11 Date Functions (datetime.hhf)

HLA contains a set of procedures and functions that simplify *correct* date calculations. There are actually two modules: a traditional set of date functions and, for those who prefer an object-oriented approach, a date class module.

Note: be sure to read the chapter on "Passing Parameters to Standard Library Routines" (parmpassing.rtf) before reading this chapter.

A Note About Thread Safety: The date and time routines maintain a couple of static global variables that track the output format and output separate characters for dates. Currently, these values apply to all threads in a process. You should take care when changing these values in threads. When the process module is added to the standard library, these values will be placed in a per-thread data structure. Until then, you should set the format/separator character before starting any other threads and avoid changing their values once other threads (that might use the date/time library module) begin execution.

Note about stack diagrams: this documentation includes stack diagrams for those functions that pass parameters on the stack. To conserve space, this documentation does not include a stack diagram for any function that does not pass data on the stack (that is, only a return address appears on the stack).

Note about function overloading: the functions in the date/time module use function overloading in order to allow you to specify the parameter lists in different ways. The macro that handles the overloading generally coerces the possible parameter types into a single object that it passes to the underlying function. The documentation for the specific functions will tell you whether a symbol is a macro or a function. For the most part, this should matter to you unless you are taking the address of a function (which you cannot do with a macro). See the HLA documentation for more details on function overloading via macros.

11.1 The Date Module

To use the date functions in your application, you will need to include one of the following statements at the beginning of your HLA application:

```
#include( "datetime.hhf" )
or
#include( "stdlib.hhf" )
```

11.2 Date Data Types

The date namespace defines the following useful data types:

`date.daterec`

Date representation. This is a dword object containing m, d, and y fields holding the obvious values. The y field is a 16-bit quantity supporting years 0..9,999. No Y2K problems here! (Of course, it does suffer from Y10K, but that's probably okay.) Since the Gregorian calendar began use in Oct, 1582, there is really no need to represent dates any earlier than this. In fact, most date calculations in the HLA stdlib will not allow dates earlier than Jan 1, 1600 for this very reason. The limitation of year 9999 is an arbitrary limit set in the library to help catch wild values. If you really need dates beyond 9999, feel free to modify the date validation code. The m and d fields are both byte objects. The date validation routines enforce the month limits of 1..12 and appropriate day limits (depending on the month and year).

Here is the current data type definition for the *daterec* data type:

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

Because of the way the fields are defined, you may compare two dates as 32-bit values and test the result using unsigned conditional branch instructions.

date.outputFormat

This is an enumerated data type that defines the following constants:

mdyy, mdyyyy, mmddyy, mmddyyyy, yymd, yyyymd, yymmdd, yyyymmdd, MONdyyyy, and MONTHdyyyy. These constants control the date output format in the (mostly) obvious way. Note that mdyy can output one digit for the day and month while mmddyy always inputs two digits for each field. The MONdyyyy format outputs dates in the form "Jan 1, 2000" while the MONTHdyyyy outputs the dates using the format "January 1, 2000".

```
type
    OutputFormat:
                    enum
                    {

                        mdyy,
                        mdyyyy,

                        mmddyy,
                        mmddyyyy,

                        yymd,
                        yyyymd,
                        yymmdd,
                        yyyymmdd,

                        MONdyyyy,
                        MONTHdyyyy,
                        badDateFormat
                    };
```

11.3 Date Tables

The date/time module includes several date/time-related data objects that may be of interest to an application programmer. Here are the declarations found in the datetime.hhf header file:

```
DaysToMonth     :uns32[13];
DaysInMonth     :uns32[13];
DaysFromMonth   :uns32[13];
Months          :string[13];
shortMonths     :string[13];
```

You must treat these tables as read-only objects. Changing their values will cause the date/time routines to produce incorrect results. Each of these tables is indexed by a month value in the range 1..12. Zero is an illegal value and the value found at index 0 in these tables is undefined. Obviously, accessing any data beyond index 12 is also undefined. The first three functions return some number of days relative to the month whose index you've supplied. These day values are relative to the first day of the specified month. The values in these tables are for non-leap years. If your date calculation is for a leap year, you must add one to the value found in these tables, as appropriate for the month you specify; details appear in the discussion of each function.

DaysToMonth contains the number of days from January 1 to the first of the month you specify as the index. For example, index 1 contains zero, index 2 contains 31, index 3 contains 59 (31+28), etc. For leap years, you will need to add one to the table entry if the index is in the range 3..12.

DaysInMonth contains the number of days in the month specified by the index. For example, DaysInMonth[1] contains 31, DaysInMonth[2] contains 28, and DaysInMonth[3] contains 31. For leap years, you need to add one to the value appearing at index 2 (of course, it's probably just easier to explicitly set the value to 29 for February in leap years).

DaysFromMonth contains the number of days from the first day of the month specified by the index to the first day of January in the following year. For example, DaysFromMonth[1] will contain 365, DaysFromMonth[2] will contain 334 (365-31), and so on. For leap years, you will want to add one to the value if the month index is less than 3.

Months is an array of strings, indexed by the month value, that contain the month's name. For example, Month[1] is the string "January" and Month[2] is the string "February".

shortMonths is an array of strings that contain shortened versions of the month names (the first three characters of each of the month names found in the Months array).

11.4 Date Predicates

The date module provides many functions that test date values. This section details those functions.

```
#macro date.isLeapYear( y:uns32 ); @returns( "al" );
#macro date.isLeapYear( dr:date.daterec ); @returns( "al" );
procedure date._isLeapYear( Year:word ); @returns( "al" );
```

This is an overloaded function. You may either pass it an unsigned integer containing the year or a date.daterec value specifying a m/d/y value (the overloading function will simply pick out the year value and pass it on to the underlying date._isLeapYear function). These functions return true or false in the AL register depending upon whether the parameter is a leap year (true if it is a leap year). Note that this function will be correct until sometime between the years 3000 and 4000, at which point people will probably have to agree upon adding an extra leap day in at some point (no such agreement has been made today, hence the absence from this function); currently, HLA date routines do not allow dates beyond the year 2999, so this won't be a problem unless you modify the maximum year value in the date/time header files. Note that these functions return the boolean result zero-extended into EAX. The @returns("al") declarations exist so that these functions will be type-compatible with boolean objects.

```
HLA high-level calling sequence examples:


date.isLeapYear( someDateVar );
mov( al, someDateVar_is_leap_year );
if( date.isLeapYear( aYearValue )) then

  // Do something if aYearValue is a leap year

  endif;

mov( &date._isLeapYear( ebx ), ptrToIsLeapYearFunction );


HLA low-level calling sequence examples:

  movzx( someDateVar.year, eax );
  push( eax );
  call date._isLeapYear;
  mov( al, someDateVar_is_leap_year );

  movzx( aYearValue, eax );
  push( eax );
  call date._isLeapYear;
  test( al, al );
  jz notALeapYear;

  // Do something if aYearValue is a leap year

  notALeapYear:
```

```
#macro date.validate( m:byte; day:byte; year:word );
#macro date.validate( dr:date.daterec );
date._validate( dr:daterec );
```

These two functions check the date passed as a parameter and raise an ex.InvalidDate exception if the data in the fields (or the m/d/y) parameter is not a valid date between 1/1/1600 and 12/31/2999.

```
HLA high-level calling sequence examples:


try

  date.validate( someDateVar );

  anyexception

  // Do something if the date is invalid

endtry;

try

  date.validate( someMonth, someDay, someYear );

  anyexception

  // Do something if the date is invalid

endtry;

try

  date._validate( someDateVar );

  anyexception

  // Do something if the date is invalid

endtry;


HLA low-level calling sequence examples:

push( someDateVar.date );
call date._validate;
```

```
#macro date.isValid( m:byte; day:byte; year:word ); @returns( "al" );
#macro date.isValid( dr:date.daterec ); @returns( "al" );
date._isValid( dr:daterec ); @returns( "al" );
```

Similar to the date.validate procedures, except these functions return true/false in the AL register if the date is valid/invalid. They do not raise an exception. Note that these functions return the boolean result zero-extended into EAX. The @returns("al") declarations exist so that these functions will be type-compatible with boolean objects.

11.5 Date Conversions

The functions (and macros) in this category convert dates from one format to another. Specifically, there are conversions to and from Julian day numbers, conversions to days into year, computation of week day (Sunday through Saturday), and conversions to days left in year.

#macro date.pack(m, d, y, dr);

date.pack is a macro that accepts year (*y*), month (*m*), and day (*d*) values (presumably dwords), and a *date.daterec* (dr) object. It converts the three values to *date.daterec* form and stores the result in the specified destination. If the *y*, *m*, or *d* values are constants, this macro checks them to ensure they are somewhat reasonable (days are only checked for the range 1..31, years are checked for the range 1600..2999). If *m* or *d* are memory objects, then they are coerced to a byte before use. If *y* is a memory object, it is coerced to a word before use. You may use registers for *y*, *m*, and *d*; if you do, the *m* and *d* values must be passed in 8-bit registers and the *y* value must be passed in a 16-bit register. This macro works best if all three operands are constants. If you take a look at the macro definition in the datetime.hhf header file, you'll discover that this macro efficiently translates a constant date into a single machine instruction. The macro attempts to generate good code for other operand types, but if efficiency is your primary concern, you may want to consider manually moving the data into the fields of the *daterec* object if the *d*, *m*, and *y* values are memory operands.

Because this is a macro, there are no parameters passed on the stack (hence, no stack diagram). Do note, however, that this macro preserves the value in EAX on the stack if it needs to use EAX. As such you should not specify an ESP-relative memory operand as one of the parameters to this macro. In some cases the macro will push EAX on the stack during conversion, in other cases it will not. As such, ESP-relative memory addresses may be rendered incorrect when this macro preserves EAX on the stack.

Other than a mild check for constant operands, this macro does not validate the date you pack into the *dr* argument. No checking is done because this macro is primarily intended for moving constant values into a *daterec* object (and you should be able to verify the value manually when writing the macro invocation). If you need to verify the date packed into the *dr* parameter, use the *date.validate* or *date.isValid* functions.

```
Macro invocation example:

date.pack( 6, 21, 2007, drDateVar );
```

#macro date.unpack(dr, m, d, y);

date.unpack is a macro that accepts a *daterec* object (*dr*) and converts this to three dword values (*m*, *d*, and *y*) by zero-extending the values before storing them into the destination locations. The *m*, *d*, and *y* operands must be 32-bit memory locations or registers (except EAX, which this macro uses for the zero extension operation).

```
Macro invocation example:

date.unpack( drDateVar, monthVar32, dayVar32, yearVar32 );
```

#macro date.toJulian(m:byte; d:byte; y:word); @returns("eax");
#macro date.toJulian(dr:date.daterec); @returns("eax");
date._toJulian(dr:daterec); @returns("eax");

These functions convert the Gregorian (i.e., standard) date passed as a parameter into a "Julian day number." A Julian day number treats Jan 1, 4713 as day zero and simply numbers dates forward from that point. For example, Oct 9, 1995 is JD 2,450,000. Jan 1, 2000 is JD 2,452,545. Julian dates make date calculations of the form "what date is it X days from now?" trivial. You just compute JD+X to get the new date.

A Julian Day begins at 12:00 noon (compared with Gregorian days, that begin at 12:00 midnight). Because these functions do not have a time parameter, they assume that the time is between 12:00 noon on the specified date you pass as a parameter and 11:59:59 of the next day. If the current time is before 12:00 noon, you should subtract one from the Julian day number these functions return. If you would like a true 'toJulian' function, you can easily write one thusly:

```
procedure toJulian( dr:date.daterec; tm:time.timerec );
begin toJulian;
```

```
    date.toJulian( dr );
    if( tm.hour < 12 ) then
       dec( eax );
    endif;

      end toJulian;
```

date.toJulian is actual a macro that handles the parameter overloading. It reformats the parameters (as necessary) and calls the *date._toJulian* function to do the actual work. You would not normally call the *date._toJulian* function as the *date.toJulian* macro with a single argument makes this call for you. You would normally use the *date._toJulian* function in your applications if you need to pass the address of a function as a parameter to some other function (you cannot take the address of a macro).

Note: a "Julian Date" is not the same thing as a "Julian Day Number". A Julian date is based on the Julian Calendar created by Julius Caesar in about 45 BC. It was very similar to our current calendar except they didn't get the leap years quite right. Julian Day numbers are a different calendar system that, as explained above, number days consecutively after Jan 1 4713 BC (resetting to day one 7980 years later). Out of sheer laziness, this document will use the term "Julian Date" as a description of the calendar based on Julian day numbers despite that fact that this is technically incorrect.

```
HLA high-level calling sequence examples:

date.toJulian( someDateVar );
mov( eax, JulianDayNumber );
date.toJulian( month, day, year );
mov( eax, JulianDayNumber2 );


HLA low-level calling sequence examples:

push( someDateVar.date );
call date._toJulian;
mov( eax, JulianDayNumber );
```

date.fromJulian(jd:uns32; var gd:date.daterec);

This procedure converts a Julian date to a Gregorian date. The Julian date is the first parameter, the second (reference) parameter is a Gregorian date variable (*data.daterec*). See the note above about adding some number of days to a Gregorian date via translation to Julian. Note that adding months or years to a Julian date is a real pain in the rear. It's generally easier and faster to convert the Julian date to a Gregorian date, add the months and/or years to the Gregorian date (which is relatively easy), and then convert the whole thing back to a Julian day number.

```
HLA high-level calling sequence examples:

date.fromJulian( JulianDayNumber, someDateVar );


HLA low-level calling sequence examples:

  // Assume someDateVar is a static variable:

  push( JulianDayNumber );
pushd( &someDateVar  );
call date.fromJulian;

// Assume someDateVar is not a static object

push( JulianDayNumber );
lea( eax, someDateVar );
```

```
    push( eax );
    call date.fromJulian;
```

```
#macro date.dayNumber( m:byte; d:byte; y:word ); @returns( "eax" );
#macro date.dayNumber( dr:date.daterec ); @returns( "eax" );
date._dayNumber( dr:daterec ); @returns( "eax" );
```

These functions convert the Gregorian date passed as a parameter into a day number into the current year (often erroneously called a "Julian Date" since NASA adopted this terminology in the late sixties). These functions return a value between 1 and 365 or 366 (for leap years) in the EAX register. Jan 1 is day 1, Dec 31 is day 365 or day 366.

```
    HLA high-level calling sequence examples:

    date.dayNumber( someDateVar );
    mov( eax, dayNumber1 );
    date.dayNumber( month, day, year );
    mov( eax, dayNumber2 );


    HLA low-level calling sequence examples:

    push( someDateVar.date  );
    call date._dayNumber;
    mov( eax, dayNumber1 );
```

```
#macro date.daysLeft( m:byte; d:byte; y:word ); @returns( "eax" );
#macro date.daysLeft( dr:date.daterec ); @returns( "eax" );
date._daysLeft( dr:daterec ); @returns( "eax" );
```

These functions return the number of days left in the current year *counting the date passed as a parameter* (hence Dec 31, *yyyy* always returns one).

```
    HLA high-level calling sequence examples:

    date.daysLeft( someDateVar );
    mov( eax, daysLeft1 );
    date.daysLeft( month, day, year );
    mov( eax, daysLeft2 );


    HLA low-level calling sequence examples:

    push( someDateVar.date  );
    call date._daysLeft;
    mov( eax, daysLeft1 );
```

```
#macro date.dayOfWeek( m:byte; d:byte; y:word ); @returns( "eax" );
#macro date.dayOfWeek( dr:date.daterec ); @returns( "eax" );
date._dayOfWeek( dr:daterec ); @returns( "eax" );
```

These functions return, in EAX, a value between zero and six denoting the day of the week of the given Gregorian date (0=sun, 1=mon, etc.)

```
    HLA high-level calling sequence examples:

    date.dayOfWeek( someDateVar );
```

```
    mov( eax, dayOfWeek1 );
    date.dayOfWeek( month, day, year );
    mov( eax, dayOfWeek2 );


    HLA low-level calling sequence examples:

    push( someDateVar.date  );
    call date._dayOfWeek;
    mov( eax, dayOfWeek1 );
```

11.6 Date Arithmetic

The functions in this category perform date arithmetic – adding integer (day) values to dates, subtracting integer (day) values, adding integer month or year values to a date, and computing the number of days between two dates.

```
#macro date.daysBetween
(
     m1:byte;
     d1:byte;
     y1:word;
     m2:byte;
     d2:byte;
     y2:word
); @returns( "eax" );


#macro date.daysBetween
(
     m1:byte;
     d1:byte;
     y1:word;
     dr:date.daterec
); @returns( "eax" );


#macro date.daysBetween
(
     dr:date.daterec;
     m:byte;
     d:byte;
     y:word
); @returns( "eax" );


#macro date.daysBetween
(
     dr1:date.daterec;
     dr2:date.daterec
); @returns( "eax" );


date._daysBetween( first:daterec; last:daterec ); @returns( "eax" );
```

These functions return an uns32 value in EAX that gives the number of days between the two specified dates. These functions work directly on the Gregorian dates.

```
HLA high-level calling sequence examples:

date.daysBetween( dateVar1, dateVar2 );
mov( eax, daysBetweenD1D2 );

date.daysBetween( m1, d1, y1, m2, d2, y2 );
mov( eax, daysBetween2 );

date.daysBetween( dateVar3, m4, d4, y4 );
mov( eax, daysBetween3 );

date.daysBetween( m5, d5, y5, dateVar6, );
mov( eax, daysBetween4 );


HLA low-level calling sequence examples:

push( dateVar1.date  );
push( dateVar2.date  );
call date._daysBetween;
mov( eax, daysBetween5 );
```

date.addDays(days:uns32; var dr:date.daterec);

This procedure adds the first parameter (in days) directly to the Gregorian date variable passed by reference as the second parameter.

```
HLA high-level calling sequence example:

date.addDays( days, someDateVar );

HLA low-level calling sequence example:

push( days );
push( someDateVar.date  );
call date.addDays;
```

date.addMonths(months:uns32; var dr:date.daterec);

This procedure adds the first parameter (in months) directly to the Gregorian date variable passed by reference as the second parameter.

```
HLA high-level calling sequence example:

date.addMonths( months, someDateVar );

HLA low-level calling sequence example:

push( months);
push( someDateVar.date  );
call date.addMonths;
```

date.addYears(years:uns32; var dr:date.daterec);

This procedure adds the first parameter (in years) directly to the Gregorian date variable passed by reference as the second parameter.

```
HLA high-level calling sequence example:

date.addYears( years, someDateVar );

HLA low-level calling sequence example:

push( years);
push( someDateVar.date  );
call date.addYears;
```

date.subDays(days:uns32; var dr:date.daterec);

This procedure subtracts the first parameter (in days) directly from the Gregorian date variable passed by reference as the second parameter.

```
HLA high-level calling sequence example:

date.subDays( days, someDateVar );

HLA low-level calling sequence example:

push( days );
push( someDateVar.date  );
call date.subDays;
```

date.subMonths(months:uns32; var dr:date.daterec);

This procedure subtracts the first parameter (in months) directly from the Gregorian date variable passed by reference as the second parameter.

```
HLA high-level calling sequence example:

date.subMonths( months, someDateVar );

HLA low-level calling sequence example:

push( months);
push( someDateVar.date  );
call date.subMonths;
```

date.subYears(years:uns32; var dr:date.daterec);

This procedure subtracts the first parameter (in years) directly from the Gregorian date variable passed by reference as the second parameter.

```
HLA high-level calling sequence example:

date.subYears( years, someDateVar );

HLA low-level calling sequence example:
```

```
push( years);
push( someDateVar.date  );
call date.subYears;
```

11.7 Reading the Current System Date

The functions in this category read the current date from the system.

`date.today( var dr:date.daterec );`

Stores the local date (today's date) into the specified parameter. Warning: some systems may not provide a localized date and time, if this is the case then this function will return the UTC/GMT date. If this would cause your application to fail, then you should read both the local and UTC dates and times and, if they are not different, apply an application-defined time zone difference to the local date value.

```
HLA high-level calling sequence example:

  date.today( someDateVar );

HLA low-level calling sequence example:

  // Assume that "someDateVar" is a static object:

  pushd( &someDateVar  );
  call date.today;

  // If someDateVar is not a static object:

  lea( eax, someDateVar );
  push( eax );
  call date.today;
```

`date.utc( var dr:date.daterec );`

Stores the UTC date (today's GMT date) into the specified parameter. Of course, the difference between the local and GMT date depend entirely upon which time zone you're in.

```
HLA high-level calling sequence example:

  date.utc( someDateVar );

HLA low-level calling sequence example:

  // Assume that "someDateVar" is a static object:

  pushd( &someDateVar  );
  call date.utc;

  // If someDateVar is not a static object:

  lea( eax, someDateVar );
  push( eax );
  call date.utc;
```

11.8 Date Output and String Conversion

The date module contains several functions that let you choose a date output format, convert a date to a string, and output dates (to the standard output device). This section describes those functions.

`date.setFormat( fmt : OutputFormat );`

This sets the internal format variable to the *date.OutputFormat* value you specify. This constant must be one of the *date.OutputFormat* enumerated constants (given earlier) or *date.SetFormat* will raise an *ex.InvalidDateFormat* exception.

```
HLA high-level calling sequence example:

  date.setFormat( date.mmddyyyy );
  date.setFormat( dateFmtVariable );

HLA low-level calling sequence example:

  // If the parameter is a constant:

  pushd( date.mmddyyyy  );
  call date.setFormat;

  // If someFmtVar is byte variable and the
  // three bytes following it are in paged memory:

  push( (type dword someFmtVar) );
  call date.setFormat;

  // If you cannot access the three bytes beyond someFmtVar:

  movzx( someFmtVar, eax );
  push( eax );
  call date.setFormat;
```

`date.setSeparator( chr:char );`

This procedure sets the internal date separator character (default is '/') to the character you pass as a parameter. This is used when printing dates and converting dates to strings.

```
HLA high-level calling sequence example:

  date.setSeparator( '-' );
  date.setSeparator( someCharVar );

HLA low-level calling sequence example:

  // If the parameter is a constant:

  pushd( '/'  );
  call date.setSeparator;

  // If someFmtVar is byte variable and the
  // three bytes following it are in paged memory:

  push( (type dword someCharVar) );
  call date.setSeparator;

  // If you cannot access the three bytes beyond someFmtVar:
```

```
    movzx( someCharVar, eax );
    push( eax );
    call date.setSeparator;
```

```
#macro date.toString( m:byte; d:byte; y:word; s:string );
#macro date.toString( dr:date.daterec; s:string);
date._toString( dr:daterec; s:string );
```

These functions will convert the specified date to a string (using the output format specified by *date.SetFormat* and the separator character specified by *date.SetSeparator*) and store the result in the specified string. An *ex.StringOverflow* exception occurs if the destination string's *MaxStrLen* field is too small (generally, 20 characters handles all string formats).

HLA high-level calling sequence examples:

```
    date.toString( someDateVar, dateString1 );
    date.toString( month, day, year, dateString2 );
```

HLA low-level calling sequence examples:

```
    push( someDateVar.date  );
    push( dateString3 );
    call date._toString;
```

```
#macro date.a_toString( m:byte; d:byte; y:word ); @returns( "eax" );
#macro date.a_toString( dr:date.daterec );  @returns( "eax" );
date._a_toString( dr:daterec ); @returns( "eax" );
```

These procedures are similar to the *date.toString* procedures above except they automatically allocate the storage for the string and return a pointer to the string object in the EAX register. You should free the string storage with *str.free* with you are done with this string.

HLA high-level calling sequence examples:

```
    date.a_toString( someDateVar );
    mov( eax, dateStr1 );
    date.a_toString( month, day, year );
    mov( eax, dateStr2 );
```

HLA low-level calling sequence examples:

```
    push( someDateVar.date  );
    call date._a_toString;
    mov( eax, dateStr3 );
```

11.9 Date Class Types

For those who prefer an object-oriented programming approach, the Standard Library provides the ability to create date class data types. To use one of the date class data types, you must include the following statement at the beginning of your HLA program:

#include("dtClass.hhf")

Note that stdlib.hhf does not include dtClass.hhf, so you must explicitly include the dtClass.hhf header file if you intend to use any of the date class functions or data types.

The Standard Library provides two predefined date class types: *dateClass_t* and *virtualDateClass*. The difference between these two types is that the *dateClass_t* type uses static procedures for all the date functions whereas *virtualDateClass_t* uses virtual methods for all the date functions. In certain cases, using the *dateClass_t* data type is more efficient than using *virtualDateClass_t* because you only link in the class functions you actually call. However, you lose the object-oriented method inheritence/override ability when using the *dateClass_t* type rather than the *virtualDateClass_t* type. For more details on the differences between these two class types, please see the discussion of the *dtClass.make_dateClass* macro appearing later in this section. This section will use the phrase "date class" to mean any class created by the *make.dateClass_t* macro, including the *dateClass_t* and *virtualDateClass_t* data types.

The date class types provide three data fields:

```
var
    theDate:    date.daterec;
    OutFmt:     date.OutputFormat;
    Separator:  char;
```

The first field, *theDate*, holds the date value associated with the date object. This is the standard *date.daterec* date type described earlier in this document. Note that you can pass this field to any of the standard date and time functions that expect a *date.daterec* value.

The second field, *OutFmt*, specifies the output format when using the date class string conversion routines. Note that only the date class string conversion routines respect the value of this field; if you pass *theDate* directly to a date function that takes a *date.daterec* argument, that function will use the system-wide global date format rather than the object's *OutFmt* value.

Thread Safety Issue: Although each date object has its own *OutFmt* field, this does not make the use of date class objects thread safe. When converting theDate to a string, the date class functions save the global format value, copy OutFmt to the global format value, call the date functions to do the string conversion, and then restore the original global value. If a thread is suspended during this activity then any date/string conversions during this suspension may use an incorrect format value. This issue will be corrected in a later version of the Standard Library. For now, you must manually protect all date/string conversions if you perform such conversions in multiple threads in your application.

The third field, *Separator*, holds the character that is used to separate the months, days, and years fields during a string conversion. The *dateClass_t* and *virtualDateClass_t* constructors initialize this field with a slash character ('/').

Of course, you may create a derived class from either *dateClass_t* or *virtualDateClass_t* (or create a brand new date class using the *dtClass.make_dateClass* macro) and add any other fields you like to that new date class. One suggestion for such a class is to pad the data fields to a multiple of four bytes. Currently, the *dateClass_t* and *virtualDateClass_t* objects consumes ten bytes of storage (six bytes for the three fields above plus four bytes for the VMT pointer). For performance reasons, you might want to extend the size of the data storage to 12 or even 16 bytes.

11.9.1 Date Class Methods/Procedures

In most HLA classes, there are two types of functions: (static) procedures and (dynamic) methods (there are also iterators, but the date classes do not use iterators so we will ignore that here). The only difference between a method and a procedure is how the program actually calls the function: the program calls procedures directly, it calls methods indirectly through an entry in the virtual method table (VMT). Static procedure calls are very efficient, but you lose the benefits of inheritence and functional polymorphism when you define a function as a static procedure in a class. Methods, on the other hand, fully support polymorphic calls, but introduce some efficiency issues.

First of all, unlike static procedures, your program will link in all methods defined in your program *even if you don't explicitly call those methods in your program*. Because the call is indirect, there really is no way for the assembler and linker to determine whether you've actually called the function, so it must assume that you do call it and links in the code for each method in the class. This can make your program a little larger because it may be including several date class functions that you don't actually call.

The second effiency issue concerning method calls is that they use the EDI register to make the indirect call (static procedure calls do not disturb the value in EDI). Therefore, you must ensure that EDI is free and available before making a virtual method call, or take the effort to preserve EDI's value across such a call.

A third, though exteremely minor, efficiency issue concerning methods is that the class' VMT will need an extra entry in the virtual method table. As this is only four bytes per class (not per object), this probably isn't much of a concern.

The predefined *dateClass_t* and *virtualDateClass_t* classes differ in how they define the functions appearing in the class types. The *dateClass_t* type uses static procedures for all functions, the *virtualDateClass_t* type uses methods for all class functions. Therefore, *dateClass_t* date types will make direct calls to all the functions (and

only link in the procedures you actually call); however, *dateClass_t* objects do not support function polymorphism in derived classes. The *virtualDateClass_t* type does support polymorphism for all the class methods, but whenever you use this data type you will link in all the methods (even if you don't call them all) and calls to these methods will require the use of the EDI register.

It is important to understand that *dateClass_t* and *virtualDateClass_t* are two separate types. Neither is derived from the other. Nor are the two types compatible with one another. You should take care not to confuse objects of these two types if you're using both types in the same program.

11.9.2 Creating New Date Class Types

As it turns out, the only difference between a method and a procedure (in HLA) is how that method/procedure is called. The actual function code is identical regardless of the declaration (the reason HLA supports method and procedure declarations is so that it can determine how to populate the VMT and to determine how to call the function). By pulling some tricks, it's quite possible to call a procedure using the method invocation scheme or call a method using a direct call (like a static procedure). The Standard Library date class module takes advantage of this trick to make it possible to create new date classes with a user-selectable set of procedures and methods. This allows you to create a custom date type that uses methods for those functions you want to override (as methods) and use procedures for those functions you don't call or will never override (as virtual methods). Indeed, the *dateClass_t* and *virtualDateClass_t* date types were created using this technique. The *dateClass_t* data type was created specifying all functions as procedures, the *virtualDateClass_t* data type was created specifying all functions as methods. By using the *dtClass.make_dateClass* macro, you can create new date data types that have any combination of procedures and methods.

```
dtClass.make_dateClass( className, "<list of methods>"  )
```

dtClass.make_dateClass is a macro that generates a new data type. As such, you should only invoke this macro in an HLA type declaration section. This macro requires two arguments: a class name and a string containing the list of methods to use in the new data type. The method list string must contain a sequence of method names (typically separated by spaces, though this isn't strictly necessary) from the following list:

```
today
utc
isLeapYear
isValid

validate

a_toString
toString
setSeparator
setFormat
addDays
subDays
addMonths

subMonths

addYears

subYears
fromJulian
toJulian
dayOfWeek
dayNumber
daysLeft
daysBetween

difference
```

Here is *dtClass.make_dateClass* macro invocation that creates the *virtualDateClass_t* type:

```
type
   dtClass.make_dateClass
```

```
    (
        virtualDateClass,
        "today"
        "isLeapYear"
        "isValid"
        "a_toString"
        "toString"
        "setSeparator"
        "setFormat"
        "addDays"
        "subDays"
        "addMonths"
        "addYears"
        "fromJulian"
        "toJulian"
        "dayOfWeek"
        "dayNumber"
        "daysLeft"
        "daysBetween"
    );
```

(For those unfamiliar with the syntax, HLA automatically concatenates string literals that are separated by nothing but whitespace; therefore, this macro contains exactly two arguments, the virtualDateClass_t name and a single string containing the concatenation of all the strings above.)

From this macro invocation, HLA creates a new data type using methods for each of the names appearing in the string argument. If a particular date function's name is not present in the *dtClass.make_dateClass* macro invocation, then HLA creates a static procedure for that function. As a second example, consider the declaration of the dateClass_t data type (which uses static procedures for all the date functions):

```
type
    dtClass.make_dateClass( dateClass_t, " " );
```

Because the function string does not contain any of the date function names, the *dtClass.make_dateClass* macro generates static procedures for all the date functions.

The *dateClass_t* type is great if you don't need to create a derived date class that allows you to polymorphically override any of the date functions. If you do need to create methods for certain functions and you don't mind linking in all the date class functions (and you don't mind the extra overhead of a method call, even for those functions you're not overloading), the *virtualDateClass_t* is convenient to use because it makes all the functions virtual (that is, methods). Probably 99% of the time you won't be calling the date functions very often, so the overhead of using method invocations for all date functions is irrelevant. In those rare cases where you do need to support polymorphism for a few date functions but don't want to link in the entire set of date functions, or you don't want to pay the overhead for indirect calls to functions that are never polymorphic, you can create a new date class type that specifies exactly which functions require polymorphism.

For example, if you want to create a date class that overrides the definition of the fromJulian and toJulian functions, you could declare that new type thusly:

```
type
    dtClass.make_dateClass
     (
         myDateClass,
         "fromJulian"
         "toJulian"
     );
```

This new class type (*myDateClass*) has two methods, *fromJulian* and *toJulian*, and all the other date functions are static procedures. This allows you to create a derived class that overloads the *fromJulian* and *toJulian* methods and access those methods when using a generic *myDateClass* pointer, e.g.,

```
type
    derivedMyDateClass :
```

```
    class inherits( myDateClass );

        override method fromJulian;
        override method toJulian;

    endclass;
```

It is important for you to understand that types created by *dtClass.make_dateClass* are base types. They are not derived from any other class (e.g., *virtualDateClass_t* is not derived from *dateClass_t* or vice-versa). The types created by the *dtClass.make_dateClass* macro are independent and incompatible types. For this reason, you should avoid using different base date class types in your program. Pick (or create) a base date class and use that one exclusively in an application. You'll avoid confusion by following this rule.

For the sake of completeness, here are the macros that the Standard Library uses to create date data types:

```
namespace dtClass;

    // The following macro allows us to turn a class function
    // into either a method or a procedure based on the
    // presence of "funcName" within a list of method names
    // passed to the class generating macro.

    #macro function( funcName );

        #if( @index( methods, 0, @string:funcName) = -1 )

            procedure funcName

        #else

            method funcName

        #endif

    #endmacro




    #macro make_dateClass( className, methods );

        className:
            class

                var
                    theDate:    date.daterec;
                    OutFmt:     date.OutputFormat;
                    Separator:  char;

                procedure create;
                    @external( "DATECLASS_CREATE");

                dtClass.function( today );
                    @external( "DATECLASS_TODAY");

                dtClass.function( utc );
                    @external( "DATECLASS_UTC");

                dtClass.function( isLeapYear );
                    @returns( "al" );
                    @external( "DATECLASS_ISLEAPYEAR");

                dtClass.function( isValid );
```

```
        @returns( "al" );
        @external( "DATECLASS_ISVALID");

    dtClass.function( validate );
        @returns( "al" );
        @external( "DATECLASS_VALIDATE");

    dtClass.function( a_toString );
        @returns( "eax" );
        @external( "DATECLASS_A_TOSTRING");

    dtClass.function( toString )( dest:string );
        @external( "DATECLASS_TOSTRING");

    dtClass.function( setSeparator )( c:char );
        @external( "DATECLASS_SETSEPARATOR");

    dtClass.function( setFormat )( f:date.OutputFormat );
        @external( "DATECLASS_SETFORMAT");

    dtClass.function( addDays )( days:uns32 );
        @external( "DATECLASS_ADDDAYS");

    dtClass.function( subDays )( days:uns32 );
        @external( "DATECLASS_SUBDAYS");

    dtClass.function( addMonths )( months:uns32 );
        @external( "DATECLASS_ADDMONTHS");

    dtClass.function( subMonths )( days:uns32 );
        @external( "DATECLASS_SUBMONTHS");

    dtClass.function( addYears )( years:uns32 );
        @external( "DATECLASS_ADDYEARS");

    dtClass.function( subYears )( days:uns32 );
        @external( "DATECLASS_SUBYEARS");

    dtClass.function( fromJulian )( Julian:uns32 );
        @external( "DATECLASS_FROMJULIAN");

    dtClass.function( toJulian );
        @returns( "eax" );
        @external( "DATECLASS_TOJULIAN");

    dtClass.function( dayOfWeek );
        @returns( "eax" );
        @external( "DATECLASS_DAYOFWEEK");

    dtClass.function( dayNumber );
        @returns( "eax" );
        @external( "DATECLASS_DAYNUMBER");

    dtClass.function( daysLeft );
        @returns( "eax" );
        @external( "DATECLASS_DAYSLEFT");

    dtClass.function( daysBetween )
    (
        otherDate:date.daterec
    );
```

```
                    @returns( "eax" );
                    @external( "DATECLASS_DAYSBETWEEN");

                dtClass.function( difference )
                (
                    var otherDate:className in eax
                );
                    @returns( "eax" );
                    @external( "DATECLASS_DIFFERENCE");

            endclass
        #endmacro

    end dtClass;
```

If you look closely at the *make_dateClass* macro, you'll notice that it maps all the functions, be they methods or procedures, to the *dateClass_t* names (which are all procedures, if you look at the source code for these functions). As noted earlier, the function code for methods and procedures is exactly the same, only the call to a given function is different based on whether it is a method or a procedure. Therefore, the *dtClass.make_dateClass* macro maps all functions to the same set of procedures. Therefore, if you do create and use multiple date classes in the same application, the linker will only link in one set of routines (unless, of course, you overload some methods, in which case the linker will link in your new functions as well as the original *dateClass_t* set).

11.9.3 Date Class Functions

The date class type supports most of the functions associated with the date type. The main difference is that the date class functions operate directly on the date object rather than on a date value you pass as a parameter. For this reason, there aren't any macros that overload the date function parameter lists.

The following sections do not include sample code demonstrating the calling sequences for a couple of reasons:

For high level calls, the syntax deponds on the object name and type.

Low-level calling sequences don't appear here because it doesn't really make sense to make a low-level object invocation; people wanting to make low-level calls will probably use the standard date procedures rather than the object-oriented ones.

These functions are really intended for use by programmers experienced with HLA's Object-oriented assembly facilities. Note that the dtClass.hhf header file is not automatically included by stdlib.hhf; this reflects the more advanced nature of the date class module.

For the same reasons, there are no stack diagrams for these function calls. If you want more information on making calls to HLA class methods and procedures, please consult the HLA documentation.

In the following function descriptions, the symbol <object> is used to specify a date class object or a pointer to a date class object. Note that class invocations of static procedures (e.g., "dateClass_t.isLeapYear") are illegal with the single exception of the constructor (the create procedure). If you call a date class procedure directly, the system will raise an exception (as ESI, which should be pointing at the object's data, will contain NULL).

`<object>.create();`

The *<name>.create* procedure is the object constructor. This is the only function that you may call using a class name rather than an object name. For example, *dateClass_t.create();* is a perfectly legitimate constructor call. As is the convention for HLA class constructors, if you call a class constructor directly (using the class name rather than an object name), the date class constructor will allocate storage for a new date class object on the heap and return a pointer to the new object in ESI. Once the storage is allocated (or if you specify the name of a previously-allocated object rather than the class name), the date class constructor will initialize all the fields of the object to reasonable values (in particular, the constructor initializes the VMT pointer, initializes *theDate* to a valid date, and sets up the *OutFmt* and *Separator* fields with default values).

If you create a derived date class and add new data fields to the data type, you should override the *create* procedure and initialize those new fields in the overridden procedure. See the HLA documentation or *The Art of Assembly Language* for more details on derived classes and overriding constructors.

<object>.isLeapYear(); @returns("al");

This function returns true or false in the AL register depending upon whether the object's *theDate* field is a leap year (true if it is a leap year). See the discussion of *date.isLeapYear* for more details.

<object>.validate();

This function checks the object's *theDate* field and raises an ex.InvalidDate exception if the date is not a valid date between 1/1/1600 and 12/31/2999. See the discussion of *date.validate* for more details.

<object>.isValid(); @returns("al");

This function checks the object's *theDate* field and returns false in EAX if the date is not a valid date between 1/1/1600 and 12/31/2999 (it returns true otherwise). See the discussion of *date.isValid* for more details.

<object>.toJulian(); @returns("eax");

This function converts the Gregorian (i.e., standard) date found in the object's *theDate* field into a Julian day number. See the discussion of *date.toJulian* for more details.

<object>.fromJulian(jd:uns32);

This function converts the Julian Day Number passed as the argument to a Gregorian (i.e., standard) date and stores the result the object's *theDate* field. See the discussion of *date.fromJulian* for more details. This function will raise an *ex.InvalidDate* exception if the Julian Day Number conversion produces a date outside the range 1/1/1600 to 12/31/2999. See the discussion of *date.fromJulian* for more details.

<object>.dayNumber(); @returns("eax");

This function converts the Gregorian date found in the objects *theDate* field into a day number into the current year. It returns the day number in the EAX register. See the discussion of *date.dayNumber* for more details.

<object>.daysLeft(); @returns("eax");

This function returns the number of days left in the current year *counting the object's theDate field.* See the discussion of *date.daysLeft* for more details.

<object>.dayOfWeek(); @returns("eax");

These functions return, in EAX, a value between zero and six denoting the day of the week of *theDate*. See the discussion of *date.dayOfWeek* for more details.

<object>.daysBetween(otherDate:daterec); @returns("eax");

This function returns an uns32 value in EAX that gives the number of days between object's date and the daterec value passed as a parameter dates. See the discussion of *date.daysBetween* for more details.

<object>.difference(var otherDate:<classType>); @returns("eax");

This function returns an uns32 value in EAX that gives the number of days between object's date and the date class object value passed as a parameter. The parameter type ("<classType>") must be the same type as <object>. See the discussion of *date.daysBetween* for more details.

<object>.addDays(days:uns32);

This procedure adds the parameter value (in days) directly to the object's *theDate* field. See the discussion of *date.addDays* for more details.

```
<object>.addMonths( months:uns32; var dr:date.daterec );
```

This procedure adds the parameter value (in months) directly to the object's *theDate* field. See the discussion of *date.addMonths* for more details.

```
<object>.addYears( years:uns32; var dr:date.daterec );
```

This procedure adds the parameter value (in years) directly to the object's *theDate* field. See the discussion of *date.addYears* for more details.

```
<object>.subDays( days:uns32; var dr:date.daterec );
```

This procedure subtracts the parameter value (in days) directly from the object's *theDate* field. See the discussion of *date.subDays* for more details.

```
<object>.subMonths( months:uns32; var dr:date.daterec );
```

This procedure subtracts the parameter value (in months) directly from the object's *theDate* field. See the discussion of *date.subMonths* for more details.

```
<object>.subYears( years:uns32; var dr:date.daterec );
```

This procedure subtracts the parameter value (in years) directly from the object's *theDate* field. See the discussion of *date.subYears* for more details.

```
<object>.today();
```

Stores the local date (today's date) into the object's *theDate* field. See the discussion of *date.today* for more details.

```
<object>.utc( var dr:date.daterec );
```

Stores the UTC date (today's GMT date) into the object's *theDate* field. See the discussion of *date.utc* for more details.

```
<object>.toString( s:string );
```

This function converts the object's *theDate* field to a string (using the output format specified by the object's *theDate* field and the separator character specified by the object's *OutFmt* field) and stores the result in the specified string. See the discussion of *date.toString* for more details.

```
<object>.a_toString( dr:daterec ); @returns( "eax" );
```

This function converts the object's *theDate* field to a string (using the output format specified by the object's *theDate* field and the separator character specified by the object's *OutFmt* field) and stores the result in storage it allocates on the heap. This function returns a pointer to the new string in the EAX register. See the discussion of *date.a_toString* for more details.