35 Time Functions (datetime.hhf)

HLA contains a set of procedures and functions that simplify *correct* time calculations. The time module contains functions and other objects that manipulate time in terms of hours, minutes, and seconds. This includes functions that read the current time, perform time arithmetic, do time conversions, and output time values.

There are two sets of time functions available in the standard library: the standard time functions and a set of time classes. This document will describe both sets of time functions.

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 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.

35.1 Time Module

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

35.2 Time Data Types

The principal time data structure is the *time.timerec* record:

`time.timerec`

This data structure has the following definition:

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

The *time* field allows you to treat the entire object as a single 32-bit value. This is great for comparisons or for passing the *timerec* value around in an opaque fashion.

The standard library uses the *time.timerec* data type to hold valid times in the range 00:00:00 to 23:59:59. Values outside this range are invalid and the standard library will raise an exception if you try to use such values in a *time.timerec* object. Sometimes, however, it is convenient to measure time as a duration rather than as a time of day. The standard library provides the *time.duration* data type for this purpose. The *time.duration* data type is structurally identical to the time.timerec data type. However, the standard library routines allow any 16-bit signed value for the hours *fields*. Note that the *mins* and *secs* fields are still limited to the range 0..59 (and are considered invalid if they are outside this range).

```
duration:
 record;
```

```
            secs:int8;
            mins:int8;
            hours:int16;

        endrecord;
```

The *time.OutputFormat* data type controls how the string conversion functions format time values when converting them to strings. This is an enumerated data type with the following values:

```
OutputFormat:   enum
                {
                    hhmmssAMPM,
                    hhmmssAP,
                    hhmmss12,
                    hhmmss24,

                    hhmmAMPM,
                    hhmmAP,
                    hhmm12,
                    hhmm24,

                    badTimeFormat
                };
```

The Standard Library maintains an internal static variable that keeps track of the current output format (which you can change via the time.setFormat function). The various settings affect the output format as follows:

hhmmssAMPM: Date is output using a 12-hour clock in the range 01:00:00 to 12:59:59 with an "AM" or "PM" suffix on the time.

hhmmssAP: Date conversion uses a 12-hour clock in the range 01:00:00 to 12:59:59 with an "A" or "P" suffix on the time.

hhmmss12: Date conversion uses a 12-hour clock in the range 01:00:00 to 12:59:59 with no suffix to denote morning or evening times.

hhmmss24: Date conversion uses a 24-hour clock in the range 00:00:00 to 23:59:59.

hhmmAMPM: Date is output using a 12-hour clock in the range 01:00 to 12:59 with an "AM" or "PM" suffix on the time.

hhmmAP: Date conversion uses a 12-hour clock in the range 01:00 to 12:59 with an "A" or "P" suffix on the time.

hhmm12: Date conversion uses a 12-hour clock in the range 01:00 to 12:59 with no suffix to denote morning or evening times.

hhmm24: Date conversion uses a 24-hour clock in the range 00:00 to 23:59.

35.3 Time Predicates

The functions in this category test times for validity and do other checks on times.

```
time.validate( h:word; m:byte; s:byte );
time.validate( hms:time.timerec );
time._validate( tm:timerec );
```

HLA high-level calling sequence examples:

```
try

  time.validate( someTimeVar );

  anyexception

  // Do something if the time is invalid

endtry;

try

  time.validate( someHour, someMinute, someSecond );

  anyexception

  // Do something if the time is invalid

endtry;

try

  time._validate( someTimeVar );

  anyexception

  // Do something if the time is invalid

endtry;


HLA low-level calling sequence examples:

push( someTimeVar.time );
call time._validate;
```

The functions in this category test times for validity and do other checks on times.

time.isValid(h:word; m:byte; s:byte);
time.isValid(hms:time.timerec);
time._isValid(tm:timerec);

```
HLA high-level calling sequence examples:

time.isValid( someTimeVar );
mov( al, timeIsValidVar1 );
time.isValid( someHour, someMinute, someSecond );
mov( al, timeIsValid2 );
time._validate( someTimeVar );
mov( al, timeIsValid3 );


HLA low-level calling sequence examples:

push( someTimeVar.time );
call time._isValid;
mov( al, timeIsValid3 );
```

35.4 Time Conversions

The functions in this category convert time between hours/minutes/second format and an integer specifying some number of seconds, and the functions in this category also perform basic time arithmetic functions such as the addtion and subtraction of time.

#macro unpack(tm, h, m, s)

This macro takes a *time.timerec* object as its first argument and extracts the *hours*, *mins*, and *secs* fields (zero-extending them to 32 bits) and stores the extract values in the dword *h*, *m*, and *s* arguments (respectively).

```
HLA macro invocation examples:

time.unpack( sometimeVar, hoursVar32, MinutesVar32, SecondsVar32 );
```

#macro pack(h, m, s, _tm_)

This macro takes the hours (h), minutes (m), and seconds (s) arguments and packs them into a *time.timerec* object. This macro is very similar to the *date.pack* macro, see the description of that macro for more details about the operation of this macro. Note that if h, m, or s are constant values, this macro will check them to see if they are valid (that is, values in the range 00:00:00 to 23:59:59).

```
HLA macro invocation examples:

time.unpack( hoursVar32, MinutesVar32, SecondsVar32, someTimeVar );
```

time.durationToSecs(hours:word; mins:byte; secs:byte); @returns("eax");

This function converts a time span in HHMMSS format to some number of seconds (if HHMMSS is the time of day, then these functions return the time in seconds since midnight). Note that HHMMSS does not have to be a 12-hour or 24-hour clock value. You may specify any number of hours between 0 and 65535, and any number of seconds or minutes between 0 and 255 for this function.

```
time.secsToDuration
(
        seconds :uns32;
    var hours   :word;
    var mins    :byte;
    var secs    :byte
);
```

This function converts some number of seconds to a duration (the number of hours, minutes, and seconds) and stores that duration in the *hours*, *mins*, and *seconds* parameters passed by reference. If the number of seconds exceeds 65535 hours, 59 minutes, and 59 seeconds, then this function raises an *ex.TimeOverflow* exception.

```
HLA high-level calling sequence examples:

time.secsToDuration( seconds, hoursVar32, MinsVar32, SecsVar32 );

HLA low-level calling sequence examples:

push( seconds );
pushd( &hoursVar32 );// Assumes hoursVar32 is STATIC
lea( eax, MinsVar32 );// MinsVar32 need not be static
push( eax );
lea( eax, SecsVar32 );// SecsVar32 need not be static
push( eax );
call time.secsToDuration;
```

#macro time.toSecs(theTime: time.timerec); @returns("eax");
#macro time.toSecs(h:uns16; m:byte; s:byte); @returns("eax");
time._toSecs(HMS:timerec); @returns("eax");

These functions convert a time span in HHMMSS format to some number of seconds (if HHMMSS is the time of day, then these functions return the time in seconds since midnight). Technically, these functions and macros don't care if their parameters are valid times (that is, within the range 00:00:00 to 23:59:59), however you should use *time.durationToSecs* when convertion durations (versus valid time of day values) to seconds.

```
HLA high-level calling sequence examples:

time.toSecs( someTimeVar );
mov( eax, numSeconds1 );
time.toSecs( hours, minutes, seconds );
mov( eax, numSeconds2 );
time._toSecs( someTimeVar );
mov( eax, numSeconds3 );

HLA low-level calling sequence examples:

push( someTimeVar.time );
call time.toSecs;
mov( eax, numSeconds3 );
```

time.fromSecs(seconds:uns32; var HMS:time.timerec);

This function converts the seconds parameter to an HMS time value. The first parameter must be less than 235,929,600 since this is the maximum time representable by 65535 hours. If the seconds parameter exceeds this value, then *time.secsToHMS* will raise an *ex.TimeOverflow* exception.

```
HLA high-level calling sequence example:

time.fromSecs( seconds, someTimeVar );

HLA low-level calling sequence examples:

push( seconds );
lea( eax, someTimeVar );// If someTimeVar is non-static
call time.fromSecs;

push( seconds );
pushd( &someTimeVar );// If someTimeVar is static
```

time.toUnixTime(DMY:date.daterec; HMS:timerec);
@returns("edx:eax");

This function converts a Standard Library date and time value to a UNIX/C stdlib date/time value. UNIX/C stdlib time values are specified as the number of seconds since midnight, Jan 1, 1970. This function raises an *ex.InvalidDate* exception if the DMY parameter specifies a date prior to Jan 1, 1970.

This function returns a 64-bit value. Most UNIX systems and C standard library packages currently specify a "*time_t*" object as a 32-bit signed integer. This data type will fail to properly maintain dates sometime during the year 2038. Newer system define *time_t* as an unsigned 32-bit integer, thereby doubling the effective range of the date. Nevertheless, the *date.daterec* data type can represent dates outside the range of even a 32-bit unsigned integer, so this function returns a 64-bit value in EDX:EAX. If you need to work with a 32-bit value, simply ignore the value returned in EDX.

```
HLA high-level calling sequence example:
```

```
time.toUnixTime( someDate, someTime );
mov( eax, (type dword unixTimeVar));
mov( edx, (type dword unixTimeVar[4]));// Assuming it's 64 bits.

HLA low-level calling sequence examples:

push( someDate.date );
push( someTime.time );
call time.toUnixTime;
mov( eax, (type dword unixTimeVar));
mov( edx, (type dword unixTimeVar[4]));// Assuming it's 64 bits.
```

```
time.fromUnixTime
(
        unixTime    :qword;
    var HMS         :timerec;
    var DMY         :date.daterec
);
```

This function converts the UNIX/C standard library *time_t* object passed in *unixTime* to HLA Standard Library *date.daterec* (*DMY*) and *time.timerec* (*HMS*) objects. Note that the *time_t* type on most Unix systems (and in the C standard library) is a 32-bit value whereas this function expects a 64-bit value. If working with actual 32-bit *time_t* values, simply zero extend them to 64 bits before calling this function.

```
HLA high-level calling sequence example:

time.fromUnixTime( unixDateTime, someDate, someTime );

HLA low-level calling sequence examples:

  // If the unix date/time on your system is 32 bits:

  pushd( 0 );
  push( (type dword unixTimeVar));
pushd( &someDate.date );// Assumes someDate.date and
pushd( &someTime.time );// someTime.time are STATIC
call time.fromUnixTime;

  // If the unix date/time on your system is 64 bits:

  push( (type dword unixTimeVar[4]));
  push( (type dword unixTimeVar));
lea( eax, someDate.date );// Assumes someDate.date and
push( eax );              // someTime.time are not STATIC
lea( eax, someTime.time );
push( eax );
call time.fromUnixTime;
```

```
time.toWinFileTime( DMY:date.daterec; HMS:timerec );
     @returns( "edx:eax" );
```

Windows file times are 64-bit values that represent the number of 100 nanosecond periods since midnight, Jan 1, 1601. This function converts a Standard Library date and time value (passed in the *DMY* and *HMS* parameters) to a Windows file time and returns that value in the EDX:EAX register pair. Because HLA time values only maintain seconds precision, the resulting value will have a granularity of one second. If you actually need to create a value with finer granularity, add the number of 0.1 microseconds to the result that *time.toWinFileTime* returns. This function raises an *ex.InvalidDate* exception if the Standard library date is less than Jan 1, 1601.

```
HLA high-level calling sequence example:

time.toWinFileTime( someDate, someTime );
mov( eax, (type dword win32TimeVar));
mov( edx, (type dword win32TimeVar[4]));

HLA low-level calling sequence examples:

push( someDate.date );
push( someTime.time );
call time.toWinFileTime;
mov( eax, (type dword win32TimeVar));
mov( edx, (type dword win32TimeVar[4]));
```

```
time.fromWinFileTime
(
        winTime  :qword;
    var HMS      :timerec;
    var DMY      :date.daterec
);
```

This function converts a Windows file time to a Standard Library date and time. This function stores the resulting date in the *DMY* parameter and the time to the *HMS* parameter (both passed by reference). Because Windows file times provide 100ns precision whereas the Standard Library functions only work with 1 sec precision, this function rounds the Windows time to the nearest second during the conversion (specfically, if there are 0.5 or more fractional seconds, this function bumps up the seconds value by one).

```
HLA high-level calling sequence example:

time.fromWinFileTime( win32DateTime, someDate, someTime );

HLA low-level calling sequence example:

  push( (type dword win32DateTime[4]));
  push( (type dword win32DateTime));
lea( eax, someDate.date );// Assumes someDate.date and
push( eax );             // someTime.time are not STATIC
lea( eax, someTime.time );
push( eax );
call time.fromWinFileTime;
```

35.5 Time Arithmetic

time.secsBetweenTimes(time1:timerec; time2:timerec); @returns("eax");

This function computes the number of seconds between the two times passed as parameters. It returns the value in the EAX register. Note that this is the absolute value of their difference, so the relative sizes of the two operands is immaterial. Both times must be valid Standard Library timerec values in the range 00:00:00..23:59:59 or this function will raise an *ex.InvalidTime* exception.

```
HLA high-level calling sequence example:

time.secsBetweenTimes( someTime1, someTime2 );
mov( eax, secondsBetween );

HLA low-level calling sequence example:
      push( someTime1.time );
      push( someTime2.time );
    call time.secsBetweenTimes;
    mov( secondsBetween );
```

time.subHours(hours:uns32; var HMS:timerec); @returns("eax");

This function subtracts the number of *hours* from the *timerec* object passed by reference as the second parameter (*HMS*). This function returns in EAX the number of days "borrowed" during the calculation (that is, for each transition past midnight during this calculation, the calculation "borrows" one day).

```
    HLA high-level calling sequence example:

    time.subHours( hours, someTime );

    HLA low-level calling sequence examples:

      push( hours );
      pushd( &someTime);// Assuming someTime is STATIC
    call time.subHours;

      push( hours );
      lea( eax, someTime );// Assuming someTime is not STATIC
      push( eax );
    call time.subHours;
```

time.subMins(minutes:uns32; var HMS:timerec); @returns("eax");

This function subtracts the number of *minutes* from the *timerec* object passed by reference as the second parameter (*HMS*). This function returns in EAX the number of days "borrowed" during the calculation (that is, for each transition past midnight during this calculation, the calculation "borrows" one day).

```
    HLA high-level calling sequence example:

    time.subMins( minutes, someTime );

    HLA low-level calling sequence examples:

      push( minutes );
      pushd( &someTime );// Assuming someTime is STATIC
    call time.subMins;

      push( minutes );
      lea( eax, someTime );// Assuming someTime is not STATIC
      push( eax );
    call time.subMins;
```

time.subSecs(seconds:uns32; var HMS:timerec); @returns("eax");

This function subtracts the number of *seconds* from the *timerec* object passed by reference as the second parameter (*HMS*). This function returns in EAX the number of days "borrowed" during the calculation (that is, for each transition past midnight during this calculation, the calculation "borrows" one day).

```
    HLA high-level calling sequence example:

    time.subSecs( seconds, someTime );
```

```
HLA low-level calling sequence examples:

  push( seconds);
  pushd( &someTime );// Assuming someTime is STATIC
call time.subSecs;

  push( seconds);
  lea( eax, someTime );// Assuming someTime is not STATIC
  push( eax );
call time.subSecs;
```

time.addHours(hours:uns32; var HMS:timerec); @returns("eax");

This function adds the number of *hours* to the *timerec* object passed by reference as the second parameter (*HMS*). This function returns in EAX the number of days skipped during the calculation (that is, for each transition past midnight during this calculation, the calculation "skips" one day).

```
HLA high-level calling sequence example:

time.addHours( hours, someTime );

HLA low-level calling sequence examples:

  push( hours );
  pushd( &someTime);// Assuming someTime is STATIC
call time.addHours;

  push( hours );
  lea( eax, someTime );// Assuming someTime is not STATIC
  push( eax );
call time.addHours;
```

time.addMins(minutes:uns32; var HMS:timerec); @returns("eax");

This function adds the number of *minutes* to the *timerec* object passed by reference as the second parameter (*HMS*). This function returns in EAX the number of days skipped during the calculation (that is, for each transition past midnight during this calculation, the calculation "skips" one day).

```
HLA high-level calling sequence example:

time.addMins( minutes, someTime );

HLA low-level calling sequence examples:

  push( minutes );
  pushd( &someTime );// Assuming someTime is STATIC
call time.addMins;

  push( minutes );
  lea( eax, someTime );// Assuming someTime is not STATIC
  push( eax );
call time.addMins;
```

time.addSecs(seconds:uns32; var HMS:timerec); @returns("eax");

This function adds the number of *seconds* to the *timerec* object passed by reference as the second parameter (*HMS*). This function returns in EAX the number of days skipped during the calculation (that is, for each transition past midnight during this calculation, the calculation "skips" one day).

```
HLA high-level calling sequence example:

time.addSecs( seconds, someTime );

HLA low-level calling sequence examples:

  push( seconds);
  pushd( &someTime );// Assuming someTime is STATIC
call time.addSecs;

  push( seconds);
  lea( eax, someTime );// Assuming someTime is not STATIC
  push( eax );
call time.addSecs;
```

35.6 Reading the Current System Time

`time.curTime( var theTime: time.timerec );`

This returns the local time (provided by the system clock) in the specified time variable.

```
HLA high-level calling sequence example:

time.curTime( someTime );

HLA low-level calling sequence examples:

  pushd( &someTime );// Assuming someTime is STATIC
call time.curTime;

  lea( eax, someTime );// Assuming someTime is not STATIC
  push( eax );
call time.curTime;
```

`time.utcTime( var theTime: time.timerec );`

This returns the UTC time (the current GMT time provided by the system clock) in the specified time variable.

```
HLA high-level calling sequence example:

time.utcTime( someTime );

HLA low-level calling sequence examples:

  pushd( &someTime );// Assuming someTime is STATIC
call time.utcTime;

  lea( eax, someTime );// Assuming someTime is not STATIC
  push( eax );
call time.utcTime;
```

35.7 Time String Conversions and Output

`time.setFormat( f:OutputFormat );`

This function sets the global system time conversion value. The parameter must be one of the following time.OutputFormat enumerated values:

```
hhmmssAMPM
hhmmssAP
hhmmss12
hhmmss24
hhmmAMPM
hhmmAP
hhmm12
hhmm24
```

The first four constants tell the time/string conversion routines to emit hours, minutes, and seconds in a "00:00:00" format, the last four output only the hours and minutes in a "00:00" format. The *hhmmssAMPM* and *hhmmAMPM* constants emit a 12-hour time format with either "am" or "pm" appended to the string to denote midnight to noon or noon to midnight. The *hhmmssAP* and *hhmmAP* formats do the same, except that they only display an "a" or a "p" after the time. The *hhmmss12* and *hhmm12* formats display a 12-hour time with no indication of which half of the day the time represents. The *hhmmss24* and *hhmm24* formats specify a 24-hour time.

The *time.toString* and *time.a_toString* functions use the value of the global time format to determine how they convert a Standard Library timerec value to a string.

`time.toString( HMS:timerec; dest:string );`

This function converts the *HMS* parameter to a string using the format specified by the global *OutputFormat* variable (set by the time.setFormat function). The destination string must have sufficient storage associated with it or this function will raise an exception. This function will also raise an *ex.InvalidTime* exception if *HMS* contains an invalid time.

```
HLA high-level calling sequence example:

time.toString( someTime, destStr );

HLA low-level calling sequence example:

  push( someTime.time );
  push( destStr );
call time.toString;
```

`time.a_toString( HMS:timerec ); @returns( "eax" );`

This function is similar to time.toString except you don’t supply a destination string. Instead, this function allocates storage for the string on the heap. Note that it is the caller’s responsibility to free this storage when the caller is done with the string (i.e., by calling *str.free*).

```
HLA high-level calling sequence example:

time.a_toString( someTime );
mov( eax, destStr );

HLA low-level calling sequence example:

  push( someTime.time );
call time.a_toString;
mov( eax, destStr );
```

35.8 Time Class Types

```
#include( "dtClass.hhf" )
```

Note: the stdlib.hhf header file does not include dtClass.hhf. If you want to use the time class data types you will need to explicitly include the dtClass.hhf header file.

For those who prefer an object-oriented programming approach, the Standard Library provides the ability to create time class data types. The Standard Library provides two predefined time class types: *timeClass_t* and *virtualTimeClass_t*. The difference between these two types is that the *timeClass_t* type uses static procedures for all the time functions whereas *virtualTimeClass_t* uses virtual methods. In certain cases, using the *timeClass_t* data type is more efficient than using *virtualTimeClass_t* because you only link in the class functions you actually call. However, you lose the ability to make polymorphic method calls when using the *timeClass_t*. For more details on the differences between these two class types, please see the discussion of the *dtClass.make_timeClass* macro appearing later in this section. This section will use the phrase "time class" to mean any class created by the *make_timeClass* macro, including the *timeClass_t* and *virtualTimeClass_t* data types.

The time class types provide two data fields:

```
var
    theTime :time.timerec;
    timeFmt :time.OutputFormat;
```

The first field, *theTime*, holds the time value associated with the time object. This is the standard *time.timerec* 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 *time.timerec* value.

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

Thread Safety Issue: Although each time object has its own *timeFmt* field, this does not make the use of time class objects thread safe. When converting *theTime* to a string, the time class functions save the global format value, copy *timeFmt* to the global variable, call the time functions to do the string conversion, and then restore the original global value. If a thread is suspended during this activity then any time/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 time/string conversions if you perform such conversions in multiple threads in your application.

Of course, you may create a derived class from either *timeClass_t* or *virtualTimeClass_t* (or create a brand new time class using the *dtClass.make_timeClass* macro) and add any other fields you like to that new time class. One suggestion for such a class is to pad the data fields to a multiple of four bytes. Currently, the *timeClass_t* and *virtualTimeClass_t* objects consumes nine bytes of storage (five 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. Another suggestion might be to add a *Separator* field that specifies the hours/minutes/seconds separator character when converting a time to a string; of course, you'll need to override the *toString* and *a_toString* methods to achieve this.

35.9 Time Class Methods/Procedures

In most HLA classes, there are two types of functions: (static) procedures and (dynamic) methods (there are also iterators, but the time 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 object-oriented benefits of 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. Let's consider those issues here.

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 the linker links in the code for each method in the class. This can make your program a little larger because it may be including several time class functions that you don't actually call. For large applications, the amount of extra storage required by linking in all the time functions is inconsequential, but if you don't like linking in code that the program will never call, specifying virtual methods for all the time functions may annoy you.

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 *timeClass_t* and *virtualTimeClass_t* functions differ in how they define the functions appearing in the class types. The *timeClass_t* type uses static procedures for all functions, the *virtualTimeClass_t* type uses methods for all class functions (except the constructor, *create*, because constructors are always static procedures). Therefore, *timeClass_t* data types will make direct calls to all the functions (and only link in the procedures you actually call); however, *timeClass_t* objects do not support function polymorphism in derived classes. The *virtualTimeClass_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 *timeClass_t* and *virtualTimeClass_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.

35.10 Creating New Time 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 time class module takes advantage of this trick to make it possible to create new time classes with a user-selectable set of procedures and methods. This allows you to create a custom time 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 *timeClass_t* and *virtualTimeClass_t* time types were created using this technique. The *timeClass_t* data type was created specifying all functions as procedures, the *virtualTimeClass_t* data type was created specifying all functions, except create, as methods. By using the *dtClass.make_timeClass* macro, you can create new time data types that have any combination of procedures and methods.

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

dtClass.make_timeClass 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 from the following list:

```
create
curTime
utcTime
addSecs
addMins
addHours
subSecs
subMins
subHours
fromSecs
toSecs
isValid
validate
difference
secsBetweenTimes
toString
a_toString
```

Here is *dtClass.make_timeClass* macro invocation that creates the *virtualTimeClass_t* type; note that the create function is always a static procedure and its name must not appear in the list of method names:

```
type
   dtClass.make_timeClass
    (
        virtualTimeClass_t,
        "curTime "
        "utcTime "
        "addSecs "
        "addMins "
        "addHours "
        "subSecs "
        "subMins "
        "subHours "
        "fromSecs "
        "toSecs "
        "isValid "
        "validate "
        "difference "
        "secsBetweenTimes "
        "toString "
        "a_toString "

    );
```

(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 *virtualTimeClass_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 time function's name is not present in the *dtClass.make_timeClass* macro invocation, then HLA creates a static procedure for that function. As a second example, consider the declaration of the *timeClass_t* data type (which uses static procedures for all the time functions):

```
type
   dtClass.make_timeClass( timeClass_t, " " );
```

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

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

For example, if you want to create a time class that overrides the definition of the *fromSecs* and *toSecs* functions, you could declare that new type thusly:

```
type
   dtClass.make_timeClass
    (
        myTimeClass,
        "fromSecs"
        "toSecs"
    );
```

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

```
type
   derivedMyTimeClass :
      class inherits( myTimeClass );

          override method fromSecs;
          override method toSecs;

      endclass;
```

It is important for you to understand that types created by *dtClass.make_timeClass* are base types. They are not derived from any other class (e.g., *virtualTimeClass_t* is not derived from *timeClass_t* or vice-versa). The types created by the *dtClass.make_timeClass* macro are independent and incompatible types. For this reason, you should avoid using different base time class types in your program. Pick (or create) a base time 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 time 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




    // make_timeClass -
    //
    //  This macro is used to create a base time class.
    // The first parameter is the name of the class to create.
    // The second parameter is a string listing the 'function'
    // names that you want converted to a class method (if not
    // present, it will be a class procedure).

    #macro make_timeClass( className, methods );

        className:
            class

                var
                    theTime :time.timerec;
                    timeFmt :time.OutputFormat;

                procedure create;
                    @external( "TIMECLASS_CREATE" );

                dtClass.function( curTime );
                    @external( "TIMECLASS_CURTIME" );
                    dtClass.function( utcTime );
                        @external( "TIMECLASS_UTCTIME" );


                    dtClass.function( addSecs )( seconds:uns32 );
                        @external( "TIMECLASS_ADDSECS" );

                    dtClass.function( addMins )( minutes:uns32 );
                        @external( "TIMECLASS_ADDMINS" );

                    dtClass.function( addHours )( hours:uns32 );
                        @external( "TIMECLASS_ADDHOURS" );


                    dtClass.function( subSecs )( seconds:uns32 );
                        @external( "TIMECLASS_SUBSECS" );

                    dtClass.function( subMins )( minutes:uns32 );
                        @external( "TIMECLASS_SUBMINS" );

                    dtClass.function( subHours )( hours:uns32 );
                        @external( "TIMECLASS_SUBHOURS" );


                    dtClass.function( fromSecs )( seconds:uns32 );
                        @external( "TIMECLASS_FROMSECS" );

                    dtClass.function( toSecs );
                        @returns( "eax" );
                        @external( "TIMECLASS_TOSECS" );


                    dtClass.function( isValid );
                        @returns( "al" );
                        @external( "TIMECLASS_ISVALID" );

                    dtClass.function( validate );
                        @external( "TIMECLASS_VALIDATE" );


                    dtClass.function( difference )( var time2:className );
                        @returns( "eax" );
                        @external( "TIMECLASS_DIFFERENCE" );

                    dtClass.function( secsBetweenTimes )( time2:time.timerec );
                        @returns( "eax" );
                        @external( "TIMECLASS_SECSBETWEENTIMES" );


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

                    dtClass.function( a_toString );
                        @external( "TIMECLASS_A_TOSTRING" );

                endclass;

        #endmacro

    end dtClass;
```

If you look closely at the *make_timeClass* macro, you'll notice that it maps all the functions, be they methods or procedures, to the *timeClass_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_timeClass* 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 *timeClass_t* set).

35.11 Time Class Functions

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

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

Note: because the syntax varies from declaration to declaration, the following sections do not provide examples of calling these functions, please see the HLA documentation under object-oriented programming or *The Art of Assembly Language Programming* for more details.

`<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, *timeClass_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 time class constructor will allocate storage for a new time 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 time class constructor will initialize all the fields of the object to reasonable values (in particular, the constructor initializes the VMT pointer, initializes *theTime* to a valid time (00:00:00), and sets up the *theFmt* field with a default value).

If you create a derived time 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>.validate();`

The <*object*>*.validate* function checks the validity of an object's *theTime* field. It raises an ex.InvalidTime exception if the object's *theTime* field contains an invalid value (hours outside the range 0..23 or minutes/ seconds outside the range 0..59). See *time.validate* for more details.

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

The <*object*>*.isValid* function checks the validity of an object's *theTime* field. It returns true (in AL, zero-extended into EAX) if *theTime* field contains a valid time value, it returns false otherwise. See *time.isValid* for more details.

`<object>.toSecs(); @returns( "eax" );`

This function converts the object's theTime field (in HH:MM:SS format) to the number of seconds since midnight. This function returns the result in the EAX register. See the *time.toSecs* function description for more details.

`<object>.fromSecs( seconds:uns32 );`

This function converts the parameter value (*seconds*, the number of seconds since midnight) into a standard library compatible HH:MM:SS time format and stores the result in the object's *theTime* field. This function returns the number of overflow days (that is, the number of 24-hour periods) in the EAX register, the value this function stores into *theTime* is always a valid time between 00:00:00 and 23:59:59. See the *time.toSecs* function description for more details.

`<object>.secsBetweenTimes( otherTime:timerec ); @returns( "eax" );`

This function computes the number of seconds between the object's *theTime* value and a time.timerec value you pass as a parameter. It returns the difference, in seconds, in the EAX register. See the *time.secsBetweenTimes* function for more information.

`<object>.difference( var otherTime:<object's class> ); @returns( "eax" );`

This function computes the number of seconds between the object's *theTime* value and same field in the object you pass as a parameter. It returns the difference, in seconds, in the EAX register. The type of the parameter object must be the same type as <*object*> (i.e., *timeClass_t*, *virtualTimeClass_t*, or whatever other time class you've created and defined <*object*> to be). See the *time.secsBetweenTimes* function for more information.

`<object>.subHours( hours:uns32 );`

This function subtracts the number of hours specified by the parameter from the object's *theTime* field. See the *time.subHours* function for more information.

`<object>.subMins( hours:uns32 );`

This function subtracts the number of minutes specified by the parameter from the object's *theTime* field. See the *time.subMins* function for more information.

`<object>.subSecs( hours:uns32 );`

This function subtracts the number of seconds specified by the parameter from the object's *theTime* field. See the *time.subSecs* function for more information.

`<object>.addHours( hours:uns32 );`

This function adds the number of hours specified by the parameter to the object's *theTime* field. See the *time.addHours* function for more information.

`<object>.addMins( minutes:uns32 );`

This function adds the number of minutes specified by the parameter to the object's *theTime* field. See the *time.addMins* function for more information.

`<object>.addSecs( seconds:uns32 );`

This function adds the number of seconds specified by the parameter to the object's *theTime* field. See the *time.addSecs* function for more information.

`<object>.curTime();`

This stores the local time (provided by the system clock) in the object's *theTime* field. See the *time.curTime* function for additional details.

`<object>.utcTime();`

This stores the UTC time (the current GMT time provided by the system clock) in the objects *theTime* field. See the *time.utcTime* function for additional details.

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

This function converts the object's *theTime* field to a string using the object's *OutFmt* field to guide the conversion. This function stores the character data in the string object pointed at by the *dest* parameter (there must be sufficient space allocated for the string or the function will raise an exception). See the *time.toString* function for more information.

`<object>.a_toString( HMS:timerec ); @returns( "eax" );`

This function converts the object's *theTime* field to a string using the object's *OutFmt* field to guide the conversion. This function allocates storage for the resultant string and returns a pointer to this new string in EAX. It is the caller's responsibility to deallocate the storage associated with this string when the caller is done with it. See the *time.a_toString* function for more information.

Because the time class includes an "a_toString" function, you may print time object values using stdout.put and similar *.put Standard Library functions. Note that those functionsl automatically deallocate the storage associated with the string created by *<object>.a_toString*.