
PROCEDURE MyConvertSecondsAndDates;
VAR
   myDateTimeRec:       DateRec;
   mySeconds:           DateTime;
   myErr:               OSErr;
BEGIN
   GetDateTime(mySeconds);
   SecondsToDate(mySeconds, myDateTimeRec);
   WITH myDateTimeRec DO
      year := year + 1;
   DateToSeconds (myDateTimeRec, mySeconds);
   myErr := SetDateTime(mySeconds);
END;
```

If you access date-time information through a long date-time value or a long date-time record, you can use the `LongSecondsToDate` procedure to convert a number of seconds to a date and time and use the `LongDateToSeconds` procedure to convert a date and time to a number of seconds.

If the type of data structure that you are using to access date-time information is insufficient, you can use a different date-time structure.

- To access a number of seconds through a long date-time value instead of a standard date-time value, set the `lHigh` field of a long date-time conversion record (described on page 4-25) to 0 and the `lLow` field to the total number of seconds since midnight, January 1, 1904. Then copy the value of the `c` field into a variable of type `LongDateTime`.
- To access a date and time through a long date-time record instead of a date-time record, set the `oldDate` field of the `LongDateRec` to the date-time record, and set the `eraAlt` field to 0, indicating that the date you have specified is A.D.
- To access a number of seconds through a standard date-time value instead of a long date-time value, truncate the long date-time value to just the low-order 32 bits. The year of the date being converted must fall within 1904 to 2040 of the Gregorian calendar.

  This type of conversion is important when you work with a script system that uses a calendar system other than the Gregorian. Because you cannot write a long date-time value to the clock chip, you must first convert the long date-time value (if possible) to a standard date-time value. See "Working With Different Calendar Systems" beginning on page 4-16 for more information about calendar systems.
- To access a date and time through a date-time record instead of a long date-time record, truncate the long date-time record so just the `year` through `dayOfWeek` fields are left. Once again, the year of the date being converted must fall within 1904 to 2040 of the Gregorian calendar.

- To access date-time information through a long date-time value instead of a date-time record, use the `DateToSeconds` procedure to convert the date and time to a number of seconds. Then set the `lHigh` field of a long date-time conversion record (described on page 4-25) to 0 and the `lLow` field to the total number of seconds since midnight, January 1, 1904.
- To access date-time information through a long date-time record (described on page 4-26) instead of a standard date-time value, use the `SecondsToDate` procedure to translate the number of seconds to a date and time. Then set the `oldDate` field of the long date-time record to the date-time record, and set the `eraAlt` field to 0.
- To access date-time information through a date-time value instead of long date-time record, use the `LongDateToSeconds` procedure to translate the date and time to a number of seconds. Then truncate the long date-time value (returned by the `LongDateToSeconds` procedure) to just the low-order 32 bits. The year of the date being converted must fall within 1904 to 2040 in the Gregorian calendar.

The Gregorian calendar is the default for converting to and from the long date-time forms. The current range allowed in conversion is roughly 30,000 B.C. to 30,000 A.D.

To present a date and time value as a date and time text string, you need to use Text Utilities routines, such as the `DateString`, `TimeString`, `StringToDate`, `StringToTime`, `LongDateString`, and `LongTimeString` routines. (Note that the date-string conversion routines do not append strings for A.D. or B.C.) For a complete description of these routines, see *Inside Macintosh: Text.*

Calculating Dates

In the date-time record and long date-time record, any value in the `month`, `day`, `hour`, `minute`, or `second` field that exceeds the maximum value allowed for that field, will cause a wraparound to a future date and time when you modify the date-time format.

- In the `month` field, values greater than 12 cause a wraparound to a future year and month.
- In the `day` field, values greater than the number of days in a given month cause a wraparound to a future month and day.
- In the `hour` field, values greater than 23 cause a wraparound to a future day and hour.
- In the `minute` field, values greater than 59 cause a wraparound to a future hour and minute.
- In the `seconds` field, values greater than 59 cause a wraparound to a future minute and seconds.

You can use these wraparound facts to calculate and retrieve information about a specific date. For example, you can use a date-time record and the `DateToSeconds` and `SecondsToDate` procedures to calculate the 300th day of 1994. Set the `month` field of the date-time record to 1 and the `year` field to 1994. To find the 300th day of 1994, set the `day` field of the date-time record to 300. Initialize the rest of the fields in the record to values that do not exceed the maximum value allowed for that field. (Refer to the description of the date-time record on page 4-23 for a complete list of possible values).

To force a wrap-around, first convert the date and time (in this example, January 1, 1994) to the number of seconds elapsed since midnight, January 1, 1904 (by calling the `DateToSeconds` procedure). Once you have converted the date and time to a number of seconds, you convert the number of seconds back to a date and time (by calling the `SecondsToDate` procedure). The fields in the date-time record now contain the values that represent the 300th day of 1994. Listing 4-5 shows an application-defined procedure that calculates the 300th day of the Gregorian calendar year using a date-time record.

Listing 4-6 Calculating the 300th day of the year

```
PROCEDURE MyCalculate300Day;
VAR
   myDateTimeRec:    DateTimeRec;
   mySeconds:        LongInt;
BEGIN
   WITH myDateTimeRec DO
   BEGIN
      year := 1994;
      month := 1;
      day := 300;
      hour := 0;
      minute := 0;
      second := 0;
      dayOfWeek := 1;
   END;
   DateToSeconds (myDateTimeRec, mySeconds);
   SecondsToDate (mySeconds, myDateTimeRec);
END;
```

The `DateToSeconds` procedure converts the date and time to the number of seconds elapsed since midnight, January 1, 1904, and the `SecondsToDate` procedure converts the number of seconds back to a date and time. After the conversions, the values in the `year`, `month`, `day`, and `dayOfWeek` fields of the `myDateTimeRec` record represent the year, month, day of the month, and day of the week for the 300th day of 1994. If the values in the `hour`, `minute`, and `second` fields do not exceed the maximum value allowed for each field, the values remain the same after the conversions (in this example, the time is exactly 12:00 A.M.).

Similarly, you can use a long date-time record and the `LongDateToSeconds` and `LongSecondsToDate` procedures to compute the day of the week corresponding to a given date. Listing 4-7 shows an application-defined procedure that computes and retrieves the name of the day for July 4, 1776. Note that because the year is prior to 1904, it is necessary to use a long date-time record.

Listing 4-7 Computing the day of the week

```
PROCEDURE DoDayCalc;
VAR
   myLongDateRec: LongDateRec;
   myLongSeconds: LongDateTime;
   myDayOfWeek:   Integer;
BEGIN
   WITH myLongDateRec DO
   BEGIN
      era := 0;         /*initialize era field*/
      year := 1776;
      month := 7;
      day := 4;
      hour := 0;        /*initialize hour field*/
      minute := 0;      /*initialize minute field*/
      second := 0;      /*initialize second field*/
      dayOfWeek := 1;   /*initialize dayOfWeek field*/
      dayOfYear := 1;   /*initialize dayOfYear field*/
      weekOfYear := 1;  /*initialize weekOfYear field*/
      pm := 1;          /*initialize pm field*/
   END;
   LongDateToSeconds (myLongDateRec, myLongSeconds);
   LongSecondsToDate (myLongSeconds, myLongDateRec);
   myDayOfWeek := myLongDateRec.dayOfWeek;
END;
```

The `LongDateToSeconds` procedure converts the date and time to the number of seconds, and the `LongSecondsToDate` procedure converts the number of seconds back to a date and time. After the conversions, the value in the `dayOfWeek` field of the `myLongDateRec` record represent the day of the week corresponding to July 4, 1776. If the values in the `hour`, `minute`, and `second` fields do not exceed the maximum value allowed for each field, the values remain the same after the conversions (in this example, the time is exactly 12:00 A.M.). The values in the `dayOfYear`, `weekOfYear`, and `pm` fields correspond to the date July 4, 1776 and the time 12:00 A.M.

Working With Different Calendar Systems

The additional fields and wider ranges allowed by the long date-time record can help you to do calculations and conversions for different calendar systems. For example, the date January 1, 1993 in the Gregorian calendar year converts to 7 Rajab 1413 in the Arabic Civil Lunar Calendar (CLC) and 4 Tevet 5753 in the Jewish calendar; the years 1413 and 5753 are outside of the `year` field's range in the date-time record.

Note
Depending on the country, the change from the Julian calendar to the Gregorian calendar occurred in different years. In western European countries, the change occurred in 1582; in Russia, the calendar changed in 1918. In these countries, dates before the calendar change should use the Julian calendar for conversion. (The Julian calendar differs from the Gregorian calendar by three days every four centuries.) ◆

In addition, the beginning of the year for one calendar system falls on different dates in other calendar systems. Table 4-1 shows the equivalent dates for the first day of the calendar year in the Gregorian, Arabic CLC, and Jewish calendars.

Table 4-1 Equivalent dates in the Gregorian, Arabic CLC, and Jewish calendars

Gregorian calendar	Arabic CLC	Jewish calendar
January 1, 1993	7 Rajab 1413	4 Tevet 5753
June 20, 1993	1 Muharram 1414	1 Tammuz 5753
September 16, 1993	29 Rabi I 1414	1 Tishri 5754

Converting from one calendar system to another produces different values in the `dayOfYear` and `weekOfYear` fields of a long date-time record. For example, assuming all the data for the date 1 Muharram 1414 is correctly put into a long date-time record, the `dayOfYear` field value is 1, and the `weekOfYear` value is also 1. Converting this date to the Gregorian calendar results in June 20, 1993. The `dayOfYear` field value is then 171, and the `weekOfYear` value is 26. Table 4-2 shows these values.

Table 4-2 Values for the `dayOfYear` and `weekOfYear` fields for the date 1 Muharram 1414 and equivalent values in the Gregorian calendar

LongDateRec field	Arabic CLC	Gregorian calendar
`dayOfYear`	1	171
`weekOfYear`	1	26

Note
Language-specific information, such as the name of the day, name of the month, and so on, are stored in the international resources. The international resources are provided by a script system, and the information in these resources varies according to the language associated with the script system. ◆

Table 4-3 shows how some of the fields in the long date-time record are set to show the first day of the year 1414 in the Arabic CLC and the equivalent dates in the Gregorian and Jewish calendars.

Table 4-3 Comparison of settings in fields of the long date-time record for Arabic CLC, Gregorian, and Jewish calendars

Field of a long date-time record	Arabic CLC calendar	Gregorian calendar	Jewish calendar
`era`	0	0	0
`year`	1413	1993	5753
`month`	1	6	
`day`	1	21	
…			
`dayOfWeek`	4	2	3
`dayOfYear`	1	172	
`weekOfYear`	1	26	

Note
The Arabic script system supports two lunar calendars: the astronomical lunar calendar (ALC) and the civil lunar calendar (CLC). The Macintosh user may choose either of the Arabic calendars or the Gregorian calendar by clicking buttons in the Arabic Calendar control panel.

The Hebrew script system supports the Jewish calendar besides the Gregorian calendar.

For more information on the different calendar systems supported by localized versions of the Macintosh system software, see *Guide to Macintosh Software Localization*. ◆

For calendars that have more than seven day names and 12 month names (for example, the Jewish calendar sometimes has 13 months), you use the `'itl1'` resource, defined by the `Itl1ExtRec` data type. To get more information on the format of the `'itl1'` resource, see the appendix "International Resources" in *Inside Macintosh: Text*.

Handling Geographic Location and Time-Zone Data

Geographic locations and time zones can affect date and time information. For example, time-zone information can be used to derive the Greenwich mean time (GMT) at which a document or mail message was created. With this information, when the document is received by an application or user in a different time zone, the creation date and time are correct. Otherwise, documents can appear to be created after they are read (for example, a user creates a message in Tokyo on Tuesday and sends it to San Francisco, where it is received and read on Monday). Geographic location information can also be used by applications that require it.

The geographic location and time-zone information for a particular Macintosh computers are stored in parameter RAM. You can work with this information through the `ReadLocation` and `WriteLocation` procedures. These procedures use the

geographic location record (of date type `MachineLocation`) to help you read and store latitude, longitude, daylight saving time (DST), and GMT values.

```
TYPE MachineLocation =              {geographic location record}
RECORD
   latitude:         Fract;         {latitude}
   longitude:        Fract;         {longitude}
   CASE Integer OF
   0:
      (dlsDelta:     SignedByte);   {daylight saving time}
   1:
      (gmtDelta:     LongInt);      {Greenwich mean time}
END;
```

The daylight savings time value is a signed byte value that you can use to specify the offset for the `hour` field—whether to add 1 hour, subtract 1 hour, or make no change at all.

The Greenwich mean time value is in seconds east of GMT. For example, San Francisco is at –28,800 seconds (8 hours * 3,600 seconds per hour) east of GMT.

If the geographic location record has never been set, all fields contain 0.

Generally, latitude and longitude are measured in degrees. These values also can be thought of as fractions of a great circle.

Latitude and longitude information is stored in the geographic location record as values of type `Fract`. These values give accuracy to within 1 foot, which should be sufficient for most purposes. For example, the `Fract` value 1.0 equals 90 degrees; –1.0 equals –90 degrees; and –2.0 equals –180 degrees.

To store latitude and longitude values, you need to convert them first to the `Fixed` data type, then to the `Fract` data type. You can use the Operating System Utilities routines `Long2Fix` and `Fix2Fract` to accomplish this task. Listing 4-8 is an application-defined procedure that converts San Francisco's latitude and longitude to `Fract` values, then writes the `Fract` values to parameter RAM using the `WriteLocation` procedure.

Listing 4-8 Converting latitude and longitude to `Fract` values

```
PROCEDURE MyConvertLatLong;
VAR
   myLatitude, myLongitude:         LongInt;
   fixedLatitude, fixedLongitude:   Fixed;
   latFract, longFract:             Fract;
   myLocation:                      MachineLocation;
BEGIN
   myLatitude:= 37.48;                    {degrees latitude}
   myLongitude:= 122.24;                  {degrees longitude}
```

```
   {convert from long to fixed data type}
   fixedLatitude:= Long2Fix(myLatitude);
   fixedLongitude:= Long2Fix(myLongitude);

   {convert from fixed to Fract data type}
   latFract:= Fix2Frac(fixedLatitude);
   longFract:= Fix2Frac(fixedLongitude);

   {write latitude and logitude to myLocation}
   myLocation.latitude:= latFract;
   myLocation.longitude:= longFract;

   {write latitude and longitude to parameter RAM}
   WriteLocation(myLocation);

END;
```

To read the latitude and longitude values from parameter RAM, you use the `ReadLocation` procedure. To convert these values to a degrees format, you need to convert the `Fract` values first to the `Fixed` data type, then to the `LongInt` data type. You can use the Mathematical and Logical Utilities routines `Fract2Fix` and `Fix2Long` to accomplish this task. (For more information on the `Fract` data type and the conversion routines `Long2Fix`, `Fix2Fract`, `Fract2Fix`, and `Fix2Long`, see the chapter "Mathematical and Logical Utilities" in this book.)

The `gmtDelta` field of the geographic location record is a 3-byte value contained in a long word, so you must take care to get and set it properly. Listing 4-9 shows an application-defined function for obtaining the value of `gmtDelta`.

Listing 4-9 Getting `gmtDelta`

```
FUNCTION MyGetGmtDelta (myLocation: MachineLocation): LongInt;
VAR
   internalGmtDelta: LongInt;
BEGIN
   WITH myLocation DO
   BEGIN
      internalGmtDelta := BitAnd(gmtDelta, $00FFFFFF);
      IF BitTst(internalGmtDelta, 23) THEN
                                       {test sign extend bit}
         internalGmtDelta := BitOr(internalGmtDelta, $FF000000);
      MyGetGmtDelta := internalGmtDelta;
   END;
END;
```

When writing `gmtDelta`, you should preserve the value of `dlsDelta`. Listing 4-10 shows an application-defined procedure that writes `gmtDelta` while preserving the value of `dlsDelta`.

Listing 4-10 Setting `gmtDelta`

```
PROCEDURE MySetGmtDelta (VAR myLocation: Location;
                         myGmtDelta: LongInt);
VAR
   tempSignedByte: SignedByte;
BEGIN
   WITH myLocation DO
   BEGIN
      tempSignedByte := dlsDelta;         {preserve dlsDelta}
      gmtDelta := myGmtDelta;             {write gmtDelta}
      dlsDelta := tempSignedByte;         {restore dlsDelta}
   END;
END;
```

Note that you should mask off the top byte of the long word containing `gmtDelta` because it is reserved.

Determining the Measurement System

To implement measuring devices in applications, such as rulers in a word processor or in drawing applications, you need to determine which measurement system your application should use. You can use the `IsMetric` function to determine if the measurement system needs to be the metric system or the English system. The `IsMetric` function reads the numeric-format resource (resource type `'itl0'`) of the current script system to determine whether the user is using the metric system or the English system.

Listing 4-11 shows an application-defined procedure that uses the result of the `IsMetric` function to determine which application-defined ruler setup to use for a document window.

Listing 4-11 Getting the current units of measurement

```
PROCEDURE DoRuler (window: WindowPtr);
VAR
   myMeasure: BOOLEAN;           {response returned by IsMetric}
BEGIN
   myMeasure := IsMetric;
   IF myMeasure = TRUE THEN      {metric system is default}
```

```
      DoMetricRulerSetup         {set up metric system ruler}
   ELSE
      DoEnglishRulerSetup;       {set up English system ruler}
END;
```

If you want to use a measurement system different from that of the current script, you need to override the value of the `metricSys` field in the current numeric-format resource (resource type `'itl0'`). You can do this by using your own version of the numeric-format resource instead of the current script system's default international resources. See the chapter "Script Manager" in *Inside Macintosh: Text* for information on how to replace a script system's default international resources.

Determining the Number of Elapsed Microseconds

Your application can use the `Microseconds` procedure to obtain the number of elapsed microseconds since system startup time. You can use the value returned by the `Microseconds` procedure to time an event. For example, Listing 4-11 shows an application-defined function `MyEventTimer` that computes and returns the time it takes to execute an application-defined procedure `DoMyEvent`. The application-defined function `MyCalulateElapsedTime` function uses the returned value of the `Microseconds` procedure to compute the time it takes to execute the `DoMyEvent` procedure.

Listing 4-12 Timing an event using the `Microseconds` procedure

```
FUNCTION MyEventTimer: UnsignedWide;
VAR
   myStartTime:UnsignedWide;
   myEndTime: UnsignedWide;
BEGIN
   Microseconds(&myStartTime);
   DoMyEvent;
   Microseconds(&myEndTime);
   MyEventTimer := MyComputeElapsedTime(&myStartTime, &myEndTime);
END;
```

Because there is no compiler support for 64-bit integers, you must write an application-defined routine that calculates the elapsed time; you cannot obtain the elapsed time by subtracting the value in the `myStartTime` parameter from the value in the `myEndTime` parameter.

Date, Time, and Measurement Utilities Reference

This section describes the data structures and routines that are specific to the Date, Time, and Measurement Utilities. The section "Data Structures" shows the Pascal data structures for the date-time record, long date-time record, standard date-time value, long date-time value, and more. The section "Routines" describes the routines you can use to read, write, and manipulate date-time information.

Data Structures

This section describes the data structures that you use to exchange information with the Date, Time, and Measurement Utilities.

The Date-Time Record

The date-time record describes the date-time information as a date and time. The Date, Time, and Measurement Utilities use a date-time record to read and write date-time information to and from the clock chip. The `DateTimeRec` data type defines the date-time record.

Note
The date-time record can be used to hold date and time values only for a Gregorian calendar. The long date-time record (described on page 4-26) can be used for a Gregorian calendar as well as other calendar systems. ◆

```
TYPE DateTimeRec =
RECORD
   year:       Integer;    {year, ranging from 1904 to 2040}
   month:      Integer;    {month, 1= January and 12 = December}
   day:        Integer;    {day of the month, from 1 to 31}
   hour:       Integer;    {hour, from 0 to 23}
   minute:     Integer;    {minute, from 0 to 59}
   second:     Integer;    {second, from 0 to 59}
   dayOfWeek:  Integer;    {day of the week, 1 = Sunday, }
                           { 7 = Saturday}
END;
```

Field descriptions

`year` The year, ranging from 1904 to 2040. Note that to indicate the year 1984, this field would store the integer 1984, not just 84. This field accepts input of 0 or negative values, but these values produce unpredictable results in the `year`, `month`, and `day` fields when you

use the SecondsToDate and DateToSeconds procedures. In addition, using SecondsToDate and DateToSeconds with year values greater than 2040 causes a wraparound to 1904 plus the number of years over 2040. For example, setting the year to 2045 returns a value of 1909, and the other fields in this record return unpredictable results.

`month` The month of the year, where 1 represents January, and 12 represents December. Values greater than 12 cause a wraparound to a future year and month. This field accepts input of 0 or negative values, but these values produce unpredictable results in the `year`, `month`, and `day` fields when you use the SecondsToDate and DateToSeconds procedures.

`day` The day of the month, ranging from 1 to 31. Values greater than the number of days in a given month cause a wraparound to a future month and day. This feature is useful for working with leap years. For example, the 366th day of January in 1992 (1992 was a leap year) evaluates as December 31, 1992, and the 367th day of that year evaluates as January 1, 1993.

This field accepts 0 or negative values, but when you use the SecondsToDate and DateToSeconds procedures, a value of 0 in this field returns the last day of the previous month. For example, a month value of 2 and a day value of 0 return 1 and 31, respectively.

Using SecondsToDate and DateToSeconds with a negative number in this field subtracts that number of days from the last day in the previous month. For example, a month value of 5 and a day value of –1 return 4 for the month and 29 for the day; a month value of 2 and a day value of –15 return 1 and 16, respectively.

`hour` The hour of the day, ranging from 0 to 23, where 0 represents midnight and 23 represents 11:00 P.M. Values greater than 23 cause a wraparound to a future day and hour. This field accepts input of negative values, but these values produce unpredictable results in the `month`, `day`, `hour`, and `minute` fields you use the SecondsToDate and DateToSeconds procedures.

`minute` The minute of the hour, ranging from 0 to 59. Values greater than 59 cause a wraparound to a future hour and minute. When you use the SecondsToDate and DateToSeconds procedures, a negative value in this field has the effect of subtracting that number from the beginning of the given hour. For example, an `hour` value of 1 and a `minute` value of –10 return 0 hours and 50 minutes. However, if the negative value causes the `hour` value to be less than 0, for example `hour` = 0, `minute` = –61, unpredictable results occur.

`second` The second of the minute, ranging from 0 to 59. Values greater than 59 cause a wraparound to a future minute and second. When you use the SecondsToDate and DateToSeconds procedures, a negative value in this field has the effect of subtracting that number from the beginning of the given minute. For example, a `minute` value of 1 and a `second` value of –10 returns 0 minutes and 50 seconds. However, if the negative value causes the `hour` value to be

less than 0, for example `hour` = 0, `minute` = 0, and `second` = –61, unpredictable results occur.

`dayOfWeek` The day of the week, where 1 indicates Sunday and 7 indicates Saturday. This field accepts 0, negative values, or values greater than 7. When you use the `SecondsToDate` and `DateToSeconds` procedures, you get correct values because this field is automatically calculated from the values in the `year`, `month`, and `day` fields.

Long Date-Time Value and Long Date-Time Conversion Record

The long date-time value specifies the date and time as seconds relative to midnight, January 1, 1904. But where the standard date-time value is an unsigned, 32-bit long integer, the long date-time value is a signed, 64-bit integer in SANE `comp` format. This format lets you use dates and times with a much longer span—roughly 500 billion years. You can use this value to represent dates and times prior to midnight, January 1, 1904. The `LongDateTime` data type defines the long date-time value.

```
TYPE LongDateTime = comp;
```

When storing a long date-time value in files, you can use a 5-byte or 6-byte format for a range of roughly 35,000 years. You should sign extend this value to restore it to a `comp` format.

The Date, Time, and Measurement Utilities provide the `LongDateCvt` record to help in setting up `LongDateTime` values.

```
TYPE LongDateCvt =
RECORD
   CASE Integer OF
   0:
      (c:      comp);      {number of seconds relative to }
                           { midnight, January 1, 1904}
   1:
      (lHigh:  LongInt;    {high long integer}
       lLow:   LongInt);   {low long integer}
END;
```

Field descriptions

`c` The date and time, specified in seconds relative to midnight, January 1, 1904, as a signed, 64-bit integer in SANE `comp` format. The high-order bit of this field represents the sign of the 64-bit integer. Negative values allow you to indicate dates and times prior to midnight, January 1, 1904.

`lHigh` The high-order 32 bits when converting from a standard date-time value. Set this field to 0.

`lLow` The low-order 32 bits when converting from a standard date-time value. Set this field to the standard date-time value representing the total number of seconds since midnight, January 1, 1904.

The Long Date-Time Record

In addition to the date-time record, system software provides the long date-time record, which extends the date-time record format by adding several more fields. This format lets you use dates and times with a much longer span (30,000 B.C. to 30,000 A.D.). In addition, the long date-time record allows conversions to different calendar systems, such as a lunar calendar.

The `LongDateRec` data type defines the format of the long date-time record.

```
TYPE LongDateRec =
RECORD
   CASE Integer OF
   0:
      (era:          Integer;    {era}
       year:         Integer;    {year, from 30,081 B.C. }
                                 { to 29,940 A.D.}
       month:        Integer;    {month}
       day:          Integer;    {day of the month}
       hour:         Integer;    {hour, from 0 to 23}
       minute:       Integer;    {minute, from 0 to 59}
       second:       Integer;    {second, from 0 to 59}
       dayOfWeek:    Integer;    {day of the week}
       dayOfYear:    Integer;    {day of the year}
       weekOfYear:   Integer;    {week of the year}
       pm:           Integer;    {morning/evening}
       res1:         Integer;    {reserved}
       res2:         Integer;    {reserved}
       res3:         Integer);   {reserved}
   1:
                                 {index by LongDateField}
      (list:         ARRAY[0..13] OF Integer);
   2:
      (eraAlt:       Integer;       {era}
      oldDate:       DateTimeRec);  {date-time record}
END;
```

Field descriptions

`era` The era, where 0 represents A.D., and –1 represents B.C.

`year` The year, ranging from 30,081 B.C. to 29,940 A.D. Values outside this range produce unpredictable results in all fields of the record. Note that to indicate the year 1984, this field would store the integer 1984,

not just 84. This field accepts input of 0 or negative values, but these values return the positive result of the value plus one for the year. For example, a `year` value of 0 returns 1, and a `year` value of –1993 returns 1994. Other fields are unaffected.

`month` The month of the year, where 1 represents January, and 12 represents December. When you use the `LongSecondsToDate` and `LongDateToSeconds` procedures, `month` values greater than 12 cause a wraparound to a future year and month. A value of 0 in this field returns the 12th month of the previous year. For example, a `month` value of 0 and a `year` value of 1993 return 12 and 1992, respectively. A negative value in this field has the effect of subtracting that number from the first month of the given year. For example, a `month` value of –2 and a `year` value of 1993 return 10 and 1992, respectively.

`day` The day of the month, ranging from 1 to 31. When using the `LongSecondsToDate` and `LongDateToSeconds` procedures, `day` values greater than the number of days in a given month cause a wraparound to a future month and day. This feature is useful for working with leap years. For example, the 366th day of January in 1992 (1992 was a leap year) evaluates as December 31, 1992, and the 367th day of that year evaluates as January 1, 1993. A value of 0 in this field produces unpredictable results in the `month` and `day` fields. A negative value in this field has the effect of subtracting that number from the first day of the given month. For example, a `day` value of –10 and a `month` value of 10 return 9 and 20, respectively.

`hour` The hour of the day, ranging from 0 to 23, where 0 represents midnight and 23 represents 11:00 P.M. When you use the `LongSecondsToDate` and `LongDateToSeconds` procedures, `hour` values greater than 23 cause a wraparound to a future day and hour. A negative value in this field produces unpredictable results. Note that this field is always maintained in 24-hour time. The `pm` field, if used, is redundant.

`minute` The minute of the hour, ranging from 0 to 59. When you use the `LongSecondsToDate` and `LongDateToSeconds` procedures, `minute` values greater than 59 cause a wraparound to a future hour and minute. A negative value in this field has the effect of subtracting that number from the first minute of the given hour. For example, an `hour` value of 10 and a `minute` value of –10 return 9 and 50, respectively. However, if the negative value causes the `hour` value to become less than 0, for example `hour` = 0 and `minute` = –61, unpredictable results occur.

`second` The second of the minute, ranging from 0 to 59. When you use the `LongSecondsToDate` and `LongDateToSeconds` procedures, `second` values greater than 59 cause a wraparound to a future minute and second. A negative value in this field has the effect of subtracting that number from the first second of the given minute. For example, an `minute` value of 10 and a `second` value of –10 return 9 and 50, respectively. However, if the negative value causes

the `hour` value to become less than 0, for example `hour` = 0, `minute` = 0, and `second` = –61, unpredictable results occur.

`dayOfWeek` The day number of the week, where 1 indicates Sunday and 7 indicates Saturday. This field accepts 0, negative values, or values greater than 7. When you use the `LongSecondsToDate` and `LongDateToSeconds` procedures, you get correct values because this field is automatically calculated from the values in the `year`, `month`, and `day` fields. For calendars that have more than 7 day names and 12 month names (for example, the Jewish calendar sometimes has 13 months), you use the `'itl1'` resource, defined by the `Itl1ExtRec` data type. To get more information on the format of the `'itl1'` resource, see the appendix "International Resources" in *Inside Macintosh: Text*.

`dayOfYear` The day number of the year, ranging from 1 to 366. Values greater than the number of days in a given year cause a wraparound to a future year and day. This feature is useful for working with leap years. For example, in a Gregorian calendar the 366th day of January in 1992 (1992 was a leap year) evaluates as December 31, 1992, and the 367th day of that year evaluates as January 1, 1993.

`weekOfYear` The week number of the year, ranging from 1 to 52. Note that out-of-range values (such as 0, negative numbers, or numbers greater than 52) can be set for this field. However, you can use the `LongSecondsToDate` procedure to convert these out-of-range values to appropriate values.

`pm` The morning or evening half of the 24-hour day cycle, where 0 represents the morning (for example, A.M.), and 1 represents the evening (for example, P.M.). Note that out-of-range values can be set for this field. However, you can use the `LongSecondsToDate` procedure to convert these out-of-range values to appropriate values.

`res1` Reserved. Set this field to 0.

`res2` Reserved. Set this field to 0.

`res3` Reserved. Set this field to 0.

`list` An array of `LongDateField` values. The `field` parameter of the `ToggleDate` function uses the enumerated data type `LongDateField` to indicate the `LongDateRec` fields that the `ValidDate` function should check. The following values are available:

```
TYPE LongDateField =
     (eraField, yearField, monthField, dayField,
      hourField, minuteField, secondField,
      dayOfWeekField, dayOfYearField,
      weekOfYearField, pmField, res1Field,
      res2Field, res3Field);
```

`eraAlt` The era, where 0 represents A.D., and –1 represents B.C. Use this field and the `oldDate` field to convert from a date-time record.

`oldDate` The date-time record to convert. Use this field and the `eraAlt` field to convert from a date-time record.

The Geographic Location Record

The geographic location and time-zone information of a Macintosh computer are stored in extended parameter RAM. The `MachineLocation` data type defines the format for the geographic location record.

```
TYPE MachineLocation =  {geographic location record}
RECORD
   latitude:            Fract;         {latitude}
   longitude:           Fract;         {longitude}
   CASE Integer OF
   0:
      (dlsDelta:        SignedByte);   {daylight saving time}
   1:
      (gmtDelta:        LongInt);      {Greenwich mean time}
END;
```

Field descriptions

`latitude` The location's latitude, in fractions of a great circle. For example, Copenhagen, Denmark is at 55.43 degrees north latitude. When writing the latitude to extended parameter RAM with the `WriteLocation` procedure, you must convert this value to a `Fract` data type. (For example, a `Fract` value of 1.0 equals 90 degrees; –1.0 equals –90 degrees; and –2.0 equals –180 degrees.) For an example that shows this conversion process, see Listing 4-8 on page 4-19. For more information on the `Fract` data type, see the chapter "Mathematical and Logical Utilities" in this book.

`longitude` The location's longitude, in fractions of a great circle. For example, Copenhagen, Denmark is at 12.34 degrees east longitude. When writing the longitude to extended parameter RAM with the `WriteLocation` procedure, you must convert this value to a `Fract` data type. (For example, a `Fract` value of 1.0 equals 90 degrees; –1.0 equals –90 degrees; and –2.0 equals –180 degrees.) For an example that shows this conversion process, see Listing 4-8 on page 4-19. For more information on the `Fract` data type, see the chapter "Mathematical and Logical Utilities" in this book.

`dlsDelta` A signed byte value representing the hour offset for daylight saving time. This field is a 1-byte value contained in a long word. It should be preserved when writing `gmtDelta`. See Listing 4-10 on page 4-21 for an example that writes `gmtDelta` while preserving `dlsDelta`.

`gmtDelta` The Greenwich mean time (GMT). For example, Copenhagen, Denmark is at 1 hour west of GMT. This field is a 3-byte value contained in a long word. In addition, the top byte of this field should be masked off when writing because it is reserved. See Listing 4-9 on page 4-20 and Listing 4-10 on page 4-21 for code examples that get and set `gmtDelta` properly.

The `ReadLocation` and `WriteLocation` procedures use the geographic location record to read and store the geographic location and time zone information in extended parameter RAM. If the geographic location record has never been set, all fields contain 0.

The Toggle Parameter Block

The `ToggleDate` function exchanges information with your application using the toggle parameter block, defined by the `TogglePB` data type.

```
TYPE TogglePB =
RECORD
   togFlags:      LongInt;    {flags}
   amChars:       ResType;    {A.M. characters from 'itl0' }
                              { resource, but made uppercase}
   pmChars:       ResType;    {P.M. characters from 'itl0' }
                              { resource, but made uppercase}
   reserved:      ARRAY[0..3] OF LongInt; {reserved}
END;
```

Field descriptions

`togFlags` The high-order word of this field contains flags that specify special conditions for the `ToggleDate` function:

```
genCdevRangeBit   = 27;    {restrict date/time to }
                           { range used by }
                           { General Controls }
                           { control panel}
togDelta12HourBit = 28;    {if modifying hour }
                           { up/down, restrict to }
                           { 12-hour range}
togCharZCycleBit  = 29;    {modifier for }
                           { togChar12HourBit to }
                           { accept hours }
                           { 0…11 only}
togChar12HourBit     = 30; {if modifying hour by }
                           { char, accept hours }
                           { 1…12 only}
smallDateBit         = 31; {restrict valid }
                           { date/time to }
                           { range of Time global}
```

`genCdevRangeBit`
If this bit is set in addition to `smallDateBit`, then the date range is restricted to that used by the General Controls control panel—

January 1, 1920 to December 31, 2019 in the Gregorian calendar (the routine works correctly for other calendars as well). For dates outside this range but within the range specified by the system global variable `Time`—January 1, 1904 to February 6, 2040 in the Gregorian calendar—`ToggleDate` adds or subtracts 100 years to bring the dates into the range of the General Controls control panel if these bits are set. The `ToggleDate` function returns an error if the `smallDateBit` is set and the date is outside the range specified by the system global variable `Time`. This bit works with system software version 6.0.4 and later.

`togDelta12HourBit`
If this bit is set, modifying the hour up or down is limited to a 12-hour range. For example, increasing by one from 11 produces 0, increasing by one from 23 produces 12, and so on. This bit works with system software version 6.0.4 and later.

`togCharZCycleBit`
If this bit is set, the input character is treated as if it modifies an hour whose value is in the range 0–11. If this bit is not set, the input character is treated as if it modifies an hour whose value is in the range 12, 1–11. This bit works with system software version 6.0.4 and later.

`togChar12HourBit`
If this bit is set, modifying the hour by character is limited to the 12-hour range defined by `togCharZCycleBit`, mapped to the appropriate half of the 24-hour range, as determined by the `pm` field. This bit works with system software version 6.0.4 and later.

`smallDateBit`
If this bit is set, the valid date and time are restricted to the range of the system global variable `Time`—that is, between midnight on January 1, 1904 and 6:28:15 A.M. on February 6, 2040.

The low-order word of this field contains masks representing fields to be checked by the `ValidDate` function. Each mask corresponds to a value in the enumerated type `LongDateField`. You can set this field to check the `era` through `second` fields by using the predeclared constant `dateStdMask`. The following constants specify the `LongDateRec` fields for the `ValidDate` function to check.

```
CONST
eraMask             = $0001;    {verify the era}
yearMask            = $0002;    {verify the year}
monthMask           = $0004;    {verify the month}
dayMask             = $0008;    {verify the day}
hourMask            = $0010;    {verify the hour}
minuteMask          = $0020;    {verify the }
```

```
                                                { minute}
                secondMask              = $0040;    {verify the }
                                                    { second}
                dateStdMask             = $007F;    {verify the era }
                                                    { through second}
                dayOfWeekMask           = $0080;    {verify the day }
                                                    { of the week}
                dayOfYearMask           = $0100;    {verify the day }
                                                    { of the year}
                weekOfYearMask          = $0200;    {verify the week }
                                                    { of the year}
                pmMask                  = $0400;    {verify the }
                                                    { evening (P.M.)}
```

`amChars` The trailing string to display for morning (for example, A.M.). This string is read from the numeric-format resource (resource type `'itl0'`) of the current script system.

`pmChars` The trailing to display for evening (for example, P.M.). This string is read from the numeric-format resource (resource type `'itl0'`) of the current script system.

`reserved` Reserved. Set each of the three elements of this field to 0.

The Unsigned Wide Record

The Microseconds procedure uses the unsigned wide record to return the number of microseconds elapsed since system startup time. The `UnsignedWide` data type defines the format for the unsigned wide record.

```
UnsignedWide =    {Microseconds procedure return type}
   PACKED RECORD
      hi:       LongInt;          {high-order 32 bits}
      lo:       LongInt;          {low-order 32 bits}
   END;
```

Field descriptions

`hi` The high-order 32 bits

`lo` The low-order 32 bits

Routines

The Date, Time, and Measurement Utilities provide routines you can use to read and write current date-time information, convert between internal date and time formats (for example, you can access date-time information as a number of seconds elapsed since midnight, January 1, 1904 or as a date and time), manipulate date-time information, read and write location information, and determine the current measurement system.

Some of the routines provided by the Date, Time, and Measurement Utilities were previously associated with the Script Manager or the International Utilities Package. In addition, some routines have been renamed to reflect their functions more clearly. You can access the renamed routines using more than one spelling of the routine's name, depending on the interface files supported by your development environment. For example, the `IsMetric` function is also available as the `IUMetric` function. Table 4-4 provides a summary of these changes.

Table 4-4 Renamed and relocated routines

Current name	Previous name	Former location
`DateToSeconds`	`Date2Secs`	(Unchanged)
`IsMetric`	`IUMetric`	International Utilities Package
`LongDateToSeconds`	`LongDate2Secs`	Script Manager
`LongSecondsToDate`	`LongSecs2Date`	Script Manager
`ReadLocation`	`ReadLocation`	Script Manager
`SecondsToDate`	`Secs2Date`	(Unchanged)
`ToggleDate`	`ToggleDate`	Script Manager
`ValidDate`	`ValidDate`	Script Manager
`WriteLocation`	`WriteLocation`	Script Manager

Getting the Current Date and Time

At system startup time, system software uses the `ReadDateTime` function to copy the current date-time information from the clock chip into low memory. You can access this date-time information as the number of seconds elapsed since midnight of January 1, 1904 or as a date and time. To obtain the current date-time information expressed as the number of seconds elapsed since midnight of January 1, 1904, use the `GetDateTime` procedure. To obtain the current date-time information expressed as a date and time, use the `GetTime` procedure.

IMPORTANT

If an application disables interrupts for longer than a second, the date-time information returned by the `GetDateTime` and `GetTime` procedures might not be exact. The `GetDateTime` and `GetTime` procedures are intended to provide fairly accurate time information, but not scientifically precise data. ▲

ReadDateTime

System software uses at system startup time the `ReadDateTime` function to copy the date-time information from the clock chip into low memory. Your application should never need to use this function.

```
FUNCTION ReadDateTime (VAR time: LongInt): OSErr;
```

`time` On return, the current time expressed as the number of seconds elapsed since midnight, January 1, 1904.

DESCRIPTION

The `ReadDateTime` function copies the current date-time information from the clock chip into low memory. It then returns in the `time` parameter a copy of the date-time information, expressed as the number of seconds elapsed since midnight, January 1, 1904.

The low-memory copy of the date and time information is accessible through the global variable `Time`.

If the clock chip cannot be read, `ReadDateTime` returns the `clkRdErr` result code. The operation might fail if the clock chip is damaged. Otherwise, the function returns the `noErr` result code.

ASSEMBLY-LANGUAGE INFORMATION

You must set up register A0 with a pointer to a long integer in which you wish to store the current date-time information. On exit, register A0 contains the same pointer to the now-changed long integer, and register D0 contains the result code.

The registers on entry and exit for this routine are

Registers on entry

A0 Pointer to long word

Registers on exit

A0 Pointer to current time

D0 Result code

RESULT CODES

`noErr`	0	No error
`clkRdErr`	–85	Unable to read clock

GetDateTime

You can use the `GetDateTime` procedure to obtain the current date-time information, expressed as the number of seconds elapsed since midnight, January 1, 1904.

```
PROCEDURE GetDateTime (VAR secs: LongInt);
```

`secs` On return, the number of seconds elapsed since midnight, January 1, 1904.

DESCRIPTION

The `GetDateTime` procedure returns in the `secs` parameter the number of seconds elapsed since midnight, January 1, 1904.

The low-memory copy of the date and time information (expressed as the number of seconds elapsed since midnight, January 1, 1904) is also accessible through the global variable `Time`.

SEE ALSO

For an example that uses the `GetDateTime` procedure to get the current date and time, see Listing 4-1 on page 4-10.

GetTime

You can use the `GetTime` procedure to obtain the current date-time information, expressed as a date and time.

```
PROCEDURE GetTime (VAR d: DateTimeRec);
```

`d` On return, the fields of the date-time record contain the current date and time.

DESCRIPTION

The `GetTime` procedure returns in the `d` parameter the current date and time. The `GetTime` procedure first calls the `GetDateTime` procedure to obtain the number of seconds elapsed since midnight, January 1, 1904. It then calls the `SecondsToDate` procedure to convert the number of seconds (returned by the `GetDateTime` procedure) into a date and time.

As an alternative to using the `GetTime` procedure, you can pass the value of the global variable `Time` to the `SecondsToDate` procedure; a `SecondsToDate(Time)` procedure call is identical to a `GetTime(d)` procedure call.

SEE ALSO

For more information about the `SecondsToDate` procedure, see page 4-38. The `GetDateTime` procedure is described on page 4-35. For sample code that uses the `GetTime` procedure to get the current date and time, see Listing 4-2 on page 4-10. The date-time record is described in detail beginning on page 4-23.

Setting the Current Date and Time

You can modify the date-time information stored in the clock chip by using the `SetDateTime` function or the `SetTime` procedure. The two routines differ in the type of arguments they require. The `SetDateTime` function requires that the new date-time information be expressed as the number of seconds elapsed since midnight of January 1, 1904 (using a value of type `LongInt`). The `SetTime` procedure requires that the new date-time information be expressed as a date and time (using a value of type `DateTimeRec`).

IMPORTANT

Users can change the current date and time stored in both the system global variable `Time` and in the clock chip by using the General Controls control panel, Date & Time control panel, or the Alarm Clock desk accessory. In general, your application should not directly change the current date-time information. If your application does need to modify the current date-time information, it should instruct the user how to change the date and time. ▲

SetDateTime

You can use the `SetDateTime` function to modify the date-time information stored in the clock chip. The `SetDateTime` function requires that the new date-time information be passed to the function as the number of seconds elapsed since midnight, January 1, 1904.

```
FUNCTION SetDateTime (time: LongInt): OSErr;
```

`time` The number of seconds elapsed since midnight, January 1, 1904; this value is written to the clock chip.

DESCRIPTION

The `SetDateTime` function writes the number of seconds, specified by the `time` parameter, to the clock chip. The `SetDateTime` function also updates the low-memory copy of the date-time information.

The `SetDateTime` function attempts to verify the value written by reading it back in and comparing it to the value in the low-memory copy. If a problem occurs, the `SetDateTime` function returns either the `clkRdErr` result code, because the clock chip

could not be read, or the `clkWrErr` result code, because the time written to the clock chip could not be verified. Otherwise, the function returns the `noErr` result code.

ASSEMBLY-LANGUAGE INFORMATION

You must set up register D0 with the number of seconds to which you wish to change the clock chip. When the `SetDateTime` function returns, register D0 contains the result code.

The registers on entry and exit for this routine are

Registers on entry

D0 Seconds elapsed since midnight, January 1, 1904

Registers on exit

D0 Result code

RESULT CODES

`noErr`	0	No error
`clkRdErr`	–85	Unable to read clock
`clkWrErr`	–86	Time written did not verify

SEE ALSO

For sample code that uses the `SetDateTime` function to write date-time information (represented as a number of seconds) to the clock-chip, see Listing 4-3 on page 4-11.

SetTime

You can use the `SetTime` procedure to modify the date-time information in the clock chip. The `SetTime` requires that the new date-time information be passed to the function as a date and time.

```
PROCEDURE SetTime (d: DateTimeRec);
```

`d` The date and time to which to set the clock chip.

DESCRIPTION

The `SetTime` procedure writes the date and time specified by the `d` parameter to the clock chip. The `SetTime` procedure first converts the date and time to the number of seconds elapsed since midnight, January 1, 1904 (by calling the `DateToSeconds` procedure). It then writes these seconds to the clock chip and to the system global variable `Time` (by calling the `SetDateTime` function).

As an alternative to using the `SetTime` procedure, you can use the `DateToSeconds` and `SetDateTime` routines.

Note
The `SetTime` procedure does not return a result code. If you need to know whether an attempt to change the date and time information in the clock chip is successful, you must use the `SetDateTime` function. ◆

SEE ALSO

See page 4-23 for a description of the fields of a date-time record. For more information on the `DateToSeconds` procedure, see page 4-39. The `SetDateTime` function is described on page 4-36. For sample code that uses the `SetTime` procedure to write date-time information (represented as a date and time) to the clock-chip, see Listing 4-4 on page 4-11.

Converting Between Date-Time Formats

The Date, Time, and Measurement Utilities provide two procedure, `SecondsToDate` and `DateToSeconds`, that you can use to convert between date-time formats. You can convert a number of seconds to a date and time and a date and time to a number of seconds.

If you use a standard date-time value (used to access a number of seconds) or a date-time record (used to access a date and time) to access date-time information, you can use the `SecondsToDate` and `DateToSeconds` procedures to convert between these date-time formats. Use the `SecondsToDate` procedure to convert a number of seconds to a date and time, and use the `DateToSeconds` procedure to convert a date and time to a number of seconds.

Note
The system software uses the `SecondsToDate` and `DateToSeconds` procedures provided by the current script system. ◆

SecondsToDate

You can use the `SecondsToDate` procedure to convert a number of seconds elapsed since midnight, January 1, 1904 to a date and time.

```
PROCEDURE SecondsToDate (s: LongInt; VAR d: DateTimeRec);
```

`s`	The number of seconds elapsed since midnight, January 1, 1904.
`d`	On return, the fields of the date-time record that contain the date and time corresponding to the value indicated in the `s` parameter.

DESCRIPTION

The `SecondsToDate` procedure converts the number of seconds, specified in the `s` parameter, to a date and time. The date and time values are returned in the `d` parameter.

The `SecondsToDate` procedure is also available as the `Secs2Date` procedure.

ASSEMBLY-LANGUAGE INFORMATION

The registers on entry and exit for this routine are

Registers on entry

D0 Seconds since midnight, January 1, 1904

A0 Pointer to a date-time record

Registers on exit

A0 Pointer to a date-time record

SEE ALSO

For a complete description of the date-time record, see page 4-23.

DateToSeconds

You can use the `DateToSeconds` procedure to convert a date and time to a number of seconds elapsed since midnight, January 1, 1904.

```
PROCEDURE DateToSeconds (d: DateTimeRec; VAR s: LongInt);
```

`d` The date-time record containing the date and time to convert.

`s` On return, the number of seconds elapsed between midnight, January 1, 1904, and the time specified in the `d` parameter.

DESCRIPTION

The `DateToSeconds` procedure converts the date and time specified in the `d` parameter to the number of seconds elapsed since midnight, January 1, 1904. The number of seconds are returned in the `s` parameter. For example, specifying a date and time of 5:50 A.M. on June 13, 1990 results in 41627 being returned in the `s` parameter.

The `DateToSeconds` procedure is also available as the `Date2Secs` procedure.

ASSEMBLY-LANGUAGE INFORMATION

You must set up register A0 with a pointer to the date and time record containing the date and time you wish to convert. When `DateToSeconds` returns, register D0 contains a long integer representing the converted date and time.

The registers on entry and exit for this routine are

Registers on entry

A0 Pointer to date-time record

Registers on exit

D0 Corresponding seconds since midnight, January 1, 1904

SEE ALSO

For a complete description of the date-time record, see page 4-23.

Converting Between Long Date-Time Format

The Date, Time, and Measurement Utilities provide two procedures, `LongSecondsToDate` and `LongDateToSeconds`, that you can use to convert between long date-time formats. You can convert a number of seconds to a date and time and a date and time to a number of seconds.

If you use a long date-time value (used to access a number of seconds) or a long date-time record (used to access a date and time) to access date-time information, you can use the `LongSecondsToDate` and `LongDateToSeconds` procedures to convert between these date-time formats. Use the `LongSecondsToDate` procedure to convert a number of seconds to a date and time, and use the `LongDateToSeconds` procedure to convert a date and time to a number of seconds.

Note

The system software uses the `LongSecondsToDate` and `LongDateToSeconds` procedures provided by the current script system. ◆

LongSecondsToDate

You can use the `LongSecondsToDate` procedure to convert the number of seconds elapsed since midnight, January 1, 1904 to a date and time.

```
PROCEDURE LongSecondsToDate (lSecs: LongDateTime;
                             VAR lDate: LongDateRec);
```

`lSecs` The number of seconds elapsed since midnight, January 1, 1904.

`lDate` On return, the fields of the long date-time record that contain the date and time corresponding to the value indicated in the `lSecs` parameter.

DESCRIPTION

The `LongSecondsToDate` procedure converts the representation of the date-time information from a number of seconds, specified in the `lSecs` parameter, to a date and time. The date and time are returned in the `lDate` parameter as values in the date-time record. For example, specifying the number of seconds 41627 results in the date and time 5:50 A.M. on June 13, 1990 being returned in the `lDate` parameter.

The `LongSecondsToDate` procedure is also available as the `LongSecs2Date` procedure.

SEE ALSO

To learn more about the long date-time value, see the section page 4-25. For more information on the long date-time record, see page 4-26.

LongDateToSeconds

You can use the `LongDateToSeconds` procedure to convert a date and time to the number of seconds elapsed since midnight, January 1, 1904.

```
PROCEDURE LongDateToSeconds (lDate: LongDateRec;
                             VAR lSecs: LongDateTime);
```

`lDate`	The long date-time record containing the date and time to convert.
`lSecs`	On return, the number of seconds elapsed since midnight, January 1, 1904, and the time specified in the `lDate` parameter.

DESCRIPTION

The `LongDateToSeconds` procedure converts the representation of the date-time information from a date and time, specified in the `lDate` parameter, to the number of seconds elapsed since midnight, January 1, 1904. The number of seconds are returned as a long date-time value in the `lSecs` parameter. For example, specifying the date and time 5:50 A.M. on June 13, 1990 results in 41627 being returned in the `lSecs` parameter.

The `LongDateToSeconds` procedure is also available as the `LongDate2Secs` procedure.

SEE ALSO

To learn more about the long date-time value, see page 4-25. For more information on the long date-time record, see page 4-26.

Modifying and Verifying Long Date-Time Records

You can modify and verify the values in a long date-time record by using the `ToggleDate` function and the `ValidDate` function, respectively.

The `ToggleDate` function accepts a pointer to a toggle parameter block as a parameter. Information about the fields in the toggle parameter block appears in the following format:

Parameter block

→	`input1`	`LongInt`	Input parameter comment.
←	`output1`	`LongInt`	Output parameter comment.

The arrow on the far left indicates whether the field is an input or output parameter. You must supply values for all input parameters. The routine returns values in the output parameters. The next column shows the field name as defined in the MPW interface files, followed by the data type of that field. This matches the MPW interface name of the data type as shown in the parameter block. The fourth column contains a comment about or brief definition of the field.

ToggleDate

You can use the `ToggleDate` function to modify a date and time, by modifying one specific component of a date and time (day, hour, minute, seconds, day of week, and so on). For example, you can use the `ToggleDate` function to increase a date and time by one minute, decrease a date and time by one minute, or explicitly add or subtract a number of seconds to or from a date and time.

```
FUNCTION ToggleDate (VAR lSecs: LongDateTime;
                     field: LongDateField; delta: DateDelta;
                     ch: Integer; params: TogglePB)
                     : ToggleResults;
```

`lSecs` The date-time information to modify, expressed as the number of seconds elapsed since midnight, January 1, 1904.

`field` The name of the field in the date-time record you want modify. Use one of the `LongDateField` enumeration constants for the value of this parameter.

`delta` A signed byte specifying the action you want to perform on the value specified in the `field` parameter. Set `delta` to 1, to increase the value in the field by 1. Set `delta` to -1, to decrease the value of the field by 1. Set `delta` to 0. If you want to set the value of the field explicitly; pass the new value through the `ch` field, described next.

ch	If the value in the delta field is 0, the value of the field in the date-time record (specified by the field parameter) is set to the value in the ch parameter. If the value in the delta field is not equal to 0, the value in the ch parameter is ignored.
params	The settings of the toggle parameter block settings. Note that you are responsible for setting this field.

Parameter block

→	togFlags	LongInt	The fields to be checked by the ValidDate function.
→	amChars	ResType	A.M. characters from 'itl0' resource.
→	pmChars	ResType	P.M. characters from 'itl0' resource.
→	reserved	ARRAY [0…3] OF LongInt	Reserved; set each element to 0.

DESCRIPTION

The ToggleDate function first converts the number of seconds, specified in the lSecs parameter, to a date and time—making each component of the date and time (day, minute, seconds, day of week, and so on) available through a long date-time record. The ToggleDate function then modifies the value of the field, specified by the field parameter. If the value in the delta field is greater than 0, the value of the field (specified in the field parameter) increases by 1; if the value in the delta field is less than 0, the value of the field decreases by 1; and if the value of delta is 0, the value of the field is explicitly set to the value specified in the ch field.

After the ToggleDate function modifies the field, it calls the ValidDate function. The ValidDate function checks the long date-time record for correctness, using the values of the togFlags field in the toggle parameter block that the ToggleDate function passes to it. If any of the record fields are invalid, the ValidDate function returns a LongDateField value corresponding to the field in error. Otherwise, it returns the result code for validDateFields. Note that ValidDate reports only the least significant erroneous field.

After the ToggleDate function checks the validity of the modified field, it converts the modified date and time back into a number of seconds (the number of seconds elapsed since midnight, January 1, 1904) and returns these seconds in the lSecs parameter.

The following constants specify the LongDateRec fields for the ValidDate function to check:

```
CONST
   eraMask              = $0001;    {verify the era}
   yearMask             = $0002;    {verify the year}
   monthMask            = $0004;    {verify the month}
   dayMask              = $0008;    {verify the day}
   hourMask             = $0010;    {verify the hour}
   minuteMask           = $0020;    {verify the minute}
   secondMask           = $0040;    {verify the second}
```

```
dateStdMask          = $007F;    {verify the era through second}
dayOfWeekMask        = $0080;    {verify the day of the week}
dayOfYearMask        = $0100;    {verify the day of the year}
weekOfYearMask       = $0200;    {verify the week of the year}
pmMask               = $0400;    {verify the evening (P.M.)}
```

SPECIAL CONSIDERATIONS

Although `ToggleDate` does not move or purge memory, you should not call it at interrupt time.

RESULT CODES

The `ToggleDate` function returns its own set of result codes. The `ToggleResults` data type defines the result code of the `ToggleDate` function:

```
TYPE ToggleResults = Integer; {ToggleDate function return type}
```

The following list gives the result codes defined for this function:

`toggleUndefined`	0	Undefined error
`toggleOK`	1	No error
`toggleBadField`	2	Invalid field number
`toggleBadDelta`	3	Invalid delta value
`toggleBadChar`	4	Invalid character
`toggleUnknown`	5	Unknown error
`toggleBadNum`	6	Tried to use character as number
`toggleOutOfRange`	7	Out of range (synonym for `toggleErr3`)
`toggleErr3`	7	Reserved
`toggleErr4`	8	Reserved
`toggleErr5`	9	Reserved

SEE ALSO

To learn more about the `LongDateTime` data type, see page 4-25. For more information on the `LongDateRec` structure, see page 4-26. The toggle parameter block record is described on page 4-30.

For more information about the `GetIntlResource` function, see the chapter "Script Manager" in *Inside Macintosh: Text*. For details on the `UppercaseText` procedure, see the chapter "Text Utilities" in *Inside Macintosh: Text*. The `ValidDate` function is described next.

ValidDate

You can use the `ValidDate` function to verify specific date and time values in a long date-time record.

```
FUNCTION ValidDate (VAR vDate: LongDateRec; flags: LongInt;
                    VAR newSecs: LongDateTime): Integer;
```

`vDate` The long date-time record whose fields you want to verify.

`flags` The fields that you want to verify in the long date-time record.

`newSecs` The date-time information, passed by the `ToggleDate` function, that you want to verify.

DESCRIPTION

The `ValidDate` function verifies the fields, specified by the `flags` parameter, in the long date-time record specified by the `vDate` parameter. If any of the specified fields contain invalid values, the `ValidDate` function returns a `LongDateField` value indicating the field in error. Otherwise, it returns the constant `validDateFields`. Note that `ValidDate` reports only the least significant erroneous field.

The following constants specify the `LongDateRec` fields for the `ValidDate` function to check:

```
CONST
   eraMask              = $0001;    {verify the era}
   yearMask             = $0002;    {verify the year}
   monthMask            = $0004;    {verify the month}
   dayMask              = $0008;    {verify the day}
   hourMask             = $0010;    {verify the hour}
   minuteMask           = $0020;    {verify the minute}
   secondMask           = $0040;    {verify the second}
   dateStdMask          = $007F;    {verify the era through }
                                    { second}
   dayOfWeekMask        = $0080;    {verify the day of the week}
   dayOfYearMask        = $0100;    {verify the day of the year}
   weekOfYearMask       = $0200;    {verify the week of the year}
   pmMask               = $0400;    {verify the evening (P.M.)}
```

SPECIAL CONSIDERATIONS

Although `ValidDate` does not move or purge memory, you should not call it at interrupt time.

SEE ALSO

To learn more about the `LongDateTime` data type, see page 4-25. For more information on the long date-time record, see page 4-26. The `ToggleDate` function is described on page 4-42. The enumerated type `LongDateField` is described on page 4-29.

Reading and Writing Location Data

You can read and set geographic location and time-zone information using the `Readlocation` and `WriteLocation` procedures.

ReadLocation

You can use the `ReadLocation` procedure to get information about a geographic location or time zone.

```
PROCEDURE ReadLocation (VAR loc: MachineLocation);
```

`loc` On return, the fields of the geographic location record containing the geographic location and the time-zone information.

DESCRIPTION

The `ReadLocation` procedure reads the stored geographic location and time zone of the Macintosh computer from extended parameter RAM and returns it in the `loc` parameter.

You can get values for the latitude, longitude, daylight savings time (DST), or Greenwich mean time (GMT). If the geographic location record has never been set, all fields contain 0.

The latitude and longitude are stored as `Fract` values, giving accuracy to within one foot. For example, `a Fract` value of 1.0 equals 90 degrees; –1.0 equals –90 degrees; and –2.0 equals –180 degrees.

To convert these values to a degrees format, you need to convert the `Fract` values first to the `Fixed` data type, then to the `LongInt` data type. You can use the Mathematical and Logical Utilities routines `Fract2Fix` and `Fix2Long` to accomplish this task.

The DST value is a signed byte value that you can use to specify the offset for the `hour` field—whether to add one hour, subtract one hour, or make no change at all.

The GMT value is in seconds east of GMT. For example, San Francisco is at –28,800 seconds (8 hours * 3,600 seconds per hour) east of GMT. The `gmtDelta` field is a 3-byte value contained in a long word, so you must take care to get it properly.

SPECIAL CONSIDERATIONS

Although the `ReadLocation` procedure does not move or purge memory, you should not call it at interrupt time.

SEE ALSO

For more information on the geographic location record, see page 4-29. For an example of how to use the `ReadLocation` procedure to get latitude and longitude, see Listing 4-8 on page 4-19. Listing 4-9 on page 4-20 shows an application-defined procedure for obtaining the value of `gmtDelta`.

For more information on the `Fract` data type and the conversion routines `Long2Fix`, `Fix2Fract`, `Fract2Fix`, and `Fix2Long`, see the chapter "Mathematical and Logical Utilities" in this book.

WriteLocation

You can use the `WriteLocation` procedure to change the geographic location or time-zone information stored in extended parameter RAM.

```
PROCEDURE WriteLocation (loc: MachineLocation);
```

`loc` The geographic location and time-zone information to write to the extended parameter RAM.

DESCRIPTION

The `WriteLocation` procedure takes the geographic location and time-zone information, specified in the `loc` parameter, and writes it to the extended parameter RAM.

The latitude and longitude are stored in the geographic location record as `Fract` values, giving accuracy to within 1 foot. For example, a `Fract` value of 1.0 equals 90 degrees; –1.0 equals –90 degrees; and –2.0 equals –180 degrees.

To store latitude and longitude values, you need to convert them first to the `Fixed` data type, then to the `Fract` data type. You can use the Operating System Utilities routines `Long2Fix` and `Fix2Fract` to accomplish this task. Listing 4-8 on page 4-19 shows a procedure that converts San Francisco's latitude and longitude to `Fract` values, then writes the `Fract` values to extended parameter RAM using the `WriteLocation` procedure.

The daylight savings time value is a signed byte value that you can use to specify the offset for the `hour` field—whether to add one hour, subtract one hour, or make no change at all.

The Greenwich mean time value is in seconds east of GMT. For example, San Francisco is at –28,800 seconds (8 hours * 3,600 seconds per hour) east of GMT. The `gmtDelta` field is

a 3-byte value contained in a long word, so you must take care to set it properly. When writing `gmtDelta`, you should mask off the top byte because it is reserved. In addition, you should preserve the value of `dlsDelta`. Listing 4-10 on page 4-21 shows a procedure that writes `gmtDelta`, with the top byte masked off, while preserving the value of `dlsDelta`.

SPECIAL CONSIDERATIONS

Although `WriteLocation` does not move or purge memory, you should not call it at interrupt time.

SEE ALSO

For more information on the geographic location record, see page 4-29. For more information on the `Fract` data type and the conversion routines `Long2Fix`, `Fix2Fract`, `Fract2Fix`, and `Fix2Long`, see the chapter "Mathematical and Logical Utilities" in this book.

Determining the Measurement System

You can determine the type of measurement system that is used by the current script system by the using the `IsMetric` function.

IsMetric

You can use the `IsMetric` function to determine whether the current script system is using the metric system (also called the International System of Units) or the English system of measurement (also called the British imperial system). The `IsMetric` function is also available as the `IUMetric` function.

```
FUNCTION IsMetric: BOOLEAN;
```

DESCRIPTION

The `IsMetric` function examines the `metricSys` field of the numeric-format resource (resource type `'itl0'`) to determine if the current script is using the metric system. A value of 255 in the `metricSys` field indicates that the metric system (centimeters, kilometers, milligrams, degrees Celsius, and so on) is being used. In this case, `the IsMetric` function returns a value of `TRUE`. A value of 0 in the `metricSys` field indicates that the English system of measurement (inches, miles, ounces, degrees Fahrenheit, and so on) is used. In that case, the `IsMetric` function returns a value of `FALSE`.

If you want to use units of measurement different from that of the current script, you need to override the value of the `metricSys` field in the current numeric-format

resource (resource type `'itl0'`). You can do this by using your own version of the numeric-format resource instead of the current script system's default international resource.

SPECIAL CONSIDERATIONS

The `IsMetric` function may move or purge blocks in the heap; calling it may cause problems if you've dereferenced a handle. You should not call this function from within interrupt code, such as in a completion routine or a VBL task.

SEE ALSO

For a complete description of the international numeric-format resource (resource type `'itl0'`) and how to use it, see the appendix "International Resources" in *Inside Macintosh: Text.*

For information on how to replace a script system's default international resources, see the chapter "Script Manager" in *Inside Macintosh: Text*.

Measuring Time

You can measure the number of elapsed microseconds since system startup, using the `Microseconds` procedure.

Microseconds

You can use the `Microseconds` procedure to determine the number of microseconds that have elapsed since system startup time.

```
PROCEDURE Microseconds (VAR microTickCount: UnsignedWide);
```

`microsecondCount`
The number of microseconds elapsed since system startup.

DESCRIPTION

The `Microseconds` procedure returns, in the `microsecondCount` parameter, the number of microseconds that has elapsed since system startup time.

SEE ALSO

For information about the return type for this procedure—the `UnsignedWide` record—see page 4-32. For an example of how to use the `Microseconds` procedure, see Listing 4-11 on page 4-21.

Summary of the Date, Time, and Measurement Utilities

Pascal Summary

Constants

```
CONST
   {date equates for ToggleDate control bits}
   validDateFields      = -1;       {date fields are valid}
   genCdevRangeBit      = 27;       {restrict date/time to range used by }
                                    { General Controls control panel}
   togDelta12HourBit    = 28;       {if toggling hour up/down, restrict to }
                                    { 12-hour range}
   togCharZCycleBit     = 29;       {modifier for togChar12HourBit to }
                                    { accept hours 0..11 only}
   togChar12HourBit     = 30;       {if toggling hour by char, accept }
                                    { hours 1..12 only}
   smallDateBit         = 31;       {restrict valid date/time to range }
                                    { of Time global}

   {long date-time record field masks}
   eraMask              = $0001;    {era}
   yearMask             = $0002;    {year}
   monthMask            = $0004;    {month}
   dayMask              = $0008;    {day}
   hourMask             = $0010;    {hour}
   minuteMask           = $0020;    {minute}
   secondMask           = $0040;    {second}
   dayOfWeekMask        = $0080;    {day of the week}
   dayOfYearMask        = $0100;    {day of the year}
   weekOfYearMask       = $0200;    {week of the year}
   pmMask               = $0400;    {evening (P.M.)}

   {default value for togFlags field in the toggle parameter block }
   { and default value for the flags parameter passed to the Verify function}
   dateStdMask          = $007F;    {default value for checking era }
                                    { through second fields}
```

Data Types

```
TYPE

   DateTimeRec =              {date-time record}
   RECORD
      year:       Integer;    {year}
      month:      Integer;    {month}
      day:        Integer;    {day of the month}
      hour:       Integer;    {hour}
      minute:     Integer;    {minute}
      second:     Integer;    {second}
      dayOfWeek:  Integer;    {day of the week}
   END;

   LongDateField =   {long date field enumeration}
                     (eraField, yearField, monthField, dayField,
                     hourField, minuteField, secondField,dayOfWeekField,
                     dayOfYearField,weekOfYearField, pmField, res1Field,
                     res2Field, res3Field);
   LongDateTime = comp;       {date and time in 64-bit SANE comp format}

   LongDateCvt =              {long date-time conversion record}
   RECORD
      CASE Integer OF
         0:
            (c:      comp);   {copy field into a variable of type }
                              { LongDateTime}
         1:
            (lHigh:  LongInt; {high-order 32 bits}
             lLow:   LongInt);{low-order 32 bits}
   END;

   LongDateRec =              {long date-time record}
   RECORD
      CASE Integer OF
         0:
            (era:          Integer;       {era}
             year:         Integer;       {year}
             month:        Integer;       {month}
             day:          Integer;       {day of the month}
             hour:         Integer;       {hour}
             minute:       Integer;       {minute}
             second:       Integer;       {second}
```

```
            dayOfWeek:    Integer;       {day of the week}
            dayOfYear:    Integer;       {day of the year}
            weekOfYear:   Integer;       {week of the year}
            pm:           Integer;       {half of day--0 for morning, }
                                         { 1 for evening}
            res1:         Integer;       {reserved}
            res2:         Integer;       {reserved}
            res3:         Integer);      {reserved}
         1:                              {index by LongDateField}
            (list:        ARRAY[0..13] OF Integer);
         2:
            (eraAlt:      Integer;       {era}
             oldDate:     DateTimeRec);  {date-time record}
END;

TogglePB =                   {toggle parameter block}
RECORD
   togFlags:      LongInt;    {flags}
   amChars:       ResType;    {from 'itl0' resource, but made uppercase}
   pmChars:       ResType;    {from 'itl0' resource, but made uppercase}
                              {reserved}
   reserved:      ARRAY[0..3] OF LongInt;

END;

ToggleResults = Integer;         {ToggleDate function return type}

DateDelta = SignedByte;          {ToggleDate function delta field type}


MachineLocation =                   {geographic location record}
RECORD
   latitude:           Fract;      {latitude}
   longitude:          Fract;      {longitude}
   CASE Integer OF
      0:
         (dlsDelta:    SignedByte);{daylight savings time}
      1:
         (gmtDelta:    LongInt);   {Greenwich mean time}
END;
```

```
UnsignedWide =          {Microseconds procedure return type}
PACKED RECORD
hi:   longInt;       {high-order 32 bits}
lo:   longInt;       {low-order 32 bits}
END;
```

Routines

Getting the Current Date and Time

```
FUNCTION ReadDateTime       (VAR time: LongInt) : OSErr;
PROCEDURE GetDateTime       (VAR secs: LongInt);
PROCEDURE GetTime           (VAR d: DateTimeRec);
```

Setting the Current Date and Time

```
FUNCTION SetDateTime        (time: LongInt) : OSErr;
PROCEDURE SetTime           (d: DateTimeRec);
```

Converting Between Date-Time Formats

```
{each procedure has two spellings, see Table 4-4 for the alternate spelling}
PROCEDURE SecondsToDate     (secs: LongInt; VAR d: DateTimeRec);
PROCEDURE DateToSeconds     (d: DateTimeRec; VAR secs: LongInt);
```

Converting Between Long Date-Time Formats

```
{each procedure has two spellings, see Table 4-4 for the alternate spelling}
PROCEDURE LongSecondsToDate (VAR lSecs: LongDateTime;
                             VAR lDate: LongDateRec);
PROCEDURE LongDateToSeconds (lDate: LongDateRec; VAR lSecs: LongDateTime);
```

Modifying and Verifying Long Date-Time Records

```
FUNCTION ToggleDate         (VAR lSecs: LongDateTime; field: LongDateField;
                             delta: DateDelta; ch: Integer;
                             params: TogglePB): ToggleResults;
FUNCTION ValidDate          (vDate: LongDateRec; flags: LongInt;
                             VAR newSecs: LongDateTime): Integer;
```

Reading and Writing Location Data

```
PROCEDURE ReadLocation      (VAR loc: MachineLocation);
PROCEDURE WriteLocation     (VAR loc: MachineLocation);
```

Determining the Measurement System

```
{this function has two spellings, see Table 4-4 for the alternate spelling}
FUNCTION IsMetric:          Boolean;
```

Measuring Time

```
PROCEDURE Microseconds      (VAR microTickCount UnsignedWide);
```

C Summary

Constants

```
enum
{
   /*date equates for ToggleDate control bits*/
   validDateFields      = -1,       /*date fields are valid*/
   genCdevRangeBit      = 27,       /*restrict date/time to range used by */
                                    /* General Controls control panel*/
   togDelta12HourBit    = 28,       /*if toggling hour up/down, restrict */
                                    /* to 12-hour range*/
   togCharZCycleBit     = 29,       /*modifier for TogChar12HourBit to */
                                    /* accept hours 0..11 only*/
   togChar12HourBit     = 30,       /*if toggling hour by char, accept */
                                    /* hours 1..12 only*/
   smallDateBit         = 31,       /*restrict valid date/time to range */
                                    /* of Time global*/

   /*long date-time record field masks*/
   eraMask              = 0x0001,   /*era*/
   yearMask             = 0x0002,   /*year*/
   monthMask            = 0x0004,   /*day*/
   dayMask              = 0x0008,   /*month*/
   hourMask             = 0x0010,   /*hour*/
   minuteMask           = 0x0020,   /*minute*/
   secondMask           = 0x0040,   /*second*/
   dayOfWeekMask        = 0x0080,   /*day of the week*/
   dayOfYearMask        = 0x0100,   /*day of the year*/
   weekOfYearMask       = 0x0200,   /*week of the year*/
   pmMask               = 0x0400    /*evening (P.M.)*/
};
```

```
enum
{
   /*default value for togFlags field in the toggle parameter block and */
   /* default value for the flags parameter passed to the Verify function*/
   dateStdMask          = 0x007F,   /*default value for checking era */
                                    /* through second fields*/
};
```

Data Types

```
struct DateTimeRec         /*date-time record*/
{
   short       year;       /*year*/
   short       month;      /*month*/
   short       day;        /*day of the month*/
   short       hour;       /*hour*/
   short       minute;     /*minute*/
   short       second;     /*second*/
   short       dayOfWeek;  /*day of the week*/
};
typedef struct DateTimeRec DateTimeRec;

enum                       /*long date field enumeration*/
{
   eraField, yearField, monthField, dayField, hourField, minuteField,
   secondField,dayOfWeekField, dayOfYearField, weekOfYearField, pmField,
   res1Field, res2Field, res3Field
};
typedef unsigned char LongDateField;

typedef comp LongDateTime;    /*date and time in 64-bit SANE comp format*/

union LongDateCvt             /*long date-time conversion record*/
{
   comp     c;               /*copy field into a LongDateTime variable*/
   struct
   {
      long  lHigh;           /*high-order 32 bits*/
      long  lLow;            /*low-order 32 bits*/
   } hl;
};
typedef union LongDateCvt LongDateCvt;
```

```
union LongDateRec             /*long date-time record*/
{
   struct
   {
      short era;              /*era*/
      short year;             /*year*/
      short month;            /*month*/
      short day;              /*day of the month*/
      short hour;             /*hour*/
      short minute;           /*minute*/
      short second;           /*second*/
      short dayOfWeek;        /*day of the week*/
      short dayOfYear;        /*day of the year*/
      short weekOfYear;       /*week of the year*/
      short pm;               /*half of day--0 for morning, 1 for evening*/
      short res1;             /*reserved*/
      short res2;             /*reserved*/
      short res3;             /*reserved*/
   } ld;
   short list[14];            /*index by LongDateField*/
   struct
   {
      short       eraAlt;     /*era*/
      DateTimeRec oldDate;    /*date-time record*/
   } od;
};
typedef union LongDateRec LongDateRec;

struct TogglePB               /*toggle parameter block*/
{
   long     togFlags;         /*flags*/
   ResType  amChars;          /*from 'itl0' resource, but made uppercase*/
   ResType  pmChars;          /*from 'itl0' resource, but made uppercase*/
   long     reserved[4];      /*reserved*/
};
typedef struct TogglePB TogglePB;

typedef short ToggleResults;  /*ToggleDate function return type*/

typedef char DateDelta;       /*ToggleDate function delta field type*/

struct MachineLocation        /*geographic location record*/
{
   Fract    latitude;         /*latitude*/
```

```
   Fract    longitude;        /*longitude*/
   union
   {
      char  dlsDelta;         /*daylight saving time*/
      long  gmtDelta;         /*Greenwich mean time*/
   } gmtFlags;
};
typedef struct MachineLocation MachineLocation;

struct UnsignedWide           /*Microseconds procedure return type*/
{
   unsigned long     hi;      /*high-order 32 bits*/
   unsigned long     lo;      /*high-order 32 bits*/
};
typedef struct UnsignedWide UnsignedWide;
```

Routines

Getting the Current Date and Time

```
pascal OSErr ReadDateTime   (unsigned long *time);
pascal void GetDateTime     (unsigned long *secs);
pascal void GetTime         (DateTimeRec *d);
```

Setting the Current Date and Time

```
pascal OSErr SetDateTime    (unsigned long time);
pascal void SetTime         (const DateTimeRec *d);
```

Converting Between Date-Time Formats

```
{each procedure has two spellings, see Table 4-4 for the alternate spelling}
pascal void SecondsToDate   (unsigned long secs, DateTimeRec *d);
pascal void DateToSeconds   (const DateTimeRec *d, unsigned long *secs);
```

Converting Between Long Date-Time Formats

```
{each procedure has two spellings, see Table 4-4 for the alternate spelling}
pascal void LongSecondsToDate
                            (LongDateTime *lSecs, LongDateRec *lDate);
pascal void LongDateToSeconds
                            (const LongDateRec *lDate, LongDateTime *lSecs);
```

Modifying and Verifying Long Date-Time Records

```
pascal ToggleResults ToggleDate
                            (LongDateTime *lSecs, LongDateField field,
                             DateDelta delta, short ch,
                             const TogglePB *params);
pascal short ValidDate      (const LongDateRec vDate, long flags,
                              LongDateTime *newSecs);
```

Reading and Writing Location Data

```
pascal void ReadLocation    (MachineLocation *loc);
pascal void WriteLocation   (MachineLocation *loc);
```

Determining the Measurement System

```
{this functiosn has two spellings, see Table 4-4 for the alternate spelling}
pascal Boolean IsMetric     (void);
```

Measuring Time

```
pascal void Microseconds    (UnsignedWide *microTickCount);
```

Assembly-Language Summary

Data Structures

Date-Time Record

0	`dtYear`	word	year
2	`dtMonth`	word	month
4	`dtDay`	word	day of the month
6	`dtHour`	word	hour
8	`dtMinute`	word	minute
10	`dtSecond`	word	second
12	`dtDayOfWeek`	word	day of the week

Long Date Field Enumeration

0	`eraField`	byte	era
1	`yearField`	byte	year
2	`monthField`	byte	month
3	`dayField`	byte	day of the month
4	`hourField`	byte	hour
5	`minuteField`	byte	minute
6	`secondField`	byte	second
7	`dayOfWeekField`	byte	day of the week
8	`dayOfYearField`	byte	day of the year
9	`weekOfYearField`	byte	week of the year
10	`pmField`	byte	pm
11	`res1Field`	byte	reserved
12	`res2Field`	byte	reserved
13	`res3Field`	byte	reserved

Long Date-Time Value

0	`highLong`	long	high-order 32 bits
4	`lowLong`	long	low-order 32 bits

Long Date-Time Record

0	`era`	word	era
2	`year`	word	year
4	`month`	word	month
6	`day`	word	day of the month
8	`hour`	word	hour
10	`minute`	word	minute
12	`second`	word	second
14	`dayOfWeek`	word	day of the week
16	`dayOfYear`	word	day of the year
18	`weekOfYear`	word	week of the year
20	`pm`	word	half of day, morning or evening
22	`ldReserved`	6 bytes	reserved

Geographic Location Record

0	`latitude`	long	latitude
4	`longitude`	long	longitude
8	`dlsDelta`	byte	daylight savings time
9	`gmtDelta`	3 bytes	Greenwich mean time

Toggle Parameter Block

0	`togFlags`	long	flags
2	`amChars`	word	ResType from `'itl0'` made uppercase
4	`pmChars`	word	ResType from `'itl0'` made uppercase
6	`reserved`	word	reserved

Unsigned Wide Record

0	`hi`	long	high-order 32 bits
4	`lo`	long	low-order 32 bits

Global Variables

`Time` The number of seconds since midnight, January 1, 1904

Result Codes

`toggleErr5`	9	Reserved
`toggleErr4`	8	Reserved
`toggleErr3`	7	Reserved
`toggleOutOfRange`	7	Out of range (synonym for `toggleErr3`)
`toggleBadNum`	6	Tried to use character as number
`toggleUnknown`	5	Unknown error
`toggleBadChar`	4	Invalid character
`toggleBadDelta`	3	Invalid delta value
`toggleBadField`	2	Invalid field number
`toggleOK`	1	No error
`toggleUndefined`	0	Undefined error
`noErr`	0	No error
`clkRdErr`	–85	Unable to read clock
`clkWrErr`	–86	Time written did not verify