Dates and Times Chapter Six

6.1 Chapter Overview

This chapter discusses dates and times as a data type. In particular, this chapter discusses the data/time data structures the HLA Standard Library defines and it also discusses date arithmetic and other operations on dates and times.

6.2 Dates

For the first 50 years, or so, of the computer's existence, programmers did not give much thought to date calculations. They either used a date/time package provided with their programming language, or they kludged together their own date processing libraries. It wasn't until the Y2K[1] problem came along that programmers began to give dates serious consideration in their programs. The purpose of this chapter is two-fold. First, this chapter teaches that date manipulation is not as trivial as most people would like to believe – it takes a lot of work to properly compute various date functions. Second, this chapter presents the HLA date and time formats found in the "datetime.hhf" library module. Hopefully this chapter will convince you that considerable thought has gone into the HLA datetime.hhf module so you'll be inclined to use it rather than trying to create your own date/time formats and routines.

Although date and time calculations may seem like they should be trivial, they are, in fact, quite complex. Just remember the Y2K problem to get a good idea of the kinds of problems your programs may create if they don't calculate date and time values correctly. Fortunately, you don't have to deal with the complexities of date and time calculations, the HLA Standard Library does the hard stuff for you.

The HLA Standard Library date routines produce valid results for dates between January 1, 1583 and December 31, 9999[2]. HLA represents dates using the following record definition (in the *date* namespace):

```
type
    daterec:
        record
            day:uns8;
            month:uns8;
            year:uns16;
        endrecord;
```

This format (*date.daterec*) compactly represents all legal dates using only four bytes. Note that this is the same date format that the chapter on Data Representation presents for the extended data format (see "Bit Fields and Packed Data" on page 81). You should use the *date.daterec* data type when declaring date objects in your HLA programs, e.g.,

```
static
    TodaysDate: date.daterec;
    Century21:  date.daterec := date.daterec:[ 1, 1, 2001 ]; // note: d, m ,y
```

As the second example above demonstrates, the first field is the day field and the second field is the month field if you use a *date.daterec* constant to initialize a static *date.daterec* object. Don't fall into the trap of using the mm/dd/yy or yy/mm/dd organization common in most countries.

1. For those who missed it, the Y2K (or Year 2000) problem occurred when programmers used two digits for the date and assumed that the H.O. two digits were "19". Clearly this code malfunctioned when the year 2000 came along.
2. The Gregorial Calendar came into existence in Oct, 1582, so any dates earlier than this are meaningless as far as date calculations are concerned. The last legal date, 9999, was chosen arbitrarily as a trap for wild dates entering the calculation. This means, of course, that code calling the HLA Standard Library Date/Time package will suffer from the Y10K problem. However, you'll probably not consider this a severe limitation!

The HLA *date.daterec* format has a couple of advantages. First, it is a trivial matter to convert between the internal and external representations of a date. All you have to do is extract the *d*, *m*, and *y* fields and manipulate them as integers of the appropriate sizes. Second, this format makes it very easy to compare two dates to see if one date follows another in time; all you've got to do is compare the *date.daterec* object as though it were a 32-bit unsigned integer and you'll get the correct result. The Standard Library *data.daterec* format does have a few disadvantages. Specifically, certain calculations like computing the number of days between two dates is a bit difficult. Fortunately, the HLA Standard Library Date module provides most of the functions you'll ever need for date calculations, so this won't prove to be much of a disadvantage.

A second disadvantage to the *date.daterec* format is that the resolution is only one day. Some calculations need to maintain the time of day (down to some fraction of a second) as well as the calendar date. The HLA Standard Library also provides a TIME data structure. By combining these two structures together you should be able handle any problem that comes along.

Before going on and discussing the functions available in the HLA Standard Library's Date module, it's probably worthwhile to briefly discuss some other date formats that find common use. Perhaps the most common date format is to use an integer value that specifies the number of days since an *epoch*, or starting, date. The advantage to this scheme is that it's very easy to do certain kinds of date arithmetic (e.g., to compute the number of days between two dates you simply subtract them) and it's also very easy to compare these dates. The disadvantages to this scheme include the fact that it is difficult to convert between the internal representation and an external representation like "xx/yy/zzzz." Another problem with this scheme, which it shares with the HLA scheme, is that the granularity is one day. You cannot represent time with any more precision than one day.

Another popular format combines dates and times into the same value. For example, the representation of time on most UNIX systems measures the number of seconds that have passed since Jan 1, 1970. Unfortunately, many UNIX systems only use a 32-bit signed integer; therefore, those UNIX systems will experience their own "Y2.038K" problem in the year 2038 when these signed integers roll over from 2,147,483,637 seconds to -2,147,483,638 seconds. Although this format does maintain time down to seconds, it does not handle fractions of a second very well. Most UNIX system include an extra field in their date/time format to handle milliseconds, but this extra field is a kludge. One could just as easily add a time field to an existing date format if you're willing to kludge.

For those who want to be able to accurately measure dates and times, a good solution is to use a 64-bit unsigned integer to count the number of microseconds since some epoch data. A 64-bit unsigned integer will provide microsecond accuracy for a little better than 278,000 years. Probably sufficient for most needs. If you need better than microsecond accuracy, you can get nanosecond accuracy that is good for about 275 years (beyond the epoch date) with a 64-bit integer. Of course, if you want to use such a date/time format, you will have to write the routines that manipulate such dates yourself; the HLA Standard Library's Date/Time module doesn't use that format.

6.3 A Brief History of the Calendar

Man has been interested in keeping track of time since the time man became interested in keeping track of history. To understand why we need to perform various calculations, you'll need to know a little bit about the history of the calendar. So this section will digress a bit from computers and discuss that history.

What exactly is time? Time is a concept that we are all intuitively familiar with, but try and state a concrete definition that does not define time in terms of itself. Before you run off and grab a dictionary, you should note that many of the definitions of time in a typical dictionary contain a circular reference (that is, they define time in terms of itself). The *American Heritage Dictionary of the English Language* provides the following definition:

> **A nonspatial continuum in which events occur in apparently irreversible succession from the past through the present to the future.**

As horrible as this definition sounds, it is one of the few that doesn't define time by how we measure it or by a sequence of observable events.

Why are we so obsessed with keeping track of time? This question is much more easily answered. We need to keep track of time so we can predict certain future events. Historically, important events the human race has needed to predict include the arrival of spring (for planting), the observance of religious anniversaries (e.g., Christmas, Passover), or the gestation period for livestock (or even humans). Of course, modern life may seem much more complex and tracking time more important, but we track time for the same reasons the human race always has, to predict the future. Today, we predict business meetings, when a department store will open to the public, the start of a college lecture, periods of high traffic on the highways, and the start of our favorite television shows by using time. The better we are able to measure time, the better we will be able to predict when certain types of events will occur (e.g., the start of spring so we can begin planting).

To measure time, we need some predictable, periodic, event. Since ancient times, there have been three celestial events that suit this purpose: the solar *day*, the lunar *month*, and the solar *year*. The solar day (or *tropical* day) consists of one complete rotation of the Earth on its axis. The lunar month consists of one complete set of moon phases. The solar year is one complete orbit of the Earth around the Sun. Since these periodic events are easy to measure (crudely, at least), they have become the primary basis by which we measure time.

Since these three celestial events were obvious even in prehistoric times, it should come as no surprise that one society would base their measurement of time on one cyclic standard such as the lunar month while another social group would base their time unit on a different cycle such as the solar year. Clearly, such fundamentally different time keeping schemes would complicate business transactions between the two societies effectively erecting an artificial barrier between them. Nevertheless, until about the year 46 BC (by our modern calendar), most countries used their own system for time keeping.

One major problem with reconciling the different calendars is that the celestial cycles are not integral. That is, there are not an even number of solar days in a lunar month, there are not an integral number of solar days in a solar year, and there are not an integral number of lunar months in a solar year. Indeed, there are approximately 365.2422 days in a solar year and approximately 29.5 days in a lunar month. Twelve lunar months are 354 days, a little over a week short of a full year. Therefore, it is very difficult to reconcile these three periodic events if you want to use two of them or all three of them in your calendar.

In 46 BC (or BCE, for *Before Common Era*, as it is more modernly written) Julius Caesar introduced the calendar upon which our modern calendar is based. He decreed that each year would be exactly $365\ ^1/_4$ days long by having three successive years having 365 days each and every fourth year having 366 days. He also abolished reliance upon the lunar cycle from the calendar. However, $365\ ^1/_4$ is just a little bit more than 365.2422, so Julius Caesar's calendar lost a day every 128 years or so.

Around 700 AD (or CE, for *Common Era*, as it is more modernly written) it was common to use the birth of Jesus Christ as the *Epoch* year. Unfortunately, the equinox kept losing a full day every 128 years and by the year 1500 the equinoxes occurred on March 12th, and September 12th. This was of increasing concern to the Church since it was using the Calendar to predict Easter, the most important Christian holiday[3]. In 1582 CE, Pope Gregory XIII dropped ten days from the Calendar so that the equinoxes would fall on March 21st and September 21st, as before, and as advised by Christoph Clavius, he dropped three leap years every 400 years. From that point forward, century years were leap years only if divisible by 400. Hence 1700, 1800, 1900 are *not* leap years, but 2000 *is* a leap year. This new calendar is known as the Gregorian Calendar (named after Pope Gregory XIII) and with the exception of the change from BC/AD to BCE/CE is, essentially, the calendar in common use today[4].

The Gregorian Calendar wasn't accepted universally until well into the twentieth century. Largely Roman Catholic countries (e.g., Spain and France) adopted the Gregorian Calendar the same year as Rome. Other countries followed later. For example, portions of Germany did not adopt the Gregorian Calendar until the year 1700 AD while England held out until 1750. For this reason, many of the American founding fathers have *two* birthdates listed. The first date is the date in force at the time of their birth, the second date

3. Easter is especially important since the Church computed all other holidays relative to Easter. If the date of Easter was off, then all holidays would be off.
4. One can appreciate that non-Christian cultures might be offended at by the abbreviations BC (Before Christ) and AD (Anno Domini [day of our Lord]).

is their birthdate using the Gregorian Calendar. For example, George Washington was actually born on February 11th by the English Calendar, but after England adopted the Gregorian Calendar, this date changed to February 22nd. Note that George Washington's birthday didn't actually change, only the calendar used to measure dates at the time changed.

The Gregorian Calendar still isn't correct, though the error is very small. After approximately 3323 years it will be off by a day. Although there has been some proposals thrown around to adjust for this in the year 4000, that is such a long time off that it's hardly worth contemporary concern (with any luck, mankind will be a spacefaring race by then and the concept of a year, month, or day, may be a quaint anachronism).

There is one final problem with the calendar- the length of the solar day is constantly changing. Ocean tidal forces, meteors burning up in our atmosphere, and other effects are slowing down the Earth's rotation resulting in longer days. The effect is small, but compared to the length of a day, but it amounts to a loss of one to three milliseconds (that is, about 1/500th of a second) every 100 years since the defining Epoch (Jan 1, 1900). That means that Jan 1, 2000 is about two seconds longer than Jan 1, 1900. Since there are 86,400 seconds in a day, it will probably take on the order of 100,000 years before we lose a day due to the Earth's rotation slowing down. However, those who want to measure especially small time intervals have a problem: hours and seconds have been defined as submultiples of a single day. If the length of a day is constantly changing, that means that the definition of a second is constantly changing as well. In other words, two very precise measurements of equivalent events taken 10 years apart may show measurable differences.

To solve this problem scientists have developed the *Cesium-155 Atomic Clock*, the most accurate timing device ever invented. The Cesium atom, under special conditions, vibrates at exactly 9,192,631,770 cycles per second, for the year 1900. Because the clock is so accurate, it has to be adjusted periodically (about every 500 days, currently) so that its time (known as Universal Coordinated Time or UTC) matches that of the Earth (UT1). A high-quality Cesium Clock (like the one at the National Institute of Standards and Technology in Boulder, Colorado, USA) is very large (about the size of a large truck) and can keep accurate time to about one second in a million and a half years. Commercial units (about the size of a large suitcase) are available and they keep time accurate to about one second every 5-10,000 years.

The wall calendar you purchase each year is a device that is very similar to the Cesium Atomic Clock- it lets you measure time. The Cesium clock, clearly, lets time two discrete events that are very close to one another, but either device will probably let you predict that you start two week's vacation in Mexico starting next Monday (and the wall calendar does it for a whole lot less money). Most people don't think of a calendar as a time keeping device, but the only difference between it and a watch is the *granularity*, that is, the finest amount of time one can measure with the device. With a typical electronic watch, you can probably measure (accurately) to as little as $^1/_{100}$ seconds. With a calendar, the minimum interval you can measure is one day. While the watch is appropriate for measuring the 100 meter dash, it is inappropriate for measuring the duration of the Second World War; the calendar, however, is perfect for this latter task.

Time measurement devices, be they a Cesium Clock, a wristwatch, or a Calendar, do not measure time in an absolute sense. Instead, these devices measure time between two events. For the Gregorian Calendar, the (intended) Epoch event that marks year one was the birth of Christ. Unfortunately in 1582, the use of negative numbers was not widespread and even the use of zero was not common. Therefore, 1 AD was (supposed to be) the first year of Christ's life. The year prior to that point was considered 1BC. This unfortunate choice created some mathematical problems that tend to bother people 2,000 years later. For example, the first decade was the first 10 years of Christ's life, that is, 1 AD through 10 AD. Likewise, the first century was considered the first 100 years after Christ's birth, that is, 1 AD through 100 AD. Likewise, the first millennium was the first 1,000 years after Christ's birth, specifically 1 AD through 1000 AD. Similarly, the second millennium is the next 1,000 years, specifically 1001 AD through 2000 AD. The third, millennium, contrary to popular belief, began on January 1, 2001 (Hence the title of Clark's book: "2001: A Space Odyssey"). It is an unfortunately accident of human psychology that people attach special significance to round numbers; there were many people mistakenly celebrating the turn of the millennium on December 31st, 1999 when, in fact, the actual date was still a year away.

Now you're probably wondering what this has to do with computers and the representation of dates in the computer... The reason for taking a close look at the history of the Calendar is so that you don't misuse the date and time representations found in the HLA Standard Library. In particular, note that the HLA date format is based on the Gregorian Calendar. Since the Gregorian Calendar was "born" in October of 1582, it

makes absolutely no sense to represent any date earlier than about Jan 1, 1583 using the HLA date format. Granted, the data type can represent earlier dates numerically, but any date computations would be severely off if one or both of the dates in the computation are pre-1583 (remember, Pope Gregory droped 10 days from the calendar; right off the bat your "days between two dates" computation would be off by 10 real days if the two dates crossed the date that Rome adopted the Gregorian Calendar).

In fact, you should be wary of any dates prior to about January 1, 1800. Prior to this point there were a couple of different (though similar) calendars in use in various countries. Unless you're a historian and have the appropriate tables to convert between these dates, you should not use dates prior to this point in calculations. Fortunately, by the year 1800, most countries that had a calendar based on Juilus Caesar's calendar fell into line and adopted the Gregorian Calendar. Some other calendars (most notably, the Chinese Calendar) were in common use into the middle of the 20th century. However, it is unlikely you would ever confuse a Chinese date with a Gregorian date.

6.4 HLA Date Functions

HLA provides a wide array of date functions you can use to manipulate date objects. The following subsections describe many of these functions and how you use them.

6.4.1 date.IsValid and date.validate

When storing data directly into the fields of a *date.daterec* object, you must be careful to ensure that the resulting date is correct. The HLA date procedures will raise an *ex.InvalidDate* exception if the date values are out of range. The *date.IsValid* and *date.validate* procedures provide some handy code to check the validity of a date object. These two routines use either of the following calling seqeuences:

```
date.IsValid( dateVar ); // dateVar is type date.daterec
date.IsValid( m, d, y ); // m, d, y are uns8, uns8, uns16, respectively

date.validate( dateVar ); // See comments above.
date.validate( m, d, y );
```

The *date.IsValid* procedure checks the date to see if it is a valid date. This procedure returns true or false in the AL register to indicate whether the date is valid. The *date.validate* procedure also checks the validity of the date; however, it raises the *ex.InvalidDate* exception if the date is invalid. The following sample program demonstrates the use of these two routines:

```
program DateTimeDemo;
#include( "stdlib.hhf" );

static
    m:          uns8;
    d:          uns8;
    y:          uns16;

    theDate:    date.daterec;


begin DateTimeDemo;

    try
```



```
            stdout.put( “Enter the month (1-12):” );
            stdin.get( m );

            stdin.flushInput();
            stdout.put( “Enter the day (1-31):” );
            stdin.get( d );

            stdin.flushInput();
            stdout.put( “Enter the year (1583-9999): “ );
            stdin.get( y );

            if( date.isValid( m, d, y )) then

                stdout.put( m, “/”, d, “/”, y, “ is a valid date.” nl );

            endif;

            // Assign the fields to a date variable.

            mov( m, al );
            mov( al, theDate.month );
            mov( d, al );
            mov( al, theDate.day );
            mov( y, ax );
            mov( ax, theDate.year );

            // Force an exception if the date is illegal.

            date.validate( theDate );

          exception( ex.ConversionError )

            stdout.put
            (
                “One of the input values contained illegal characters” nl
            );

          exception( ex.ValueOutOfRange )

            stdout.put
            (
                “One of the input values was too large” nl
            );

          exception( ex.InvalidDate )

            stdout.put
            (
                “The input date (“, m, “/”, d, “/”, y, “) was invalid” nl
            );

        endtry;


    end DateTimeDemo;
```

Program 6.1 Date Validation Example

6.4.2 Checking for Leap Years

Determining whether a given year is a leap year is somewhat complex. The exact algorithm is "any year that is evenly divisible by four and is not evenly divisible by 100 or is evenly divisible by 400 is a leap year[5]." The HLA "datetime.hhf" module provides a convenient function, *date.IsLeapYear*, that efficiently determines whether a given year is a leap year. There are two different ways you can call this function; either of the following will work:

```
date.IsLeapYear( dateVar );    // dateVar is a date.dateRec variable.
date.IsLeapYear( y );          // y is a word value.
```

The following code demonstrates the use of this routine.

```
program DemoIsLeapYear;
#include( "stdlib.hhf" );

static
    m:          uns8;
    d:          uns8;
    y:          uns16;

    theDate:    date.daterec;


begin DemoIsLeapYear;

    try

        stdout.put( "Enter the month (1-12):" );
        stdin.get( m );

        stdin.flushInput();
        stdout.put( "Enter the day (1-31):" );
        stdin.get( d );

        stdin.flushInput();
        stdout.put( "Enter the year (1583-9999): " );
        stdin.get( y );

        // Assign the fields to a date variable.

        mov( m, al );
        mov( al, theDate.month );
        mov( d, al );
        mov( al, theDate.day );
        mov( y, ax );
        mov( ax, theDate.year );

        // Force an exception if the date is illegal.

        date.validate( theDate );
```

5. The Gregorian Calendar does not account for the fact that sometime between the years 3,000 and 4,000 we will have to add an extra leap day to keep the Calendar in sync with the Earth's rotation around the Sun. The HLA date.IsLeapYear does not handle this situation either. Keep this in mind if you are doing date calculations that involve dates after the year 3,000. This is a defect in the current definition of the Gregorian Calendar, which HLA's routines faithfully reproduce.


```
            // Okay, report whether this is a leap year:

            if( date.isLeapYear( theDate )) then

                stdout.put( "The year ", y, " is a leap year." nl );

            else

                stdout.put( "The year ", y, " is not a leap year." nl );

            endif;

            // Technically, the leap day is Feb 25, but most people don't
            // realize this, so use the following output to keep them happy:

            if( date.isLeapYear( y )) then

                if( m = 2 ) then

                    if( d = 29 ) then

                        stdout.put( m, "/", d, "/", y, " is the leap day." nl );

                    endif;

                endif;

            endif;


          exception( ex.ConversionError )

            stdout.put
            (
                "One of the input values contained illegal characters" nl
            );

          exception( ex.ValueOutOfRange )

            stdout.put
            (
                "One of the input values was too large" nl
            );

          exception( ex.InvalidDate )

            stdout.put
            (
                "The input date (", m, "/", d, "/", y, ") was invalid" nl
            );

        endtry;


    end DemoIsLeapYear;
```

Program 6.2 Calling the date.IsLeapYear Function

6.4.3 Obtaining the System Date

The *date.today* function returns the current system date in the *date.daterec* variable you pass as a parameter[6]. The following program demonstrates how to call this routine:

```
program DemoToday;
#include( "stdlib.hhf" );

static
    TodaysDate: date.daterec;


begin DemoToday;


    date.today( TodaysDate );

    stdout.put
    (
        "Today is ",
        (type uns8 TodaysDate.month), "/",
        (type uns8 TodaysDate.day), "/",
        (type uns16 TodaysDate.year),
        nl
    );


    // Okay, report whether this is a leap year:

    if( date.isLeapYear( TodaysDate )) then

        stdout.put( "This is a leap year." nl );

    else

        stdout.put( "This is not a leap year." nl );

    endif;

end DemoToday;
```

Program 6.3 Reading the System Date

Linux users should be aware that *date.today* returns the current date based on Universal Coordinated Time (UTC). Depending upon your time zone, date.today may return yesterday's or tomorrow's date within your particular timezone.

6. This function was not available in the Linux version of the HLA Standard Library as this was written. It may have been added by the time you read this, however.

6.4.4 Date to String Conversions and Date Output

The HLA date module provides a set of routines that will convert a *date.daterec* object to the string representation of that date. HLA provides a mechanism that lets you select from one of several different conversion formats when translating dates to strings. The date package defines an enumerated data type, *date.OutputFormat*, that specifies the different conversion mechanisms. The possible conversions are (these examples assume you are converting the date January 2, 2033):

```
date.mdyy         - Outputs date as 1/2/33.
date.mdyyyy       - Outputs date as 1/2/2033.
date.mmddyy       - Outputs date as 01/02/33.
date.mmddyyyy     - Outputs date as 01/02/2033.
date.yymd         - Outputs date as 33/1/2.
date.yyyymd       - Outputs date as 2033/1/2.
date.yymmdd       - Outputs date as 33/01/02.
date.yyyymmdd     - Outputs date as 2033/01/02.
date.MONdyyyy     - Outputs date as Jan 1, 2033.
date.MONTHdyyyy   - Outputs date as January 1, 2033.
```

To set the conversion format, you must call the *date.SetFormat* procedure and pass one of the above values as the single parameter[7]. For all but the last two formats above, the default month/day/year separator is the slash ("/") character. You can call the *date.SetSeparator* procedure, passing it a single character parameter, to change the separator character.

The *date.toString* and *date.a_toString* procedures convert a date to string data. Like the other string routines this chapter discusses, the difference between the *date.toString* and *date.a_toString* procedures is that *date.a_toString* automatically allocates storage for the string whereas you must supply a string with sufficient storage to the *date.toString* procedure. Note that a string containing 20 characters is sufficient for all the different date formats. The *date.toString* and *date.a_toString* procedures use the following calling sequences:

```
date.toString( m, d, y, s );
date.toString( dateVar, s );

date.a_toString( m, d, y );
date.a_toString( dateVar );
```

Note that *m* and *d* are byte values, *y* is a word value, *dateVar* is a *date.dateRec* value, and *s* is a string variable that must point at a string that holds at least 20 characters.

The *date.Print* procedure uses the *date.toString* function to print a date to the standard output device. This is a convenient function to use to display a date after some date calculation.

The following program demonstrates the use of the procedures this section discusses:

```
program DemoStrConv;
#include( "stdlib.hhf" );

static
    TodaysDate: date.daterec;
    s:          string;


begin DemoStrConv;
```

7. the *date.SetFormat* routine raises the *ex.InvalidDateFormat* exception if the parameter is not one of these values.


```
    date.today( TodaysDate );
    stdout.put( "Today's date is " );
    date.print( TodaysDate );
    stdout.newln();

    // Convert the date using various formats
    // and display the results:

    date.setFormat( date.mdyy );
    date.a_toString( TodaysDate );
    mov( eax, s );
    stdout.put( "Date in mdyy format: '", s, "'" nl );
    strfree( s );

    date.setFormat( date.mmddyy );
    date.a_toString( TodaysDate );
    mov( eax, s );
    stdout.put( "Date in mmddyy format: '", s, "'" nl );
    strfree( s );

    date.setFormat( date.mdyyyy );
    date.a_toString( TodaysDate );
    mov( eax, s );
    stdout.put( "Date in mdyyyy format: '", s, "'" nl );
    strfree( s );

    date.setFormat( date.mmddyyyy );
    date.a_toString( TodaysDate );
    mov( eax, s );
    stdout.put( "Date in mmddyyyy format: '", s, "'" nl );
    strfree( s );


    date.setFormat( date.MONdyyyy );
    date.a_toString( TodaysDate );
    mov( eax, s );
    stdout.put( "Date in MONdyyyy format: '", s, "'" nl );
    strfree( s );


    date.setFormat( date.MONTHdyyyy );
    date.a_toString( TodaysDate );
    mov( eax, s );
    stdout.put( "Date in MONTHdyyyy format: '", s, "'" nl );
    strfree( s );


end DemoStrConv;
```

Program 6.4 Date <-> String Conversion and Date Output Routines

6.4.5 date.unpack and data.pack

The *date.pack* and *date.unpack* functions pack and unpack date data. The calling syntax for these functions is the following:

```
date.pack( y, m, d, dr );
```



```
date.unpack( dr, y, m, d );
```

Note: *y, m, d* must be uns32 or dword variables; *dr* must be a *date.daterec* object.

The *date.pack* function takes the *y, m*, and *d* values and packs them into a *date.daterec* format and stores the result into *dr.* The *date.unpack* function does just the opposite. Neither of these routines check their parameters for proper range. It is the caller's resposibility to ensure that *d's* value is in the range 1..31 (as appropriate for the month and year), *m's* value is in the range 1..12, and *y's* value is in the range 1583..9999.

6.4.6 date.Julian, date.fromJulian

These two functions convert a Gregorian date to and from a Julian day number[8]. Julian day numbers specify January 1, 4713 BCE as day zero and number the days consecutively from that point[9]. One nice thing about Julian day numbers is that date calculations are very easy. You can compute the number of days between two dates by simply subtracting them, you can compute new dates by adding an integer number of days to a Julian day number, etc. The biggest problem with Julian day numbers is converting them to and from the Gregorian Calendar with which we're familiar. Fortunately, these two functions handle that chore. The syntax for calling these two functions is:

```
date.fromJulian( julian, dateRecVar );
date.Julian( m, d, y );
date.Julian( dateRecVar );
```

The first call above converts the Julian day number that you pass in the first parameter to a Gregorian date and stores the result into the *date.daterec* variable you pass as the second parameter. Keep in mind that Julian day numbers that correspond to dates before Jan 1, 1582, will not produce accurate calendar dates since the Gregorian calendar did not exist prior to that point.

The second two calls above compute the Julian day number and return the value in the EAX register. They differ only in the types of parameters they expect. The first call to date.Julian above expects three parameters, *m* and *b* being byte values and *y* being a word value. The second call expects a *date.daterec* parameter; it extracts those three fields and converts them to the Julian day number.

6.4.7 date.datePlusDays, date.datePlusMonths, and date.daysBetween

These two functions provide some simple date arithmetic.operations. The compute a new date by adding some number of days or months to an existing date. The calling syntax for these functions is

```
date.datePlusDays( numDays, dateRecVar );
date.datePlusMonths( numMonths, dateRecVar );
```

Note: *numDays* and *numMonths* are *uns32* values, *dateRecVar* must be a *date.daterec* variable.

The *date.datePlusDays* function computes a new date that is *numDays* days beyond the date that *dateRecVar* specifies. This function leaves the resulting date in *dateRecVar.* This function automatically compensates for the differing number of days in each month as well as the differing number of days in leap years. The *date.datePlusMonths* function does a similar calculation except it adds *numMonths* months, rather than days to *dateRecVar.*

The *date.datePlusDays* function is not particularly efficient if the *numDays* parameter is large. There is a more efficient way to calculate a new date if *numDays* exceeds 1,000: convert the date to a Julian Day Number, add the *numDays* value directly to the Julian Number, and then convert the result back to a date.

8. Note that a Julian date and a Julian day number are not the same thing. Julian dates are based on the Julian Calendar, commisioned by Julius Caesar, which is very similar to the Gregorian Calendar; Julian day numbers were invented in the 1800's and are primarily used by astronomers.
9. Jan 1, 4713 BCE was chosen as a date that predates recorded history.

The *date.daysBetween* function computes the number of days between two dates. Like *date.datePlus-Days*, this function is not particularly efficient if the two dates are more than about three years apart; it is more efficient to compute the Julian day numbers of the two dates and subtract those values. For spans of less than three years, this function is probably more efficient. The calling sequence for this function is the following:

```
date.daysBetween( m1, d1, y1, m2, d2, y2 );
date.daysBetween( m1, d1, y1, dateRecVar2 );
date.daysBetween( dateRecVar1, m2, d2, y2 );
date.daysBetween( dateRecVar1, dateRecVar2 );
```

The four different calls allow you to specify either date as a m/d/y value or as a *date.daterec* value. The *m* and *d* parameters in these calls must be byte values and the *y* parameter must be a word value. The *dateRecVar1* and *dateRecVar2* parameters must, obviously, be *date.daterec* values. These functions return the number of days between the two dates in the EAX register. Note that the dates must be valid, but there is no requirement that the first date be less than the second date.

6.4.8 date.dayNumber, date.daysLeft, and date.dayOfWeek

The *date.dayNumber* function computes the day number into the current year (with Jan 1 being day number one) and returns this value in EAX. This value is always in the range 1..365 (or 1..366 for leap years). A call to this function uses the following syntax:

```
date.dayNumber( m, d, y );
date.dayNumber( dateRecVar );
```

The two forms differ only in the way you pass the date. The first call above expects two byte values (*m* and *d*) and a word value (*y*). The second form above expects a *date.daterec* value.

The *date.daysLeft* function computes the number of days left in a year. This function returns the number of days left in a year *counting the date you pass as a parameter.* Therefore, this function returns one for Dec 31st. Like *date.dayNumber*, this function always returns a value in the range 1..365/366 (regular/leap year). The calling syntax for this function is similar to *date.dayNumber*, it is

```
date.daysLeft( m, d, y );
date.daysLeft( dateRecVar );
```

The parameters have the same meaning as for *date.dayNumber.*

The *date.dayOfWeek* function accepts a date and returns a day of week value in the EAX register. A call to this function uses the following syntax:

```
date.dayOfWeek( m, d, y );
date.dayOfWeek( dateRecVar );
```

The parameters have their usual meanings.

These function calls return a value in the range 0..7 (in EAX) as follows:

0: Sunday

1: Monday

2: Tuesday

3: Wednesday

4: Thursday

5: Friday

6: Saturday

6.5 Times

The HLA Standard Library provides a simple time module that lets you manipulate times in an HHMMSS (hours/minutes/seconds) format. The *time* namespace in the date/time module defines the time data type as follows:

```
type
    timerec:
        record
            secs:uns8;
            mins:uns8;
            hours:uns16;
        endrecord;
```

This format easily handles 60 seconds per minute and 60 minutes per hour. It also handles up to 65,535 hours (just over 2730 days or about 7-$^1/_2$ years).

The advantages to this time format parallel the advantages of the date format: it is easy to convert the time format to/from external representation (i.e., HH:MM:SS) and the storage format lets you compare times by treating them as *uns32* objects. Another advantage to this format is that it supports more than 24 hours, so you can use it to maintain timings for events that are not calendar based (up to seven years).

There are a couple of disadvantages to this format. The primary disadvantage is that the minimum granularity is one second; if you want to work with fractions of a second then you will need to use a different format and you will have to write the functions for that format. Another disadvantage is that time calculations are somewhat inconvenient. It is difficult to add *n* seconds to a time variable.

Before discussing the HLA Standard Library Time functions, a quick discussion of other possible time formats is probably wise. The only reasonable alternative to the HH:MM:SS format that the HLA Standard Library Time module uses is to use an integer value to represent some number of time units. The only question is "what time units do you want to use?" Whatever time format you use, you should be able to represent at least 86,400 seconds (24 hours) with the format. Furthermore, the granularity should be one second or less. This effectively means you will need at least 32 bits since 16 bits only provides for 65,536 seconds at one second granularity (okay, 24 bits would work, but it's much easier to work with four-byte objects than three-byte objects).

With 32-bits, we can easily represent more than 24 hours' worth of milliseconds (in fact, you can represent almost 50 days before the format rolls over). We could represent five days with a $^1/_{10,000}$ second granularity, but this is not a common timing to use (most people want microseconds if they need better than millisecond granularity), so millisecond granularity is probably the best choice for a 32-bit format. If you need better than millisecond granularity, you should use a combined date/time 64-bit format that measures microseconds since Julian Day Number zero (Jan 1, 4713 BCE). That's good for about a half million years. If you need finer granularity than microseconds, well, you're own your own! You'll have to carefully weigh the issues of granularity vs. years covered vs. the size of your data.

6.5.1 time.curTime

This function returns the current time as read from the system's time of day clock. The calling syntax for this function is the following:

```
time.curTime( timeRecVar );
```

This function call stores the current system time in the *time.timerec* variable you pass as a parameter. On Windows systems, the current time is the wall clock time for your particular time zone; under Linux, the current time is always given in UTC (Universal Coordinated Time) and you must adjust according to your particular time zone to get the local time. Keep this difference in mind when porting programs between Windows and Linux.

6.5.2 time.hmsToSecs and time.secstoHMS

These two functions convert between the HLA internal time format and a pure seconds format. Generally, when doing time arithmetic (e.g., time plus seconds, minutes, or hours), it's easiest to convert your times to seconds, do the calculations with seconds, and then translate the time back to the HLA internal format. This lets you avoid the headaches of modulo-60 arithmetic.

The calling sequences for the *time.hmsToSecs* function are

```
time.hmsToSecs( timeRecValue );
time.hmsToSecs( h, m, s );
```

Both functions return the number of seconds in the EAX register. They differ only in the type of parameters they expect. The first form above expects an HLA *time.timerec* value. The second call above lets you directly specify the hours, minutes, and seconds as separate parameters. The *h* parameter must be a word value, the *m* and *s* parameters must be byte values.

The *time.secsToHMS* function uses the following calling sequence:

```
time.secsToHMS( seconds, timeRecVar );
```

The first parameter must be an uns32 value specifying some number of seconds less than 235,939,600 seconds (which corresponds to 65,536 hours). The second parameter in this call must be a *time.timerec* variable. This function converts the seconds parameter to the HLA internal time format and stores the value into the *timeRecVar* variable.

6.5.3 Time Input/Output

The HLA Standard Library doesn't provide any specific I/O routines for time data. However, reading and writing time data in ASCII form is a fairly trivial process. This section will provide some examples of time I/O using the HLA Standard Input and Standard Output modules.

To output time in a standard HH:MM:SS format, just use the stdout.putisize routines with a width value of two and a fill character of '0' for the three fields of the HLA *time.timerec* data type. The following code demonstrates this:

```
static
    t:time.timerec;
        .
        .
        .
    stdout.putisize( t.h, 2, '0' );
    stdout.put( ':' );
    stdout.putisize( t.m, 2, '0' );
    stdout.put( ':' );
    stdout.putisize( t.s, 2, '0' );
```

If this seems like too much typing, well fear not; in a later chapter you will learn how to create your own functions and you can put this code into a function that will print the time with a single function call.

Time input is only a little more complicated. As it turns out, HLA accepts the colon (":") character as a delimiter when reading numbers from the user. Therefore, reading a time value is no more difficult than reading any other three integer values; you can do it with a single call like the following:

```
stdin.get( t.hours, t.mins, t.secs );
```

There is one remaining problem with the time input code: it does not validate the input. To do this, you must manually check the seconds and minutes fields to ensure they are values in the range 0..59. If you wish to enforce a limit on the hours field, you should check that value as well. The following code offers one possible solution:


```
stdin.get( t.hours, t.mins, t.secs );
if( t.m >= 60 ) then

    raise( ex.ValueOutOfRange );

endif;
if( t.s >= 60 ) then

    raise( ex.ValueOutOfRange );

endif;
```

6.6 Putting It All Together

Date and time data types do not get anywhere near the consideration they deserve in modern programs. To help ensure that you calculate dates properly in your HLA programs, the HLA Standard Library provides a set of date and time functions that ease the use of dates and times in your programs.

