18 Advanced HLA Programming

18.1 Writing a DLL in HLA

Dynamic link libraries provide an efficient mechanism for sharing code and cross-language linkage. The HLA language does not require any specific syntax to create a DLL; most of the work is done by the linker. However, to successfully write and call DLLs with HLA, you must follow some standard conventions.

Acknowledgement: I learned much of the material needed to write DLLs in HLA by visiting the following web page and looking at the CRCDemo file (which demonstrates how to write DLLs in assembly language). For more information on DLLs in assembly, you might want to take a look at this page yourself:

`http://www.geocities.com/SiliconValley/Heights/7394/index.html`

I certainly acknowledge stealing lots of information and ideas from this CRC code and documentation.

18.1.1 Creating a Dynamic Link Library

Win32 Dynamic Link Libraries provide a mechanism whereby two or more programs can share the same set of library object modules on the disk. At the very least, DLLs save space on the disk; if properly written and loaded into memory, DLLs can also share run-time memory and reduce swap space usage on the hard disk.

Perhaps even more important that saving space, DLLs provide a mechanism whereby two different programming languages may communicate with one another. Although there is usually no problems calling an assembly language (i.e., HLA) module from any given high level language, DLLs do provide one higher level of generality. In order to achieve this generality, Microsoft had to carefully describe the calling mechanism between DLLs and other modules. In order to communicate data, all languages that support DLLs need to agree on the calling and parameter passing mechanisms.

Microsoft has laid down the following rules for DLLs (among others):

- Procedures/functions with a fixed parameter list use the stdcall calling mechanism.
- Procedures/functions with a variable number of parameters use the C calling mechanism.
- Parameters can be bytes, words, doublewords, pointers, or strings. Pointers are machine addresses; strings are pointers to a zero-terminated sequence of characters, and it is up to the two modules to agree on how to interpret byte, word, or dword data (e.g., char, int16, uns32, etc.)

Stdcall procedures push their parameters from left to right as they are encountered in the parameter list. In stdcall procedures, it is the procedure's responsibility to clean up the parameters pushed on the stack.

HLA uses the stdcall calling mechanism for the HLL-style procedure calls, so this simplifies the interface to DLL code when using fixed parameter lists (variable parameter lists are rare in DLLs, but should they be necessary, one can always drop down into "pure" assembly in HLA and accomodate the DLL).

The only other issue, with respect to stdcall conventions, is the naming convention. The stdcall mechanism *mangles* procedure names. In particular, a procedure name like "XXXX" is translated to "_XXX@nn" where "nn" is the number of bytes of parameters passed to the procedure. HLA does not automatically mangle procedure names, but using the "external" directive you can easily specify the mangled name.

DLLs must provide a special procedure that Windows calls to initialize the procedure. This DLL entry point must use an HLA definition like the following:

procedure dll(instance:dword; reason:dword; reserved:dword); external("_dll@12");

This function must return true in AL if the DLL can be successfully initialized; it returns false if it cannot properly initialize the DLL. Note that "dll" and "_dll@12" are example names; you may use any reasonable identifiers you choose here.

The DLL initialization function always has three parameters. The second parameter is the only one of real interest to the DLL initialization code. This parameter contains the reason for calling this code, which is one of the following constants defined in the w.hhf header file:

- w.DLL_PROCESS_ATTACH
- w.DLL_PROCESS_DETACH
- w.DLL_THREAD_ATTACH
- w.DLL_THREAD_DETACH

The w.DLL_XXXXX_ATTACH values indicate that some program is linking in the DLL. During these calls, you should open any files, initialize any variables, and execute any other initialization code that may be necessary for the proper operation of the DLL. Note that, by default, all processes that attach to a DLL get their own copy of any data defined in the DLL. Therefore, you do not have to worry about disturbing previous links to the DLL during the current initialization process.

The w.DLL_XXXXX_DETACH values indicate that a process or thread is shutting down. During these calls, you should close any files and perform any other necessary cleanup (e.g., freeing memory) that you would normally do before a program ends.

The following code demonstrates a short DLL:

```
unit dllExample;
#include( "w.hhf" );

static
    ThisInstance: dword;

procedure dll( instance:dword; reason:dword; reserved:dword );
        @stdcall; @external( "_dll@12" );

procedure dllFunc1( dw:dword ); @stdcall; @external( "_dllFunc1@4" );
procedure dllFunc2( dw2:dword ); @stdcall; @external( "_dllFunc2@4" );



procedure dll( instance:dword; reason:dword; reserved:dword ); @nodisplay;
begin dll;

    // Save the instance value.

    mov( instance, eax );
    mov( eax, ThisInstance );

    if( reason = w.DLL_PROCESS_ATTACH ) then

        // Do this code if we're attaching this DLL to a process...

    endif;

    // Return true if successful, false if unsuccessful.

    mov( true, eax );

end dll;


procedure dllFunc1( dw:dword ); @nodisplay;
begin dllFunc1;

    mov( dw, eax );

end dllFunc1;


procedure dllFunc2( dw2:dword ); @nodisplay;
begin dllFunc2;

    push( edx );
    mov( dw2, eax );
    mul( dw2, eax );
    pop( edx );

end dllFunc2;

end dllExample;
```

As you can see here, there is very little difference between a standard unit and an HLA unit intended to become a DLL. The name mangling is one difference, placing the external declarations directly in the file (rather than in an include file) is another difference. The only functional difference is the presence of the DLL initialization procedure ("dll" in this example).

The real work in creating a DLL occurs during the link phase. You cannot compile a DLL the same way you compile a standard HLA program - some additional steps are necessary. Creating a DLL requires lots of command line parameters, so it is best to create a makefile and a "linker" file to avoid excess typing at the command line. Consider the following make file for the module above:

```
dll.dll: dll.obj
    link dll.obj @dll.linkresp

dll.obj: dll.hla
    hla -@ -c dll.hla
```

This makefile generates the dll.dll file (it will also produce several other files, dll.lib being the most important one). The real work appears in the "dll.linkresp" linker file. This file contains the following text:

```
-DLL
-entry:dll
-base:0x40000000
-out:dll.dll
-export:dll
-export:dllFunc1
-export:dllFunc2
```

The “-DLL” option tells the linker to produce a “dll.dll” and a “dll.lib” file rather than just a “dll.exe” file (note: the linker will also produce some other files, but these two are the ones important to us).

The “-entry:dll” option tells the linker that the name of the DLL initialization code is the procedure “dll”. If you change the name of your DLL initialization code, you should also change this option.

The “-base:0x40000000” option tells the linker that this DLL has a base address of 1GByte. For efficiency reasons, you should try to specify a unique value here. If two active DLLs specify the same base address, different processes cannot concurrently share the two DLLs. The programs will still operate, but they will not share the code, wasting some memory and requiring longer load times.

The “-out:dll.dll” command specifies the output name for the DLL. The suffix should be “.dll” and the base filename should be an appropriate name for your DLL (“dll” was appropriate in this case, it would not be appropriate in other cases).

The “-export” options specify the names of the external procedures you wish to make available to other modules. Alternately, you may create a “.DEF” file and use the “-DEF:deffilename.def” option to pass the exported file names on to the linker (see the Microsoft documentation for a description of DEF files).

If you run this make file, it will compile the dll.hla source file producing the dll.dll and dll.lib object modules.

18.1.2 Linking and Calling Procedures in a Dynamic Link Library

Creating a DLL in HLA is only half the battle. The other half is calling a procedure in a DLL from an HLA program. Here is a sample program that calls the DLL procedures in the previous section:

```
// Sample program that calls routines in dll.dll.
//
//  Compile this with the command line option:
//
//      hla dllmain dll.lib
//
//  Of course, you must build the DLL first.

program callDLL;
#include( "stdlib.hhf" );


procedure dllFunc1( dw:dword ); @stdcall; @external( "_dllFunc1@4" );
procedure dllFunc2( dw:dword ); @stdcall; @external( "_dllFunc2@4" );


begin callDLL;

    xor( eax, eax );
    dllFunc1( 12345 );
    stdout.put( "After dllFunc1, eax = ", (type uns32 eax ), nl );
    dllFunc2( 100 );
    stdout.put( "After dllFunc2, eax = ", (type uns32 eax ), nl );


end callDLL;
```

`To compile this main program, you would use the following HLA command line:`

```
hla dllmain dll.lib
```

`The “dll.lib” file contains the linkages necessary to load and link in the dll module at run-time.`

18.1.3 Going Farther

This document only explains “implicitly loaded” DLLs. Implicitly loaded DLLs are always loaded into memory when the main module loads into memory. If you want to control the loading of the DLL module into memory, you will want to take a look at “explicitly loaded” DLLs. Such DLLs, however, will have to be the subject of a different example.

18.2 Compiling HLA

Source code has been shipped with the HLA releases since HLA v1.18. However, until Bison 1.875 became available, compiling HLA required a special, hacked, version of Bison that ran under Linux. This made development of HLA (particularly under Linux) a bit painful. Fortunately, as of Bison 1.875, it is now possible to compile the HLA source code using standard versions of Flex and Bison available from the Free Software Foundation (the GNU folks). You must, however, have Bison 1.875 or later to successfully translate the HLAPARSE.BSN file. Under Windows, the CYGWIN package containing Flex and Bison works great. I (Randy Hyde) have never actually built HLAPARSE.BSN under Linux, so I don't have any experience with this process. It may be trivial, it may be impossible. I've never tried it. I always generate HLAPARSE.C under Windows using Bison and then I copy the C file over to Linux for compilation there.

First, a couple of comments about the source code: HLA v1.x and v2.x are prototype systems. This means that there are massive kludges in the code. The whole system evolved over time rather than being designed properly in the first place (no apologies for this, that's the whole purpose of a prototype). So if you looking for wonderfully structured code that's easy to follow, HLA will disappoint you. I learned quite a bit about FLEX and BISON while writing HLA and, unfortunately, it shows. There are many ways I've done things that someone who was more familiar with FLEX/Bison would have done differently (heck, there are a lot of things I would do differently, in hindsight). None of this is worth fixing since such work is better put to writing v2.x of HLA.

The HLA source code is almost 200,000 lines long. The Bison file alone is about 100,000 lines of code. Messing with HLA source code is not an undertaking for the weak of heart. Although much of the code is commented, there is very little "high level documentation" (i.e., design documentation) available that would explain why I've done certain things or to provide the general philosophy behind the code. I offer the source code in this form; it is up to you to decide whether you want to spend the time needed to figure it all out.

One note about support: I will be more than happy to answer questions about HLA in the Yahoo AoA/HLA Programming newsgroup. However, I do not have time to answer individual questions asked via email concerning the source code. I apologize ahead of time, but releasing a program of this magnitude to the public could wind up burying me with questions. Because of the possible volume of emails this product could produce, I must ignore all requests for help that arrive via email. Of course, bug reports are always welcome via email. Send everything else to one of the two aforementioned newsgroups.

I have developed HLA with the following tools:

- CodeWright Editor (it takes a decent editor to handle files in excess of 100,000 lines of code).
- Microsoft Visual C++ (v9)
- Flex
- Bison (must be 1.875 or later)
- Microsoft nmake
- GCC 2.9x (Linux, FreeBSD, and Mac OS X versions)
- HLA (a couple of modules are written in HLA itself).
- MASM v9.
- Gas (Linux, FreeBSD, Mac OSX)

I have supplied a makefile that should automatically build the HLA system for you. See the makefile in the main source directory for details. For Linux, there is a "makefile.linux" file that you should use. For FreeBSD use "makefile.freebsd" and for the Macintosh, use "makefile.mac",

HLA is probably not portable. I have made no attempt to ensure that the code compiles with anything other than GCC and MSVC++, so undoubtedly it won't compile on anything else without some effort. I have eliminated *most* of the compiler warnings, so porting to some other compilers shouldn't be too difficult.

Porting HLA to generate assembly code for an assembler other than MASM, NASM, FASM, Gas,or TASM is a *major* undertaking. TASM took a couple of weeks to pull off and TASM is mostly compatible with MASM. Gas took about a month of evenings and FASM took several weekends. Fortunately, if you choose to do this, I've made the process easier and easier with each new back-end assembler I added. Porting HLA to generate object code other than PE/COFF, ELF, or Mach-o is a *serious* undertaking. I spent a couple of months on each of the three object formats that HLA currently supports.

Porting to other operating systems (other than Windows, Mac OSX, FreeBSD and Linux) is certainly possible. The compiler should be fairly easy to port. The real work is in porting the Standard Library. I've looked into porting HLA to QNX, but haven't pursued this for a couple of reasons: (1) QNX's version of GCC is older and has problems compiling the source code, (2) QNX doesn't really support assembly level calls to the OS so I'd have to port the HLA standard library on top of the C standard library code (which is ugly). NetBSD and OpenBSD should be easy - just a simple modification of the FreeBSD port. At one time I looked into a BeOS port, but then BeOS died, so I gave up. Solaris/Sun OS is a possibility, but now that Oracle has bought out Sun, who knows where that OS is going?

18.3 Code Generation for HLA HLL Control Structures

Note: *This is a very old and incomplete document. It was written back in the early days of HLA v1.x. While the general principles still apply, the specific examples of code generated by HLA have changed quite a bit. Nevertheless, the information is still useful to some people so I've included this document here. If there is sufficient interest, I can be convinced to update and finish this document.*

One of the principal advantages of using assembly language over high level languages is the control that assembly provides. High level languages (HLLs) represent an abstraction of the underlying hardware. Those who write HLL code give up this control in exchange for the engineering efficiencies enjoyed by HLL programmers. Some advanced HLL programmers (who have a good mastery of the underlying machine architecture) are capable of writing fairly efficient programs by recognizing what the compiler does with various high level control constructs and choosing the appropriate construct to emit the machine code they want. While this "low-level programming in a high level language" does leave the programmer at the mercy of the compiler-writer, it does provide a mechanism whereby HLL programmers can write more efficient code by chosing those HLL constructs that compile into efficient machine code.

Although the High Level Assembler (HLA) allows a programmer to work at a very low level, HLA also provides structured high-level control constructs that let assembly programmers use higher-level code to help make their assembly code more readable. Those assembly language programmers who need (or want) to exercise maximum control over their programs will probably want to avoid using these statements since they tend to obscure what is happening at a really low level. At the other extreme, those who would always use these high-level control structures might question if they really want to use assembly language in their applications; after all, if they're writing high level code, perhaps they should use a high level language and take advantage of optimizing technology and other fancy features found in modern compilers. Between these two extremes lies the typical assembly language programmer. The one who realizes that most code doesn't need to be super-efficient and is more interested in productively producing lots of software rather than worrying about how many CPU cycles the one-time initialization code is going to consume. HLA is perfect for this type of programmer because it lets you work at a high level of abstraction when writing code whose performance isn't an issue and it lets you work at a low level of abstraction when working on code that requires special attention.

Between code whose performance doesn't matter and code whose performance is critical lies a big gray region: code that should be reasonably fast but speed isn't the number one priority. Such code needs to be reasonably readable, maintainable, and as free of defects as possible. In other words, code that is a good candidate for using high level control and data structures if their use is reasonably efficient.

Unlike various HLL compilers, HLA does not (yet!) attempt to optimize the code that you write. This puts HLA at a disadvantage: it relies on the optimizer between your ears rather than the one supplied with the compiler. If you write sloppy high level code in HLA then a HLL version of the same program will probably be more efficient if it is compiled with a decent HLL compiler. For code where performance matters, this can be a disturbing revelation (you took the time and bother to write the code in assembly but an equivalent C/C++ program is faster). The purpose of this article is to describe HLA's code generation in detail so you can intelligently choose when to use HLA's high level features and when you should stick with low-level assembly language.

18.3.1 The HLA Standard Library

The HLA Standard Library was designed to make learning assembly language programming easy for beginning programmers. Although the code in the library isn't terrible, very little effort was made to write top-performing code in the library. At some point in the future this may change as work on the library progresses, but if you're looking to write very high-performance code you should probably avoid calling routines in the HLA Standard Library from (speed) critical sections of your program.

Don't get the impression from the previous paragraph that HLA's Standard Library contains a bunch of slow-poke routines, however. Many of the HLA Standard Library routines use decent algorithms and data structures so they perform quite well in typical situations. For example, the HLA string format is far more efficient than strings in C/C++. The world's best C/C++ strlen routine is almost always going to be slower than HLA str.len function. This is because HLA uses a better definition for string data than C/C++, it has little to do with the actual implementation of the str.len code. This is not to say that HLA's str.len routine cannot be improved; but the routine is very fast already.

One problem with using the HLA Standard Library is the frame of mind it fosters during the development of a program. The HLA Standard Library is strongly influenced by the C/C++ Standard Library and libraries common in other high level languages. While the HLA Standard Library is a wonderful tool that can help you write assembly code faster than ever before, it also encourages you to think at a higher level. As any expert assembly language programmer can tell you, the real benefits of using assembly language occur only when you "think in assembly" rather than in a high level language. No matter how efficient the routines in the Standard Library happen to be, if you're "writing C++ programs with MOV instructions" the result is going to be little better than writing the code in C++ to begin with.

One unfortunate aspect of the HLA Standard Library is that it encourages you to think at a higher level and you'll often miss a far more efficient low-level solution as a result. A good example is the set of string routines in the HLA Standard Library. If you use those routines, even if they were written as efficiently as possible, you may not be writing the fastest possible program you can because you've limited your thinking to string objects which are a higher level abstraction. If you did not have the HLA Standard Library laying around and you had to do all the character string manipulation yourself, you might choose to treat the objects as character arrays in memory. This change of perspective can produce dramatic performance improvement under certain circumstances.

The bottom line is this: the HLA Standard Library is a wonderful collection of routines and they're not particularly inefficient. They're very easy and convenient to use. However, don't let the HLA Standard Library lull you into choosing data structures or algorithms that are not the most appropriate for a given section of your program.

18.3.2 Compiling to MASM Code -- The Final Word

The remainder of this document will discuss, in general, how HLA translates various HLL-style statements into assembly code. Sometimes a general discussion may not provide specific answers you need about HLA's code generation capabilities. Should you have a specific question about how HLA generates code with respect to a given code sequence, you can always run the compiler and observe the output it produces. To do this, it is best to create a simple program that contains only the construct you wish to study and compile that program to assembly code. For example, consider the following very simple HLA program:

```
program t;

begin t;

    if( eax = 0 ) then

        mov( 1, eax );

    endif;

end t;
```

If you compile this program using the command window prompt "hla -s t.hla" then HLA produces the following (MASM) assembly code output (in the "t.asm" output file)[1]:

```
                if      @Version lt 612
                .586
                else
                .686
                .mmx
                .xmm
                endif
                .model  flat, syscall
```

1. This code is from an older version of HLA. The actual code HLA generates today is different. But the concepts this document covers still apply.

```
offset32        equ     <offset flat:>
                assume  fs:nothing
?ExceptionPtr   equu    <(dword ptr fs:[0])>
                externdef ??HWexcept:near32
                externdef ??Raise:near32

std_output_hndl equ     -11

                externdef __imp__ExitProcess@4:dword
                externdef __imp__GetStdHandle@4:dword
                externdef __imp__WriteFile@20:dword

cseg            segment page public 'code'
cseg            ends
readonly        segment page public 'data'
readonly        ends
strings         segment page public 'data'
strings         ends
dseg            segment page public 'data'
dseg            ends
bssseg          segment page public 'data'
bssseg          ends


strings         segment page public 'data'

?dfltmsg        byte    "Unhandled exception error.",13,10
?dfltmsgsize    equ     34
?absmsg         byte    "Attempted call of abstract procedure or
method.",13,10
?absmsgsize     equ     55
strings         ends
dseg            segment page public 'data'
?dfmwritten     word    0
?dfmStdOut      dword   0

                public  ?MainPgmCoroutine
?MainPgmCoroutine byte 0 dup (?)
                dword   ?MainPgmVMT
                dword   0       ;CurrentSP
                dword   0       ;Pointer to stack
                dword   0       ;ExceptionContext
                dword   0       ;Pointer to last caller
?MainPgmVMT     dword   ?QuitMain
dseg            ends
cseg            segment page public 'code'

?QuitMain       proc    near32
                pushd   1
                call    dword ptr __imp__ExitProcess@4

?QuitMain       endp

cseg            ends


cseg            segment page public 'code'
```

```
??DfltExHndlr   proc    near32

                pushd   std_output_hndl
                call    __imp__GetStdHandle@4
                mov     ?dfmStdOut, eax
                pushd   0       ;lpOverlapped
                pushd   offset32 ?dfmwritten    ;BytesWritten
                pushd   ?dfltmsgsize    ;nNumberOfBytesToWrite
                pushd   offset32 ?dfltmsg       ;lpBuffer
                pushd   ?dfmStdOut              ;hFile
                call    __imp__WriteFile@20

                pushd   0
                call    dword ptr __imp__ExitProcess@4

??DfltExHndlr   endp

                public  ??Raise
??Raise proc    near32
                jmp     ??DfltExHndlr
??Raise endp

                public  ??HWexcept
??HWexcept      proc    near32
                mov     eax, 1
                ret
??HWexcept      endp

?abstract       proc    near32

                pushd   std_output_hndl
                call    __imp__GetStdHandle@4
                mov     ?dfmStdOut, eax
                pushd   0       ;lpOverlapped
                pushd   offset32 ?dfmwritten    ;BytesWritten
                pushd   ?absmsgsize     ;nNumberOfBytesToWrite
                pushd   offset32 ?absmsg        ;lpBuffer
                pushd   ?dfmStdOut              ;hFile
                call    __imp__WriteFile@20

                pushd   0
                call    dword ptr __imp__ExitProcess@4

?abstract       endp


                public  ?HLAMain
?HLAMain        proc    near32


; Set up the Structured Exception Handler record
; for this program.

                push    offset32 ??DfltExHndlr
                push    ebp
                push    offset32 ?MainPgmCoroutine
                push    offset32 ??HWexcept
                push    ?ExceptionPtr
                mov     ?ExceptionPtr, esp
                mov     dword ptr ?MainPgmCoroutine+12, esp

                pushd   0                ;No Dynamic Link.
                mov     ebp, esp         ;Pointer to Main's locals
                push    ebp              ;Main's display.
                mov     [ebp+16], esp
                cmp     eax, 0
                jne     ?1_false
                mov     eax, 1
?1_false:
                push    0
                call    dword ptr __imp__ExitProcess@4
?HLAMain        endp
cseg            ends
                end
```

The code of interest in this example is at the very end, after the comment ";Main's display" appears in the text. The actual code sequence that corresponds to the IF statement in the main program is the following:

```
                cmp     eax, 0
                jne     ?1_false
                mov     eax, 1
?1_false:
```

Note: you can verify that this is the code emitted by the IF statement by simply removing the IF, recompiling, and comparing the two assembly outputs. You'll find that the only difference between the two assembly output files is the four lines above. Another way to "prove" that this is the code sequence emitted by the HLA IF statement is to insert some comments into the assembly output file using HLA's #ASM..#ENDASM directives. Consider the following modification to the "t.hla" source file:

```
program t;

begin t;

    #asm
    ; Start of IF statement:
    #endasm

    if( eax = 0 ) then

        mov( 1, eax );

    endif;

    #asm
    ; End if IF statement.
    #endasm

end t;
```

HLA’s #asm directive tells the compiler to simply emit everything between the #asm and #endasm keywords directly to the assembly output file. In this example the HLA program uses these directives to emit a pair of comments that will bracket the code of interest in the output file. Compiling this to assembly code (and stripping out the irrelevant stuff before the HLA main program) yields the following:

```
                public  ?HLAMain
?HLAMain        proc    near32


; Set up the Structured Exception Handler record
; for this program.

                push    offset32 ??DfltExHndlr
                push    ebp
                push    offset32 ?MainPgmCoroutine
                push    offset32 ??HWexcept
                push    ?ExceptionPtr
                mov     ?ExceptionPtr, esp
                mov     dword ptr ?MainPgmCoroutine+12, esp

                pushd   0               ;No Dynamic Link.
                mov     ebp, esp        ;Pointer to Main's locals
                push    ebp             ;Main's display.
                mov     [ebp+16], esp

;#asm


        ; Start of IF statement:
        ;#endasm

                cmp     eax, 0
                jne     ?1_false
                mov     eax, 1
?1_false:

;#asm


        ; End if IF statement.
        ;#endasm

                push    0
                call    dword ptr __imp__ExitProcess@4
?HLAMain        endp
cseg            ends
                end
```

This technique (embedding bracketing comments into the assembly output file) is very useful if it is not possible to isolate a specific statement in its own source file when you want to see what HLA does during compilation.

18.3.3 The HLA if..then..endif Statement, Part I

Although the HLA IF statement is actually one of the more complex statements the compiler has to deal with (in terms of how it generates code), the IF statement is probably the first statement that comes to mind when something thinks about high level control structures. Furthermore, you can implement most of the other control structures if you have an IF and a GOTO (JMP) statement, so it makes sense to discuss the IF statement first. Nevertheless, there is a bit of complexity that is unnecessary at this point, so we'll begin our discussion with a simplified version of the IF statement; for this simplified version we'll not consider the ELSEIF and ELSE clauses of the IF statement.

The basic HLA IF statement uses the following syntax:

```
if( simple_boolean_expression ) then

    << statements to execute if the expression evaluates true >>

endif;
```

At the machine language level, what the compiler needs to generate is code that does the following:

```
<< Evaluate the boolean expression >>

<< Jump around the following statements if the expression was false >>

<< statements to execute if the expression evaluates true >>

<< Jump to this point if the expression was false >>
```

The example in the previous section is a good demonstration of what HLA does with a simple IF statement. As a reminder, the HLA program contained

```
    if( eax = 0 ) then

        mov( 1, eax );

    endif;
```

and the HLA compiler generated the following assembly language code:

```
                cmp     eax, 0
                jne     ?1_false
                mov     eax, 1
?1_false:
```

Evaluation of the boolean expression was accomplished with the single "cmp eax, 0" instruction. The "jne ?1_false" instruction jumps around the "mov eax, 1" instruction (which is the statement to execute if the expression evaluates true) if the expression evaluates false. Conversely, if EAX is equal to zero, then the code falls through to the MOV instruction. Hence the semantics are exactly what we want for this high level control structure.

HLA automatically generates a unique label to branch to for each IF statement. It does this properly even if you nest IF statements. Consider the following code:

```
program t;
```

```
begin t;

    if( eax > 0 ) then

        if( eax < 10 ) then

            inc( eax );

        endif;

    endif;


end t;
```

The code above generates the following assembly output:

```
                cmp     eax, 0
                jna     ?1_false
                cmp     eax, 10
                jnb     ?2_false
                inc     eax
?2_false:
?1_false:
```

As you can tell by studying this code, the INC instruction only executes if the value in EAX is greater than zero and less than ten.

Thus far, you can see that HLA's code generation isn't too bad. The code it generates for the two examples above is roughly what a good assembly language programmer would write for approximately the same semantics.

18.3.4 Boolean Expressions in HLA Control Structures

The HLA IF statement and, indeed, most of the HLA control structures rely upon the evaluation of a boolean expression in order to direct the flow of the program. Unlike high level languages, HLA restricts boolean expressions in control structures to some very simple forms. This was done for two reasons: (1) HLA's design frowns upon side effects like register modification in the compiled code, and (2) HLA is intended for use by beginning assembly language students; the restricted boolean expression model is closer to the low level machine architecture and it forces them to start thinking in these terms right away.

With just a few exceptions, HLA's boolean expressions are limited to what HLA can easily compile to a CMP and a condition jump instruction pair or some other simple instruction sequence. Specifically, HLA allows the following boolean expressions:

```
operand1 relop operand2

relop is one of:

=  or ==        (either one, both are equivalent)
<> or !=        (either one, both are equivalent)
<
<=
>
```

```
>=

A CPU flag specification.
A CPU register.
A boolean or byte variable.
```

In the expressions above operand$_1$ and operand$_2$ are restricted to those operands that are legal in a CMP instruction. This is because HLA translates expressions of this form to the two instruction sequence:

```
cmp( operand1, operand2 );
jXX someLabel;
```

where “jXX” represents some condition jump whose sense is the opposite of that of the expression (e.g., “eax > ebx” generates a “JNA” instruction since “NA” is the opposite of “>”).

Assuming you want to compare the two operands and jump around some sequence of instructions if the relationship does not hold, HLA will generate fairly efficient code for this type of expression. One thing you should watch out for, though, is that HLA’s high level statements (e.g., IF) make it very easy to write code like the following:

```
if( i = 0 ) then

   ...

elseif( i = 1 ) then

    ...

elseif( i = 2 ) then

    ...
.
.
.
endif;
```

This code looks fairly innocuous, but the programmer who is aware of the fact that HLA emits the following would probably not use the code above:

```
     cmp( i, 0 );
     jne lbl;
       .
       .
       .
lbl: cmp( i, 1 );
     jne lbl2;
       .
       .
       .
lbl2: cmp( i, 2 );
       .
       .
       .
```

A good assembly language programmer would realize that it’s much better to load the variable “i” into a register and compare the register in the chain of CMP instructions rather than compare the variable each time. The high level syntax slightly obscures this problem; just one thing to be aware of.

HLA's boolean expressions do not support conjunction (logical AND) and disjunction (logical OR). The HLA programmer must manually synthesize expressions involving these operators. Doing so forces the programmer to link in lower level terms, which is usually more efficient. However, there are many common expressions involving conjunction that HLA could efficiently compile into assembly language. Perhaps the most common example is a test to see if an operand is within (or outside) a range specified by two constants. In a HLL like C/C++ you would typically use an expression like "(value >= low_constant && value <= high_constant)" to test this condition. HLA allows four special boolean expressions that check to see if a register or a memory location is within a specified range. The allowable expressions take the following forms:

```
register in constant .. constant
register not in constant .. constant

memory in constant .. constant
memory not in constant .. constant
```

Here is a simple example of the first form with the code that HLA generates for the expression:

```
    if( eax in 1..10 ) then

        mov( 1, ebx );

    endif;
```

Resulting (MASM) assembly code:

```
                cmp     eax, 1
                jb      ?1_false
                cmp     eax, 10
                ja      ?1_false
                mov     ebx, 1
?1_false:
```

Once again, you can see that HLA generates reasonable assembly code without modifying any register values. Note that if modifying the EAX register is okay, you can write slightly better code by using the following sequence:

```
                dec     eax
                cmp     eax, 9
                ja      ?1_false
                mov     ebx, 1
?1_false:
```

While, in general, a simplification like this is not possible you should always remember how HLA generates code for the range comparisons and decide if it is appropriate for the situation.

By the way, the "not in" form of the range comparison does generate slightly different code that the form above. Consider the following:

```
    if( eax not in 1..10 ) then

        mov( 1, eax );
```

```
endif;
```

HLA generates the following (MASM) assembly language code for the sequence above:

```
                cmp     eax, 1
                jb      ?2_true
                cmp     eax, 10
                jna     ?1_false
?2_true:
                mov     eax, 1
?1_false:
```

As you can see, though the code is slightly different it is still exactly what you would probably write if you were writing the low level code yourself.

HLA also allows a limited form of the boolean expression that checks to see if a character value in an eight-bit register is a member of a character set constant or variable. These expressions use the following general syntax:

reg_8 in *CSet_Constant*

reg_8 in *CSet_Variable*

reg_8 not in *CSet_Constant*

reg_8 not in *CSet_Variable*

These forms were included in HLA because they are so similar to the range comparison syntax. However, the code they generate may not be particularly efficient so you should avoid using these expression forms if code speed and size need to be optimal. Consider the following:

```
if( al in {'A'..'Z','a'..'z', '0'..'9'} ) then

    mov( 1, eax );

endif;
```

This generates the following (MASM) assembly code:

```
strings         segment page public 'data'
?1_cset          byte 00h,00h,00h,00h,00h,00h,0ffh,03h
                 byte 0feh,0ffh,0ffh,07h,0feh,0ffh,0ffh,07h
strings         ends

                push    eax
                movzx   eax, al
                bt      dword ptr ?1_cset, eax
                pop     eax
                jnc     ?1_false
                mov     eax, 1
?1_false:
```

This code is rather lengthy because HLA never assumes that it can disturb the values in the CPU registers. So right off the bat this code has to push and pop EAX since it disturbs the value in EAX. Next, HLA doesn't assume that the upper three bytes of EAX already contain zero, so it zero fills them. Finally, as you can see above, HLA has to create a 16-byte character set in memory in order to test the value in the AL register. While this is convenient, HLA does generate a lot of code and data for such a simple looking expression. Hence, you should be careful about using boolean expressions involving character sets if speed and space is important. At the very least, you could probably reduce the code above to something like:

```
                movzx( charToTest, eax );
                bt( eax, {'A'..'Z','a'..'z', '0'..'9'});
                jnc SkipMov;
                mov(1, eax );
SkipMov:
```

This generates code like the following:

```
strings         segment page public 'data'
?cset_3          byte   00h,00h,00h,00h,00h,00h,0ffh,03h
                 byte   0feh,0ffh,0ffh,07h,0feh,0ffh,0ffh,07h
strings         ends

                movzx   eax, byte ptr ?1_charToTest[0]   ;charToTest
                bt      dword ptr ?cset_3, eax
                jnc     ?4_SkipMov
                mov     eax, 1

?4_SkipMov:
```

As you can see, this is slightly more efficient. Fortunately, testing an eight-bit register to see if it is within some character set (other than a simple range, which the previous syntax handles quite well) is a fairly rare operation, so you generally don't have to worry about the code HLA generates for this type of boolean expression.

HLA lets you specify a register name or a memory location as the only operand of a boolean expression. For registers, HLA will use the TEST instruction to see if the register is zero or non-zero. For memory locations, HLA will use the CMP instruction to compare the memory location's value against zero. In either case, HLA will emit a JNE or JE instruction to branch around the code to skip (e.g., in an IF statement) if the result is zero or non-zero (depending on the form of the expression).

```
register
!register

memory
!memory
```

You should not use this trick as an efficient way to test for zero or not zero in your code. The resulting code is very confusing and difficult to follow. If a register or memory location appears as the sole operand of a boolean expression, that register or memory location should hold a boolean value (true or false). Do not think that "if(eax) then..." is any more efficient than "if(eax<>0) then..." because HLA will actually emit the same exact code for both statements (i.e., a TEST instruction). The second is a lot easier to understand if you're really checking to see if EAX is not zero (rather than it contains the boolean value true), hence it is always preferable even if it involves a little extra typing.

Example:

```
    if( eax != 0 ) then

        mov( 1, ebx );
    endif;

    if( eax ) then

        mov( 2, ebx );

    endif;
```

The code above generates the following assembly instruction sequence:

```
                test    eax,eax ;Test for zero/false.
                je      ?2_false
                mov     ebx, 1
?2_false:
                test    eax,eax ;Test for zero/false.
                je      ?3_false
                mov     ebx, 2
?3_false:
```

Note that the pertinent code for both sequences is identical. Hence there is never a reason to sacrifice readability for efficiency in this particular case.

The last form of boolean expression that HLA allows is a flag designation. HLA uses symbols like @c, @nc, @z, and @nz to denote the use of one of the flag settings in the CPU FLAGS register. HLA supports the use of the following flag names in a boolean expression:

```
@c, @nc, @o, @no, @z, @nz, @s, @ns, @a, @na, @ae, @nae, @b, @nb, @be,
@nbe, @l, @nl, @g, @ne, @le, @nle, @ge, @nge, @e, @ne
```

Whenever HLA encounters a flag name in a boolean expression, it efficiently compiles the expression into a single conditional jump instruction. So the following IF statement's expression compiles to a single instruction:

```
if( @c ) then

    << do this if the carry flag is set >>

endif;
```

The above code is completely equivalent to the sequence:

```
    jnc SkipStmts;

    << do this if the carry flag is set >>

SkipStmts:
```

The former version, however, is more readable so you should use the IF form wherever practical.

18.3.5 The JT/JF Pseudo-Instructions

The JT (jump if true) and JF (jump if false) pseudo-instructions take a boolean expression and a label. These instructions compile into a conditional jump instruction (or sequence of instructions) that jump to the target label if the specified boolean expression evaluates false. The compilation of these two statements is almost exactly as described for boolean expressions in the previous section. The principle difference is that HLA sneaks in a (MASM) macro declaration because of technical issues involving code generation. Other than this one minor issue in the MASM source code, the code generation is exactly as described above.

The following are a couple of examples that show the usage and code generation for these two statements.

```
lbl2:
    jt( eax > 10 ) label;
label:
    jf( ebx = 10 ) lbl2;


; Translated Code:

?2_lbl2:
?3_BoolExpr     macro   target
                cmp     eax, 10
                ja      target
                endm
                ?3_BoolExpr     ?4_label

?4_label:
?5_BoolExpr     macro   target
                cmp     ebx, 10
                jne     target
                endm
                ?5_BoolExpr     ?2_lbl2
```

18.3.6 The HLA if..then..elseif..else..endif Statement, Part II

With the discussion of boolean expressions out of the way, we can return to the discussion of the HLA IF statement and expand on the material presented earlier. There are two main topics to consider: the inclusion of the ELSEIF and ELSE clauses and the HLA hybrid IF statement. This section will discuss these additions.

The ELSE clause is the easiest option to describe, so we'll start there. Consider the following short HLA code fragment:

```
    if( eax < 10 ) then

        mov( 1, ebx );

    else

        mov( 0, ebx );

    endif;
```

HLA's code generation algorithm emits a JMP instruction upon encountering the ELSE clause; this JMP transfers control to the first statement following the ENDIF clause. The other difference between the IF/ELSE/ENDIF and the IF/ENDIF statement is the fact that a false expression evaluation transfers control to the ELSE clause rather than to the first statement following the ENDIF. When HLA compiles the code above, it generates machine code like the following:

```
                cmp     eax, 10
                jnb     ?2_false   ;Branch to ELSE section if false

                mov     ebx, 1
                jmp     ?2_endif   ;Skip over ELSE section

; This is the else section:

?2_false:
                mov     ebx, 0
?2_endif:
```

About the only way you can improve upon HLA's code generation sequence for an IF/ELSE statement is with knowledge of how the program will operate. In some rare cases you can generate slightly better performing code by moving the ELSE section somewhere else in the program and letting the THEN section fall straight through to the statement following the ENDIF (of course, the ELSE section must jump back to the first statement after the ENDIF if you do this). This scheme will be slightly faster if the boolean expression evaluates true most of the time. Generally, though, this technique is a bit extreme.

The ELSEIF clause, just as its name suggests, has many of the attributes of an ELSE and and IF clause in the IF statement. Like the ELSE clause, the IF statement will jump to an ELSEIF clause (or the previous ELSEIF clause will jump to the current ELSEIF clause) if the previous boolean expression evaluates false. Like the IF clause, the ELSEIF clause will evaluate a boolean expression and transfer control to the following ELSEIF, ELSE, or ENDIF clause if the expression evaluates false; the code falls through to the THEN section of the ELSEIF clause if the expression evaluates true. The following examples demonstrate how HLA generates code for various forms of the IF..ELSEIF.. statement:

Single ELSEIF clause:

```
    if( eax < 10 ) then

        mov( 1, ebx );

    elseif( eax > 10 ) then

        mov( 0, ebx );

    endif;


; Translated code:

                cmp     eax, 10
                jnb     ?2_false
                mov     ebx, 1
                jmp     ?2_endif
?2_false:
                cmp     eax, 10
                jna     ?3_false
                mov     ebx, 0
?3_false:
?2_endif:
```

Single ELSEIF clause with an ELSE clause:

```
    if( eax < 10 ) then

        mov( 1, ebx );

    elseif( eax > 10 ) then

        mov( 0, ebx );

    else

        mov( 2, ebx );

    endif;


; Converted code:

                cmp     eax, 10
                jnb     ?2_false
                mov     ebx, 1
                jmp     ?2_endif
?2_false:
                cmp     eax, 10
                jna     ?3_false
                mov     ebx, 0
                jmp     ?2_endif
?3_false:
                mov     ebx, 2
?2_endif:
```

IF statement with two ELSEIF clauses:

```
    if( eax < 10 ) then

        mov( 1, ebx );

    elseif( eax > 10 ) then

        mov( 0, ebx );

    elseif( eax = 5 ) then

        mov( 2, ebx );

    endif;

; Translated code:

                cmp     eax, 10
                jnb     ?2_false
                mov     ebx, 1
                jmp     ?2_endif
?2_false:
                cmp     eax, 10
                jna     ?3_false
                mov     ebx, 0
                jmp     ?2_endif
?3_false:
                mov     ebx, 2
?2_endif:
```

IF statement with two ELSEIF clauses and an ELSE clause:

```
    if( eax < 10 ) then

        mov( 1, ebx );

    elseif( eax > 10 ) then

        mov( 0, ebx );

    elseif( eax = 5 ) then

        mov( 2, ebx );

    else

        mov( 3, ebx );

    endif;

; Translated code:

                cmp     eax, 10
                jnb     ?2_false
                mov     ebx, 1
                jmp     ?2_endif
?2_false:
                cmp     eax, 10
                jna     ?3_false
                mov     ebx, 0
                jmp     ?2_endif
?3_false:
                cmp     eax, 5
                jne     ?4_false
                mov     ebx, 2
                jmp     ?2_endif
?4_false:
                mov     ebx, 3
?2_endif:
```

This code generation algorithm generalizes to any number of ELSEIF clauses. If you need to see an example of an IF statement with more than two ELSEIF clauses, feel free to run a short example through the HLA compiler to see the result.

In addition to processing boolean expressions, the HLA IF statement supports a hybrid syntax that lets you combine the structured nature of the IF statement with the unstructured nature of typical assembly language control flow. The hybrid form gives you almost complete control over the code generation process without completely sacrificing the readability of an IF statement. The following is a typical example of this form of the IF statement:

```
    if
    {
        cmp( eax, 10 );
        jna false;
    }

        mov( 0, eax );

    endif;


; The above generates the following assembly code:

                cmp     eax, 10
                jna     ?2_false
?2_true:
                mov     eax, 0
?2_false:
```

Of course, the hybrid IF statement fully supports ELSE and ELSEIF clauses (in fact, the IF and ELSEIF clauses can have a potpourri of hybrid or traditional boolean expression forms). The hybrid forms, since they let you specify the sequence of instructions to compile, put the issue of efficiency squarely in your lap. About the only contribution that HLA makes to the inefficiency of the program is the insertion of a JMP instruction to skip over ELSEIF and ELSE clauses.

Although the hybrid form of the IF statement lets you write very efficient code that is more readable than the traditional “compare and jump” sequence, you should keep in mind that the hybrid form is definitely more difficult to read and comprehend than the IF statement with boolean expressions. Therefore, if the HLA compiler generates reasonable code with a boolean expression then by all means use the boolean expression form; it will probably be easier to read.

18.3.7 The While Statement

The only difference between an IF statement and a WHILE loop is a single JMP instruction. Of course, with an IF and a JMP you can simulate most control structures, the WHILE loop is probably the most typical example of this. The typical translation from WHILE to IF/JMP takes the following form:

```
while( expr ) do

    << statements >>

endwhile;


// The above translates to:
```

```
label:
    if( expr ) then

          << statements >>
          jmp label;

    endif;
```

Experienced assembly language programmers know that there is a slightly more efficient implementation if it is likely that the boolean expression is true the first time the program encounters the loop. That translation takes the following form:

```
    jmp testlabel;
label:

    << statements >>

testlabel:
    JT( expr ) label;   // Note: JT means jump if expression is true.
```

This form contains exactly the same number of instructions as the previous translation. The difference is that a JMP instruction was moved out of the loop so that it executes only once (rather than on each iteration of the loop). So this is slightly more efficient than the previous translation. HLA uses this conversion algorithm for WHILE loops with standard boolean expressions.

If you look at HLA's output code, you'll discover that it is really complex and messy. The reason has to do with HLA's code generation algorithm. In order to move code around in the program (required in order to move the test of the boolean expression below the statements that comprise the body of the loop) HLA writes a MASM macro at the top of the loop and then expands that macro at the bottom of the loop. The following short example demonstrates how HLA transforms WHILE statements:

```
    while( eax > 0 ) do

        mov( 0, eax );

    endwhile;


; Translated code:


                jmp     ?2_continue

?2_true:
?2_while:

?2_macro        macro
?2_continue:
                cmp     eax, 0
                ja      ?2_while
                endm

                mov     eax, 0
                ?2_macro
```

```
?2_exitloop:
```

As you'll find by carefully studying this code, HLA emits a macro definition at the point it encounters the WHILE statement. Then it emits an expansion of that macro at the bottom of the loop. This effectively moves the code associated with the computation of the boolean expression to the bottom of the loop.

Because of this code motion, there is very little overhead associated with a WHILE loop that you haven't already seen (i.e., the IF statement). Therefore, with one exception, the WHILE and IF statements share the same efficiency concerns. The single exception is the hybrid WHILE statement. For technical reasons, HLA cannot move the code associated with the termination check of a hybrid WHILE loop to the bottom of the loop. Therefore, whenever you use the hybrid form of the WHILE statement HLA compiles the code you supply at the top of the loop, it adds a JMP instruction to the bottom of the loop, and that JMP instruction executes on each iteration. If this is a problem for your code, you should probably consider a different implementation of the loop.

Example of the compilation of a hybrid WHILE loop:

```
    while
    {
        cmp( eax, 0 );
        jne false;

    }

        mov( 0, eax );

    endwhile;


; Translated code:

?2_while:
?2_continue:
                cmp     eax, 0
                jne     ?2_false
?2_true:
                mov     eax, 0
                jmp     ?2_while
?2_exitloop:
?2_false:
```

18.3.8 repeat..until

To Be Written...

18.3.9 for..endfor

To Be Written...

18.3.10 forever..endfor

To Be Written...

18.3.11 break, breakif

To Be Written...

18.3.12 continue, continueif

To Be Written...

18.3.13 begin..end, exit, exitif

To Be Written...

18.3.14 foreach..endfor

To Be Written...

18.3.15 try..unprotect..exception..anyexception..endtry, raise

To Be Written...

18.4 A Modified IF..ELSE..ENDIF Statement

The IF statement is another statement that doesn't always do exactly what you want. Like the *_while.._onbreak.._endwhile* example above, it's quite possible to redefine the IF statement so that it behaves the way we want it to. In this section you'll see how to implement a variant of the IF..ELSE..ENDIF statement that nests differently than the standard IF statement.

HLA's particular variant of the IF statement has several limitations. One of the major limitations is the inability to combine logical sub-expressions using logical conjunction (and) and logical disjunction (or). It is possible to simulate conjunction and disjunction if you carefully structure your code. Consider the following example:

```
// "C" code employing logical-AND operator:

if( expr1 && expr2 )
{
    << statements >>
}


// Equivalent HLA version:

if( expr1 ) then

    if( expr2 ) then

        << statements >>

    endif;

endif;
```

In both cases ("C" and HLA) the << *statements*>> block executes only if both *expr1* and *expr2* evaluate true. So other than the extra typing involved, it is often very easy to simulate logical conjunction by using two IF statements in HLA.

There is one very big problem with this scheme. Consider what happens if you modify the "C" code to be the following:

```
// "C" code employing logical-AND operator:

if( expr1 && expr2 )
{
    << 'true' statements >>
}
else
{
    << 'false' statements >>
}
```

The only way to convert this to HLA (using the standard HLA high level control constructs) is by duplicating the 'false' statements. This introduces a bit of inefficiency into your code. As a result, many HLA programmers will switch to low-level control constructs or HLA's hybrid control structures in order to avoid duplicating code. Unfortunately, dropping down into low-level code may make your program harder to read. It would be nice if you could efficiently handle this situation without making your code unreadable. Fortunately, you can do exactly this by creating a new version of the IF statement using HLA's multi-part macro facilities.

Before describing how to create this new type of IF statement, we must digress for a moment and explore an interesting feature of HLA's multi-part macro expansion: KEYWORD macros do not have to use unique names. Whenever you declare an HLA KEYWORD macro, HLA accepts whatever name you choose. If that name happens to be already defined, then the KEYWORD macro name takes precedence as long as the macro is active (that is, from the point you invoke the macro name until HLA encounters the TERMINATOR macro). Therefore, the KEYWORD macro name hides the previous definition of that name until the termination of the macro. This feature applies even to the original macro name; that is, it is possible to define a KEYWORD macro with the same name as the original macro to which the KEYWORD macro belongs. This is a very useful feature because it allows you to change the definition of the macro within the scope of the opening and terminating invocations of the macro.

Although not pertinent to the IF statement we are construction, you should note that parameter and local symbols in a macro also override any previously defined symbols of the same name. So if you use that symbol between the opening macro and the terminating macro, you will get the value of the local symbol, not the global symbol. E.g.,

```
var
    i:int32;
    j:int32;
        .
        .
        .
macro abc:i;
    ?i:text := "j";
        .
        .
        .
terminator xyz;
        .
        .
        .
endmacro
        .
        .
        .
    mov( 25, i );
    mov( 10, j );
    abc
        mov( i, eax );   // Loads j's value (10), not 25 into eax.
    xyz;
```

The code above loads 10 into EAX because the "mov(i, eax);" instruction appears between the opening and terminating macros *abc..xyz*. Between those two macros the local definition of *i* takes precedence over the global definition. Since *i* is a text constant that expands to *j*, the aforementioned MOV statement is really equivalent to "mov(j, eax);" That statement, of course, loads 10 into EAX. Since this problem is difficult to see while reading your code, you should choose local symbols in multi-part macros very carefully. A good convention to adopt is to combine your local symbol name with the macro name, e.g.,

```
macro abc : i_abc;
```

You may wonder why HLA allows something to crazy to happen in your source code, in a moment you'll see why this behavior is useful (and now, with this brief message out of the way, back to our regularly scheduled discussion).

Before we digressed to discuss this interesting feature in HLA multi-part macros, we were trying to figure out how to efficiently simulate the conjunction and disjunction operators in an IF statement without resorting to low-level code. The problem in the example appearing earlier in this section is that you would have to duplicate some code in order to convert the IF..ELSE statement properly. The following code shows this problem:

```
// "C" code employing logical-AND operator:

if( expr1 && expr2 )
{
   << 'true' statements >>
}
else
{
   << 'false' statements >>
}


// Corresponding HLA code using the "nested-IF" algorithm:

if( expr1 ) then

   if( expr2 ) then

      << 'true' statements >>

   else

      << 'false' statements >>

   endif;

else

   << 'false' statements >>

endif;
```

Note that this code must duplicate the "<< 'false' statements >>" section if the logic is to exactly match the original "C" code. This means that the program will be larger and harder to read than is absolutely necessary.

One solution to this problem is to create a new kind of IF statement that doesn't nest the same way standard IF statements nest. In particular, if we define the statement such that all IF clauses nested with an outer IF..ENDIF block share the same ELSE and ENDIF clauses. If this were the case, then you could implement the code above as follows:

```
if( expr1 ) then

   if( expr2 ) then

   << 'true' statements >>


else

   << 'false' statements >>

endif;
```

If *expr1* is false, control immediately transfers to the ELSE clause. If the value of *expr1* is true, the control falls through to the next IF statement.

If *expr2* evaluates false, then the program jumps to the single ELSE clause that all IFs share in this statement. Notice that a single ELSE clause (and corresponding 'false' statements) appear in this code; hence the code does not necessarily expand in size. If expr2 evaluates true, then control falls through to the 'true' statements, exactly like a standard IF statement.

Notice that the nested IF statement above does not have a corresponding ENDIF. Like the ELSE clause, all nested IFs in this structure share the same ENDIF. Syntactically, there is no need to end the nested IF statement; the end of the THEN section ends with the ELSE clause, just as the outer IF statement's THEN block ends.

Of course, we can't actually define a new macro named "if" because you cannot redefine HLA reserved words. Nor would it be a good idea to do so even if these were legal (since it would make your programs very difficult to comprehend if the IF keyword had different semantics in different parts of the program. The following program uses the identifiers "_if", "_then", "_else", and "_endif" instead. It is questionable if these are good identifiers in production code (perhaps something a little more different would be appropriate). The following code example uses these particular identifiers so you can easily correlate them with the corresponding high level statements.

```
/**********************************************/
/*                                            */
/* if.hla                                     */
/*                                            */
/* This program demonstrates a modification of */
/* the IF..ELSE..ENDIF statement using HLA's   */
/* multi-part macros.                         */
/*                                            */
/**********************************************/


program newIF;
#include( "stdlib.hhf" )



// Macro implementation of new form of if..then..else..endif.
//
// In this version, all nested IF statements transfer control
// to the same ELSE clause if any one of them have a false
// boolean expression.  Syntax:
//
//  _if( expression ) _then
//
//      <<statements including nested _if clauses>>
//
//  _else // this is optional
//
//      <<statements, but _if clauses are not allowed here>>
//
//  _endif
//
//
// Note that nested _if clauses do not have a corresponding
// _endif clause.  This is because the single _else and/or
// _endif clauses terminate all the nested _if clauses
// including the first one.  Of course, once the code
// encounters an _endif another _if statement may begin.


// Macro to handle the main "_if" clause.
// This code just tests the expression and jumps to the _else
// clause if the expression evaluates false.

macro _if( ifExpr ):elseLbl, hasElse, ifDone;

    ?hasElse := false;
    jf(ifExpr) elseLbl;


// Just ignore the _then keyword.

keyword _then;


// Nested _if clause (yes, HLA lets you replace the main
// macro name with a keyword macro).  Identical to the
// above _if implementation except this one does not
// require a matching _endif clause.  The single _endif
// (matching the first _if clause) terminates all nested
// _if clauses as well as the main _if clause.

keyword _if( nestedIfExpr );
    jf( nestedIfExpr ) elseLbl;

    // If this appears within the _else section, report
    // an error (we don't allow _if clauses nested in
    // the else section, that would create a loop).

    #if( hasElse )

        #error( "All _if clauses must appear before the _else clause" )

    #endif


// Handle the _else clause here.  All we need to is check to
// see if this is the only _else clause and then emit the
// jmp over the else section and output the elseLbl target.

keyword _else;
    #if( hasElse )

        #error( "Only one _else clause is legal per _if.._endif" )

    #else

        // Set hasElse true so we know that we've seen an _else
        // clause in this statement.

        ?hasElse := true;
        jmp ifDone;
        elseLbl:
```

```
    #endif

// _endif has two tasks.  First, it outputs the "ifDone" label
// that _else uses as the target of its jump to skip over the
// else section.  Second, if there was no else section, this
// code must emit the "elseLbl" label so that the false conditional(s)
// in the _if clause(s) have a legal target label.

terminator _endif;

    ifDone:
    #if( !hasElse )

        elseLbl:

    #endif

endmacro;


static
    tr:boolean := true;
    f:boolean := false;

begin newIF;

    // Real quick demo of the _if statement:

    _if( tr ) _then

        _if( tr ) _then
        _if( f ) _then

            stdout.put( "error" nl );

    _else

        stdout.put( "Success" );

    _endif

end newIF;
```

Just in case you're wondering, this program prints "Success" and then quits. This is because the nested "_if" statements are equivalent to the expression "true && true && false" which, of course, is false. Therefore, the "_else" portion of this code should execute.

The only surprise in this macro is the fact that it redefines the *_if* macro as a keyword macro upon invocation of the main *_if* macro. The reason this code does this is so that any nested *_if* clauses do not require a corresponding *_endif* and don't support an *_else* clause.

Implementing an ELSEIF clause introduces some difficulties, hence its absence in this example. The design and implementation of an ELSEIF clause is left to the more serious reader[1].

1. I.e., I don't even want to have to think about this problem!

18.5 Object Oriented Programming in Assembly

18.5.1 Hoopla and Hyperbole

Before discussing object-oriented programming (OOP) in assembly language, it is probably a good idea to take a step back and explore the general benefits of using OOP. After all, without such knowledge, the question of "why bother to use OOP in assembly" is unanswerable.

First of all, despite what some OOP proponents claim, object-oriented programming is not an all-encompassing facility that replaces whatever programming paradigm you currently use. Object-oriented programming techniques are a tool. When used in an appropriate fashion, that tool can save you considerable effort. When misapplied, it can make your programs considerably worse. In some sense, OOP techniques are like recursion: incredibly valuable where it's called for, but inefficient and kludgy when you attempt to use it to solve a problem for which it is not well suited. Fortunately, OOP is well suited for many applications, hence its popularity among high-level language (HLL) programmers.

One of the main benefits to object-oriented programming is that it makes it easier to reuse code.Traditionally, to reuse code you would create huge libraries of different functions and call those functions to perform common tasks. The only problem with the library approach is that in order to effectively reuse your code, you had to write very generic library routines. The result was bloated and slow code (that often handled lots of special cases that would never occur in a specific application); attempts to produce "lean and mean" library routines often meant writing dozens or even hundreds of minor variations of the same functions. It often wasn't possible to easily extend such routines to handle new requirements. This was especially difficult if the source code for the original library routines was not available.

Object-oriented programming techniques provid a solution to this problem. Through OOP-oriented features such as *inheritence* and *polymorphism*, it is possible to extend a simplified library function to handle the specific requirements of a given application without having to rewrite the entire code base.

Because OOP techniques allow you to extend a given set of library routines in ways specific to an application, you would get the impression that this programming paradigm is perfect for assembly language programmers (who want "lean and mean" code that doesn't carry around a lot of bloat). Unfortunately, there are two problems with this idea. First of all, you'll find that traditional OOP languages tend to have *huge* class libraries associated with them. And because of the "layered" approach that OOP fosters, including one, seemingly small, function can wind up including half of the library in your application (ever wondered why a "Hello World" program in Delphi is 256K?). Another problem with the OOP is that it does require a small amount of overhead to implement. This reason alone has scared many assembly language programmers away from using object-oriented programming techniques.

Despite the drawbacks and overblown expectations of OOP (that never seem to be met), OOP techniques are useful for solving many problems. The object-oriented programming paradigm is a handy tool that should appear in your programmer's toolbox - ready to use when the need arises. Just as you wouldn't use a hammer for a job that requires a screwdriver, you shouldn't use OOP in an inappropriate situation. However, when the job calls for a screwdriver, it's nice to have one handy; likewise, when OOP techniques are appropriate, they can provide a fast and efficient solution to a given programming problem.

18.5.2 Some Basic Definitions

To begin with, it's probably a good idea to define a few terms this paper will use. Without further ado:

- CLASS: a class is a data type template (i.e., record or structure) that specifies the data and procedure components of a "class object".
- INSTANCE: an instance is a block of memory with enough storage to hold the data associated with a class variable (see OBJECT).
- OBJECT: a variable of some class type. While there is a subtle difference between objects and instances (having to do with the lifetime of the storage bound to an object), we'll treat the two terms as synonyms for our purposes.
- METHOD: a procedure or function associated with a class.
- INHERITENCE: the ability to reuse fields from another *base* (or ancestor) class.

- POLYMORPHISM: an attribute of classes whereby different (types of) objects can be manipulated by the same method calls. For example, a "print" method could display the value of several different object types without requiring a single function that handles every possible data type one could dream up.
- INFORMATION HIDING: the use of private data fields and procedures/methods to control the access of an object’s internal representation, with the hope of keeping the implementation of a data type independent from its use. This allows easy modification to the data structure without breaking any code that uses the data structure.
- ABSTRACT DATA TYPE (ADT): An abstract data type is a collection of data objects and the functions (which we’ll call *methods*) that operate on the data. In a pure abstract data type, the ADT’s methods are the only code that has access to the data fields of the ADT; external code may only access the data using function calls to get or set data field values (these are the ADT’s *accessor* methods).

18.5.3 OOP Language Facilities

As any die-hard C programmer can tell you, you don’t need an "object-oriented programming language" in order to write object-oriented code. Then again, as any C++ programmer will tell you, it’s far easier to write the code and the resulting code is far easier to read and maintain if you do use an object-oriented programming language when writing object-oriented applications. The same is true in assembly language - you don’t need an assembler that supports object-oriented programming facilities to write object-oriented assembly code, but it’s not very effective to do so.

Today, there are two and a half 80x86 assemblers that provide reasonable support for object-oriented programming in assembly language: HLA (the High-Level Assembler), TASM, and MASM. HLA and TASM directly support classes, objects, and other object-oriented programming facilities. MASM does not, but its STRUCT directive is sufficiently flexible that you can easily create macros to simulate most of the object-oriented programming facilities provided HLA and TASM. Arguably, HLA provides the most complete set of object-oriented programming facilities, so this article will use HLA in its examples. The basic concepts, however, apply to both TASM and MASM as well as HLA.

18.5.4 Classes in HLA

HLA’s classes provide a good mechanism for creating abstract data types. Fundamentally, a class is little more than a RECORD declaration that allows the definition of fields other than data fields (e.g., procedures, constants, and macros). The inclusion of other program declaration objects in the class definition dramatically expands the capabilities of a class over that of a record. For example, with a class it is now possible to easily define an ADT since classes may include data and methods that operate on that data (procedures).

The principle way to create an abstract data type in HLA is to declare a class data type. Classes in HLA always appear in the TYPE section and use the following syntax:

```
classname :  class

                << Class declaration section >>

             endclass;
```

The class declaration section is very similar to the local declaration section for a procedure insofar as it allows CONST, VAL, VAR, and STATIC variable declaration sections. Classes also let you define macros and specify procedure, iterator, and *method* prototypes (method declarations are legal only in classes). Conspicuously absent from this list is the TYPE declaration section. You cannot declare new types within a class.

A method is a special type of procedure that appears only within a class. A little later you will see the difference between procedures and methods, for now you can treat them as being one and the same. Other than a few subtle details regarding class initialization and the use of pointers to classes, their semantics are identical[1]. Generally, if you don't know whether to use a procedure or method in a class, the safest bet is to use a method.

You do not place procedure/iterator/method code within a class. Instead you simply supply *prototypes* for these routines. A routine prototype consists of the PROCEDURE, ITERATOR, or METHOD reserved word, the routine name, any parameters, and a couple of optional procedure attributes (@USE, RETURNS, and EXTERNAL). The actual routine definition (i.e., the body of the routine and any local declarations it needs) appears outside the class.

The following example demonstrates a typical class declaration appearing in the TYPE section:

```
TYPE
    TypicalClass: class

        const
            TCconst := 5;

        val
            TCval := 6;

        var
            TCvar : uns32;       // Private field used only by TCproc.

        static
            TCstatic : int32;

        procedure TCproc( u:uns32 ); returns( "eax" );
        iterator TCiter( i:int32 ); external;
        method TCmethod( c:char );

    endclass;
```

As you can see, classes are very similar to records in HLA. Indeed, you can think of a record as being a class that only allows VAR declarations. HLA implements classes in a fashion quite similar to records insofar as it allocates sequential data fields in sequential memory locations. In fact, with only one minor exception, there is almost no difference between a RECORD declaration and a CLASS declaration that only has a VAR declaration section. Later you'll see exactly how HLA implements classes, but for now you can assume that HLA implements them the same as it does records and you won't be too far off the mark.

You can access the *TCvar* and *TCstatic* fields (in the class above) just like a record's fields. You access the CONST and VAL fields in a similar manner. If a variable of type *TypicalClass* has the name *obj*, you can access the fields of *obj* as follows:

```
        mov ( obj.TCconst, eax );
        mov( obj.TCval, ebx );
        add( obj.TCvar, eax );
        add( obj.TCstatic, ebx );
        obj.TCproc( 20 );       // Calls the TCproc procedure in
TypicalClass.
        etc.
```

If an application program includes the class declaration above, it can create variables using the *TypicalClass* type and perform operations using the above methods. Unfortunately, the application program can also access the fields of the *ADT* data type with impunity. For example, if a program created a variable *MyClass* of type *TypicalClass*, then it could easily execute instructions like

1. Note, however, that the difference between procedures and methods makes all the difference in the world to the object-oriented programming paradigm. Hence the inclusion of methods in HLA's class definitions.

"MOV(MyClass.TCvar, eax);" even though this field might be private to the implementation section. Unfortunately, if you are going to allow an application to declare a variable of type *TypicalClass*, the field names will have to be visible. While there are some tricks we could play with HLA's class definitions to help hide the private fields, the best solution is to thoroughly comment the private fields and then exercise some restraint when accessing the fields of that class. Specifically, this means that ADTs you create using HLA's classes cannot be "pure" ADTs since HLA allows direct access to the data fields. However, with a little discipline, you can simulate a pure ADT by simply electing not to access such fields outside the class' methods, procedures, and iterators.

Prototypes appearing in a class are effectively FORWARD declarations. Like normal forward declarations, all procedures, iterators, and methods you define in a class must have an actual implementation later in the code. Alternately, you may attach the EXTERNAL keyword to the end of a procedure, iterator, or method declaration within a class to inform HLA that the actual code appears in a separate module. As a general rule, class declarations appear in header files and represent the interface section of an ADT. The procedure, iterator, and method bodies appear in the implementation section which is usually a separate source file that you compile separately and link with the modules that use the class.

The following is an example of a sample class procedure implementation:

```
procedure TypicalClass.TCproc( u:uns32 ); nodisplay;
    << Local declarations for this procedure >>
begin TCproc;

    << Code to implement whatever this procedure does >>

end TCProc;
```

There are several differences between a standard procedure declaration and a class procedure declaration. First, and most obvious, the procedure name includes the class name (e.g., *TypicalClass.TCproc*). This differentiates this class procedure definition from a regular procedure that just happens to have the name *TCproc*. Note, however, that you do not have to repeat the class name before the procedure name in the BEGIN and END clauses of the procedure (this is similar to procedures you define in HLA NAMESPACEs).

A second difference between class procedures and non-class procedures is not obvious. Some procedure attributes (@USE, EXTERNAL, RETURNS, @CDECL, @PASCAL, and @STDCALL) are legal only in the prototype declaration appearing within the class while other attributes (@NOFRAME, @NODISPLAY, @NOALIGNSTACK, and ALIGN) are legal only within the procedure definition and not within the class. Fortunately, HLA provides helpful error messages if you stick the option in the wrong place, so you don't have to memorize this rule.

If a class routine's prototype does not have the EXTERNAL option, the compilation unit (that is, the PROGRAM or UNIT) containing the class declaration must also contain the routine's definition or HLA will generate an error at the end of the compilation. For small, local, classes (i.e., when you're embedding the class declaration and routine definitions in the same compilation unit) the convention is to place the class' procedure, iterator, and method definitions in the source file shortly after the class declaration. For larger systems (i.e., when separately compiling a class' routines), the convention is to place the class declaration in a header file by itself and place all the procedure, iterator, and method definitions in a separate HLA unit and compile them by themselves.

18.5.5 Objects

Remember, a class definition is just a type. Therefore, when you declare a class type you haven't created a variable whose fields you can manipulate. An *object* is an *instance* of a class; that is, an object is a variable that is some class type. You declare objects (i.e., class variables) the same way you declare other variables: in a VAR, STATIC, or STORAGE section[1]. A pair of sample object declarations follow:

1. Technically, you could also declare an object in a READONLY section, but HLA does not allow you to define class constants, so there is little utility in declaring class objects in the READONLY section.

```
var
    T1: TypicalClass;
    T2: TypicalClass;
```

For a given class object, HLA allocates storage for each variable appearing in the VAR section of the class declaration. If you have two objects, *T1* and *T2*, of type *TypicalClass* then *T1.TCvar* is unique as is *T2.TCvar*. This is the intuitive result (similar to RECORD declarations); most data fields you define in a class will appear in the VAR declaration section.

Static data objects (e.g., those you declare in the STATIC section of a class declaration) are not unique among the objects of that class; that is, HLA allocates only a single static variable that all variables of that class share. For example, consider the following (partial) class declaration and object declarations:

```
type
    sc: class

        var
            i:int32;

        static
            s:int32;
            .
            .
            .
    endclass;

var
    s1: sc;
    s2: sc;
```

In this example, *s1.i* and *s2.i* are different variables. However, *s1.s* and *s2.s* are aliases of one another Therefore, an instruction like “mov(5, s1.s);” also stores five into *s2.s*. Generally you use static class variables to maintain information about the whole class while you use class VAR objects to maintain information about the specific object. Since keeping track of class information is relatively rare, you will probably declare most class data fields in a VAR section.

You can also create dynamic instances of a class and refer to those dynamic objects via pointers. In fact, this is probably the most common form of object storage and access. The following code shows how to create pointers to objects and how you can dynamically allocate storage for an object:

```
var
    pSC: pointer to sc;
        .
        .
        .
    malloc( @size( sc ) );
    mov( eax, pSC );
        .
        .
        .
    mov( pSC, ebx );
    mov( (type sc [ebx]).i, eax );
```

Note the use of type coercion to cast the pointer in EBX as type *sc*.

18.5.6 Inheritance

Inheritance is one of the most fundamental ideas behind object-oriented programming. The basic idea behind inheritance is that a class inherits, or copies, all the fields from some class and then possibly expands the number of fields in the new data type. For example, suppose you created a data type *point* which describes a point in the planar (two dimensional) space. The class for this point might look like the following:

```
type
    point: class

        var
            x:int32;
            y:int32;

        method distance;

    endclass;
```

Suppose you want to create a point in 3D space rather than 2D space. You can easily build such a data type as follows:

```
type
    point3D: class inherits( point );

        var
            z:int32;

    endclass;
```

The INHERITS option on the CLASS declaration tells HLA to insert the fields of *point* at the beginning of the class. In this case, *point3D* inherits the fields of *point*. HLA always places the inherited fields at the beginning of a class object. The reason for this will become clear a little later. If you have an instance of *point3D* which you call *P3*, then the following 80x86 instructions are all legal:

```
        mov( P3.x, eax );
        add( P3.y, eax );
        mov( eax, P3.z );
        P3.distance();
```

Note that the *P3.distance* method invocation in this example calls the *point.distance* method. You do not have to write a separate *distance* method for the *point3D* class unless you really want to do so (see the next section for details). Just like the *x* and *y* fields, *point3D* objects inherit *point's* methods.

18.5.7 Overriding

Overriding is the process of replacing an existing method in an inherited class with one more suitable for the new class. In the *point* and *point3D* examples appearing in the previous section, the *distance* method (presumably) computes the distance from the origin to the specified point. For a point on a two-dimensional plane, you can compute the distance using the function:

[illegible]

However, the distance for a point in 3D space is given by the equation:

[illegible]

Clearly, if you call the *distance* function for *point* for a *point3D* object you will get an incorrect answer. In the previous section, however, you saw that the *P3* object calls the distance function inherited from the *point* class. Therefore, this would produce an incorrect result.

In this situation the *point3D* data type must override the *distance* method with one that computes the correct value. You cannot simply redefine the *point3D* class by adding a *distance* method prototype:

```
type
   point3D:  class inherits( point )

      var
         z:int32;

      method distance;   // This doesn't work!

   endclass;
```

The problem with the *distance* method declaration above is that *point3D* already has a distance method – the one that it inherits from the *point* class. HLA will complain because it doesn't like two methods with the same name in a single class.

To solve this problem, we need some mechanism by which we can override the declaration of *point.distance* and replace it with a declaration for *point3D.distance*. To do this, you use the OVERRIDE keyword before the method declaration:

```
type
   point3D:  class inherits( point )

      var
         z:int32;

      override method distance;   // This will work!

   endclass;
```

The OVERRIDE prefix tells HLA to ignore the fact that *point3D* inherits a method named distance from the *point* class. Now, any call to the *distance* method via a *point3D* object will call the *point3D.distance* method rather than *point.distance*. Of course, once you override a method using the OVERRIDE prefix, you must supply the method in the implementation section of your code, e.g.,

```
method point3D.distance; nodisplay;

   << local declarations for the distance function>>

begin distance;

   << Code to implement the distance function >>

end distance;
```

18.5.8 Virtual Methods vs. Static Procedures

A little earlier, this chapter suggested that you could treat class methods and class procedures the same. There are, in fact, some major differences between the two (after all, why have methods if they're the same as procedures?). As it turns out, the differences between methods and procedures is crucial if you want to develop object-oriented programs. Methods provide the second feature necessary to support true polymorphism: virtual procedure calls[1]. A virtual procedure call is just a fancy name for an indirect procedure call (using a pointer associated with the object). The key benefit of virtual procedures is that the system automatically calls the right method when using pointers to generic objects.

Consider the following declarations using the *point* class from the previous sections:

```
var
   P2: point;
   P: pointer to point;

Given the declarations above, the following assembly statements are all
legal:
   mov( P2.x, eax );
   mov( P2.y, ecx );
   P2.distance();      // Calls point3D.distance.

   lea( ebx, P2 );     // Store address of P2 into P.
   mov( ebx, P );
   P.distance();       // Calls point.distance.
```

Note that HLA lets you call a method via a pointer to an object rather than directly via an object variable. This is a crucial feature of objects in HLA and a key to implementing *virtual method calls*.

The magic behind polymorphism and inheritance is that object pointers are *generic*. In general, when your program references data indirectly through a pointer, the value of the pointer should be the address of the underlying data type associated with that pointer. For example, if you have a pointer to a 16-bit unsigned integer, you wouldn't normally use that pointer to access a 32-bit signed integer value. Similarly, if you have a pointer to some record, you would not normally cast that pointer to some other record type and access the fields of that other type[1]. With pointers to class objects, however, we can lift this restriction a bit. Pointers to objects may legally contain the address of the object's type *or the address of any object that inherits the fields of that type*. Consider the following declarations that use the *point* and *point3D* types from the previous examples:

```
var
   P2: point;
   P3: point3D;
   p: pointer to point;
      .
      .
      .
   lea( ebx, P2 );
   mov( ebx, p );
   p.distance();       // Calls the point.distance method.
      .
      .
      .
   lea( ebx, P3 );
   mov( ebx, p );      // Yes, this is semantically legal.
   p.distance();       // Surprise, this calls point3D.distance.
```

Since *p* is a pointer to a *point* object, it might seem intuitive for *p.distance* to call the *point.distance* method. However, methods are *polymorphic*. If you've got a pointer to an object

1. Polymorphism literally means "many-faced." In the context of object-oriented programming polymorphism means that the same method name, e.g., *distance*, and refer to one of several different methods.

1. Of course, assembly language programmers break rules like this all the time. For now, let's assume we're playing by the rules and only access the data using the data type associated with the pointer.

and you call a method associated with that object, the system will call the actual (overridden) method associated with the object, not the method specifically associated with the pointer's class type.

Class procedures behave differently than methods with respect to overridden procedures. When you call a class procedure indirectly through an object pointer, the system will always call the procedure associated with the underlying class associated with the pointer. So had *distance* been a procedure rather than a method in the previous examples, the "p.distance();" invocation would always call *point.distance*, even if *p* is pointing at a *point3D* object. The section on Object Initialization, later in this chapter, explains why methods and procedures are different (see "Object Implementation" on page 479).

Note that iterators are also virtual; so like methods an object iterator invocation will always call the (overridden) iterator associated with the actual object whose address the pointer contains. To differentiate the semantics of methods and iterators from procedures, we will refer to the method/iterator calling semantics as *virtual procedures* and the calling semantics of a class procedure as a *static procedure*.

18.5.9 Writing Class Methods, Iterators, and Procedures

For each class procedure, method, and iterator prototype appearing in a class definition, there must be a corresponding procedure, method, or iterator appearing within the program (for the sake of brevity, this section will use the term *routine* to mean procedure, method, or iterator from this point forward). If the prototype does not contain the EXTERNAL option, then the code must appear in the same compilation unit as the class declaration. If the EXTERNAL option does follow the prototype, then the code may appear in the same compilation unit or a different compilation unit (as long as you link the resulting object file with the code containing the class declaration). Like external (non-class) procedures and iterators, if you fail to provide the code the linker will complain when you attempt to create an executable file. To reduce the size of the following examples, they will all define their routines in the same source file as the class declaration.

HLA class routines must always follow the class declaration in a compilation unit. If you are compiling your routines in a separate unit, the class declarations must still precede the code with the class declaration (usually via an #INCLUDE file). If you haven't defined the class by the time you define a routine like *point.distance*, HLA doesn't know that *point* is a class and, therefore, doesn't know how to handle the routine's definition.

Consider the following declarations for a point2D class:

```
type
   point2D: class

      const
         UnitDistance: real32 := 1.0;

      var
         x: real32;
         y: real32;

      static
         LastDistance: real32;

      method distance( fromX: real32;  fromY:real32 ); returns( "st0" );
      procedure InitLastDistance;

   endclass;
```

The distance function for this class should compute the distance from the object's point to (fromX,fromY). The following formula describes this computation:

$$\sqrt{(x - fromX)^2 + (y - fromY)^2}$$

A first pass at writing the distance method might produce the following code:

```
method point2D.distance( fromX:real32; fromY:real32 ); nodisplay;
begin distance;

    fld( x );        // Note: this doesn't work!
    fld( fromX );    // Compute (x-fromX)
    fsub();
    fld( st0 );      // Duplicate value on TOS.
    fmul();          // Compute square of difference.

    fld( y );        // This doesn't work either.
    fld( fromY );    // Compute (y-fromY)
    fsub();
    fld( st0 );      // Compute the square of the difference.
    fmul();

    fsqrt();

end distance;
```

This code probably looks like it should work to someone who is familiar with an object-oriented programming language like C++ or Delphi. However, as the comments indicate, the instructions that push the *x* and *y* variables onto the FPU stack don't work – HLA doesn't automatically define the symbols associated with the data fields of a class within that class' routines.

To learn how to access the data fields of a class within that class' routines, we need to back up a moment and discover some very important implementation details concerning HLA's classes. To do this, consider the following variable declarations:

```
var
    Origin: point2D;
    PtInSpace: point2D;
```

Remember, whenever you create two objects like *Origin* and *PtInSpace*, HLA reserves storage for the *x* and *y* data fields for both of these objects. However, there is only one copy of the *point2D.distance* method in memory. Therefore, were you to call *Origin.distance* and *PtInSpace.distance*, the system would call the same routine for both method invocations. Once inside that method, one has to wonder what an instruction like "fld(x);" would do. How does it associate *x* with *Origin.x* or *PtInSpace.x*? Worse still, how would this code differentiate between the data field *x* and a global object *x*? In HLA, the answer is "it doesn't." You do not specify the data field names within a class routine by simply using their names as though they were common variables.

To differentiate *Origin.x* from *PtInSpace.x* within class routines, HLA automatically passes a pointer to an object's data fields whenever you call a class routine. Therefore, you can reference the data fields indirectly off this pointer. HLA passes this object pointer in the ESI register. This is one of the few places where HLA-generated code will modify one of the 80x86 registers behind your back: **anytime you call a class routine, HLA automatically loads the ESI register with the object's address**. Obviously, you cannot count on ESI's value being preserved across class routine class nor can you pass parameters to the class routine in the ESI register (though it is perfectly reasonable to specify "@USE ESI;" to allow HLA to use the ESI register when setting up other parameters). For class methods and iterators (but not procedures), HLA will also load the EDI register with the address of the class' *virtual method table* (see "Virtual Method Tables" on page 482). While the virtual method table address isn't as interesting as the object address, keep in mind that **HLA-generated code will overwrite any value in the EDI register when you call a method or an iterator**. Again, "EDI" is a good choice for the @USE operand for methods since HLA will wipe out the value in EDI anyway.

Upon entry into a class routine, ESI contains a pointer to the (non-static) data fields associated with the class. Therefore, to access fields like *x* and *y* (in our *point2D* example), you could use an address expression like the following:

```
(type point2D [esi].x
```

Since you use ESI as the base address of the object's data fields, it's a good idea not to disturb ESI's value within the class routines (or, at least, preserve ESI's value if you need to access the objects data fields after some point where you must use ESI for some other purpose). Note that if you call an iterator or a method you do not have to preserve EDI (unless, for some reason, you need access to the virtual method table, which is unlikely).

Accessing the fields of a data object within a class' routines is such a common operation that HLA provides a shorthand notation for casting ESI as a pointer to the class object: THIS. Within a class in HLA, the reserved word THIS automatically expands to a string of the form "(type *classname* [esi])" substituting, of course, the appropriate class name for *classname*. Using the THIS keyword, we can (correctly) rewrite the previous distance method as follows:

```
method point2D.distance( fromX:real32; fromY:real32 ); nodisplay;
begin distance;

    fld( this.x );
    fld( fromX );    // Compute (x-fromX)
    fsub();
    fld( st0 );      // Duplicate value on TOS.
    fmul();          // Compute square of difference.

    fld( this.y );
    fld( fromY );    // Compute (y-fromY)
    fsub();
    fld( st0 );      // Compute the square of the difference.
    fmul();

    fsqrt();

end distance;
```

Don't forget that calling a class routine wipes out the value in the ESI register. This isn't obvious from the syntax of the routine's invocation. It is especially easy to forget this when calling some class routine from inside some other class routine; don't forget that if you do this the internal call wipes out the value in ESI and on return from that call ESI no longer points at the original object. Always push and pop ESI (or otherwise preserve ESI's value) in this situation, e.g.,

```
    .
    .
    .
    fld( this.x );   // ESI points at current object.
    .
    .
    .
    push( esi );               // Preserve ESI across this method call.
    SomeObject.SomeMethod();
    pop( esi );
    .
    .
    .
    lea( ebx, this.x );        // ESI points at original object here.
```

The THIS keyword provides access to the class variables you declare in the VAR section of a class. You can also use THIS to call other class routines associated with the current object, e.g.,

```
this.distance( 5.0, 6.0 );
```

To access class constants and STATIC data fields you generally do not use the THIS pointer. HLA associates constant and static data fields with the whole class, not a specific object. To access these class members, just use the class name in place of the object name. For example, to access the *UnitDistance* constant in the *point2D* class you could use a statement like the following:

```
fld( point2D.UnitDistance );
```

As another example, if you wanted to update the *LastDistance* field in the *point2D* class each time you computed a distance, you could rewrite the *point2D.distance* method as follows:

```
method point2D.distance( fromX:real32; fromY:real32 ); nodisplay;
begin distance;

    fld( this.x );
    fld( fromX );    // Compute (x-fromX)
    fsub();
    fld( st0 );      // Duplicate value on TOS.
    fmul();          // Compute square of difference.

    fld( this.y );
    fld( fromY );    // Compute (y-fromY)
    fsub();
    fld( st0 );      // Compute the square of the difference.
    fmul();

    fsqrt();

    fst( point2D.LastDistance );  // Update shared (STATIC) field.

end distance;
```

To understand why you use the class name when referring to constants and static objects but you use THIS to access VAR objects, check out the next section.

Class procedures are also static objects, so it is possible to call a class procedure by specifying the class name rather than an object name in the procedure invocation, e.g., both of the following are legal:

```
    Origin.InitLastDistance();
    point2D.InitLastDistance();
```

There is, however, a subtle difference between these two class procedure calls. The first call above loads ESI with the address of the *Origin* object prior to actually calling the *InitLastDistance* procedure. The second call, however, is a direct call to the class procedure without referencing an object; therefore, HLA doesn't know what object address to load into the ESI register. In this case, HLA loads NULL (zero) into ESI prior to calling the *InitLastDistance* procedure. Because you can call class procedures in this manner, it's always a good idea to check the value in ESI within your class procedures to verify that HLA contains an object address. Checking the value in ESI is a good way to determine which calling mechanism is in use. Later, this chapter will discuss constructors and object initialization; there you will see a good use for static procedures and calling those procedures directly (rather than through the use of an object).

18.5.10 Object Implementation

In a high level object-oriented language like C++ or Delphi, it is quite possible to master the use of objects without really understanding how the machine implements them. One of the reasons for learning assembly language programming is to fully comprehend low-level implementation details so one can make educated decisions concerning the use of programming constructs like objects. Further, since assembly language allows you to poke around with data structures at a very low-level, knowing how HLA implements objects can help you create certain algorithms that would not be possible without a detailed knowledge of object implementation. Therefore, this section, and its corresponding subsections, explains the low-level implementation details you will need to know in order to write object-oriented HLA programs.

HLA implements objects in a manner quite similar to records. In particular, HLA allocates storage for all VAR objects in a class in a sequential fashion, just like records. Indeed, if a class consists of only VAR data fields, the memory representation of that class is nearly identical to that of a corresponding RECORD declaration. Consider the following Student record declaration and the corresponding class:

```
type
   student:  record
                Name: char[65];
                Major: int16;
                SSN:   char[12];
                Midterm1: int16;
                Midterm2: int16;
                Final: int16;
                Homework: int16;
                Projects: int16;
             endrecord;

   student2: class
                Name: char[65];
                Major: int16;
                SSN:   char[12];
                Midterm1: int16;
                Midterm2: int16;
                Final: int16;
                Homework: int16;
                Projects: int16;
             endclass;
```

Student RECORD Implementation in Memory

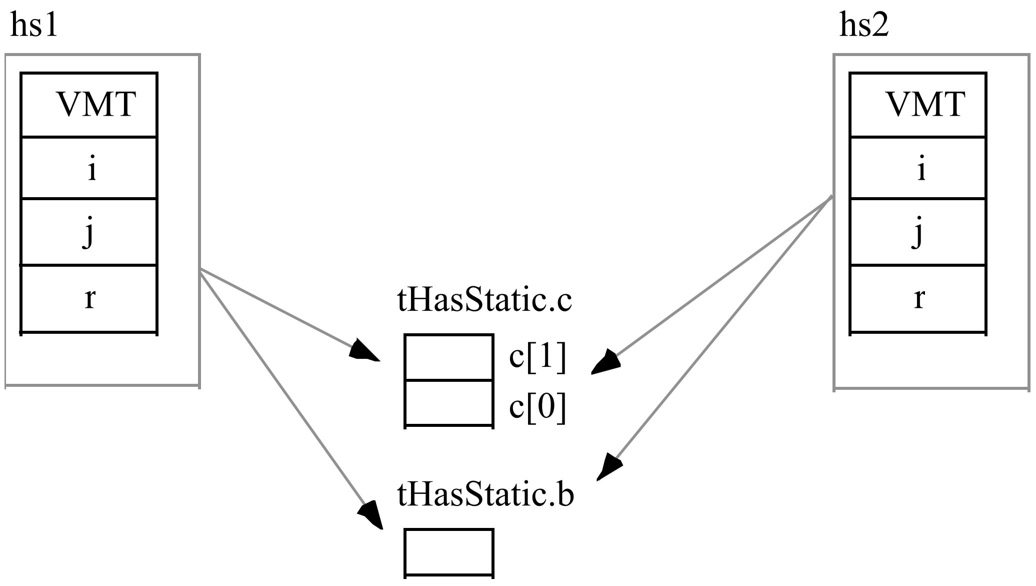
Student CLASS Implementation in Memory

If you look carefully at these two figures, you'll discover that the only difference between the class and the record implementations is the inclusion of the VMT (virtual method table) pointer field at the beginning of the class object. This field, which is always present in a class, contains the address of the class' virtual method table which, in turn, contains the addresses of all the class' methods and iterators. The VMT field, by the way, is present even if a class doesn't contain any methods or iterators.

As pointed out in previous sections, HLA does not allocate storage for STATIC objects within the object's storage. Instead, HLA allocates a single instance of each static data field that all objects share. As an example, consider the following class and object declarations:

```
type
    tHasStatic: class

        var
            i:int32;
            j:int32;
            r:real32;

        static
            c:char[2];
            b:byte;

    endclass;

var
    hs1: tHasStatic;
    hs2: tHasStatic;
```

shows the storage allocation for these two objects in memory.



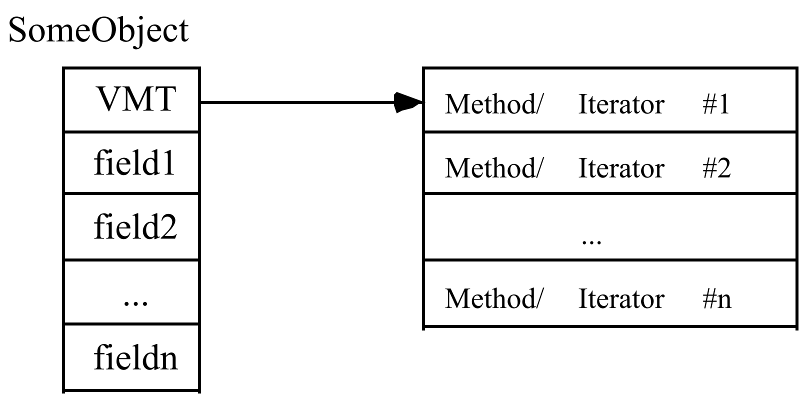
Object Allocation with Static Data Fields

Of course, CONST, VAL, and #MACRO objects do not have any run-time memory requirements associated with them, so HLA does not allocate any storage for these fields. Like the STATIC data fields, you may access CONST, VAL, and #MACRO fields using the class name as well as an object name. Hence, even if *tHasStatic* has these types of fields, the memory organization for *tHasStatic* objects would still be the same as shown in .

Other than the presence of the virtual method table pointer (VMT), the presence of methods, iterators, and procedures has no impact on the storage allocation of an object. Of course, the machine instructions associated with these routines does appear somewhere in memory. So in a sense the code for the routines is quite similar to static data fields insofar as all the objects share a single instance of the routine.

18.5.10.1 Virtual Method Tables

When HLA calls a class procedure, it directly calls that procedure using a CALL instruction, just like any normal non-class procedure call. Methods and iterators are another story altogether. Each object in the system carries a pointer to a virtual method table which is an array of pointers to all the methods and iterators appearing within the object's class.

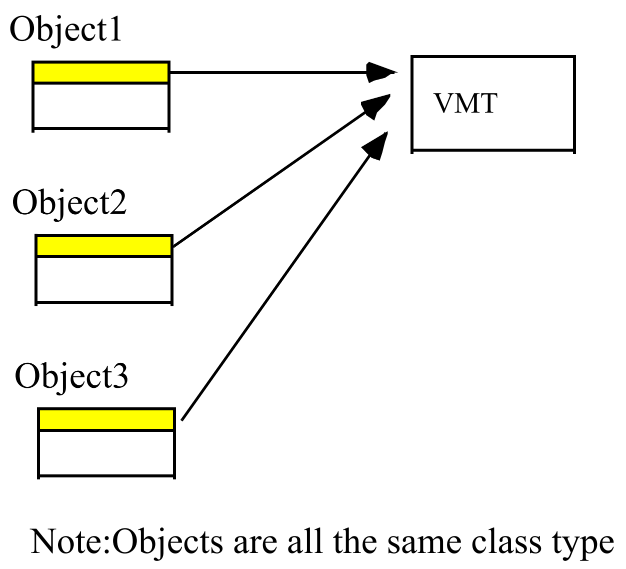


Virtual Method Table Organization

Each iterator or method you declare in a class has a corresponding entry in the virtual method table. That dword entry contains the address of the first instruction of that iterator or method. To call a class method or iterator is a bit more work than calling a class procedure (it requires one additional instruction plus the use of the EDI register). Here is a typical calling sequence for a method:

```
      mov( ObjectAdrs, ESI );       // All class routines do this.
      mov( [esi], edi );            // Get the address of the VMT into
EDI
      call( (type dword [edi+n]));  // "n" is the offset of the method's
entry
                                    //  in the VMT.
```

For a given class there is only one copy of the VMT in memory. This is a static object so all objects of a given class type share the same VMT. This is reasonable since all objects of the same class type have exactly the same methods and iterators (see).



All Objects That are the Same Class Type Share the Same VMT

Although HLA builds the VMT record structure as it encounters methods and iterators within a class, HLA does not automatically create the actual run-time virtual method table for you. You must explicitly declare this table in your program. To do this, you include a statement like the following in a STATIC or READONLY declaration section of your program, e.g.,

```
readonly
    VMT( classname );
```

Since the addresses in a virtual method table should never change during program execution, the READONLY section is probably the best choice for declaring VMTs. It should go without saying that changing the pointers in a VMT is, in general, a really bad idea. So putting VMTs in a STATIC section is usually not a good idea.

A declaration like the one above defines the variable *classname._VMT_*. In section 18.5.11 (see "Constructors and Object Initialization" on page 487) you see that you'll need this name when initializing object variables. The class declaration automatically defines the *classname._VMT_* symbol as an external static variable. The declaration above just provides the actual definition of this external symbol.

The declaration of a VMT uses a somewhat strange syntax because you aren't actually declaring a new symbol with this declaration, you're simply supplying the data for a symbol that you previously declared implicitly by defining a class. That is, the class declaration defines the static table variable *classname._VMT_*, all you're doing with the VMT declaration is telling HLA to emit the actual data for the table. If, for some reason, you would like to refer to this table using a name other than *classname._VMT_*, HLA does allow you to prefix the declaration above with a variable name, e.g.,

```
readonly
    myVMT: VMT( classname );
```

In this declaration, *myVMT* is an alias of *classname._VMT_*. As a general rule, you should avoid aliases in a program because they make the program more difficult to read and understand. Therefore, it is unlikely that you would ever really need to use this type of declaration.

Like any other global static variable, there should be only one instance of a VMT for a given class in a program. The best place to put the VMT declaration is in the same source file as the class' method, iterator, and procedure code (assuming they all appear in a single file). This way you will automatically link in the VMT whenever you link in the routines for a given class.

18.5.10.2 Object Representation with Inheritance

Up to this point, the discussion of the implementation of class objects has ignored the possibility of inheritance. Inheritance only affects the memory representation of an object by adding fields that are not explicitly stated in the class declaration.

Adding inherited fields from a *base class* to another class must be done carefully. Remember, an important attribute of a class that inherits fields from a base class is that you can use a pointer to the base class to access the inherited fields from that base class in another class. As an example, consider the following classes:

```
type
    tBaseClass: class
        var
            i:uns32;
            j:uns32;
            r:real32;

        method mBase;
    endclass;

    tChildClassA: class inherits( tBaseClass );
        var
            c:char;
            b:boolean;
            w:word;

        method mA;
    endclass;

    tChildClassB: class inherits( tBaseClass );
        var
            d:dword;
            c:char;
            a:byte[3];
    endclass;
```

Since both *tChildClassA* and *tChildClassB* inherit the fields of *tBaseClass*, these two child classes include the *i*, *j*, and *r* fields as well as their own specific fields. Furthermore, whenever you have a pointer variable whose base type is *tBaseClass*, it is legal to load this pointer with the address of any child class of *tBaseClass*; therefore, it is perfectly reasonable to load such a pointer with the address of a *tChildClassA* or *tChildClassB* variable, e.g.,

```
var
    B1: tBaseClass;
    CA: tChildClassA;
    CB: tChildClassB;
    ptr: pointer to tBaseClass;
        .
        .
        .
    lea( ebx, B1 );
    mov( ebx, ptr );
    << Use ptr >>
        .
        .
        .
    lea( eax, CA );
    mov( ebx, ptr );
    << Use ptr >>
        .
        .
        .
    lea( eax, CB );
    mov( eax, ptr );
    << Use ptr >>
```

Since *ptr* points at an object of *tBaseClass*, you may legally (from a semantic sense) access the *i*, *j*, and *r* fields of the object where *ptr* is pointing. It is not legal to access the *c, b, w,* or *d* fields of the *tChildClassA* or *tChildClassB* objects since at any one given moment the program may not know exactly what object type *ptr* references.

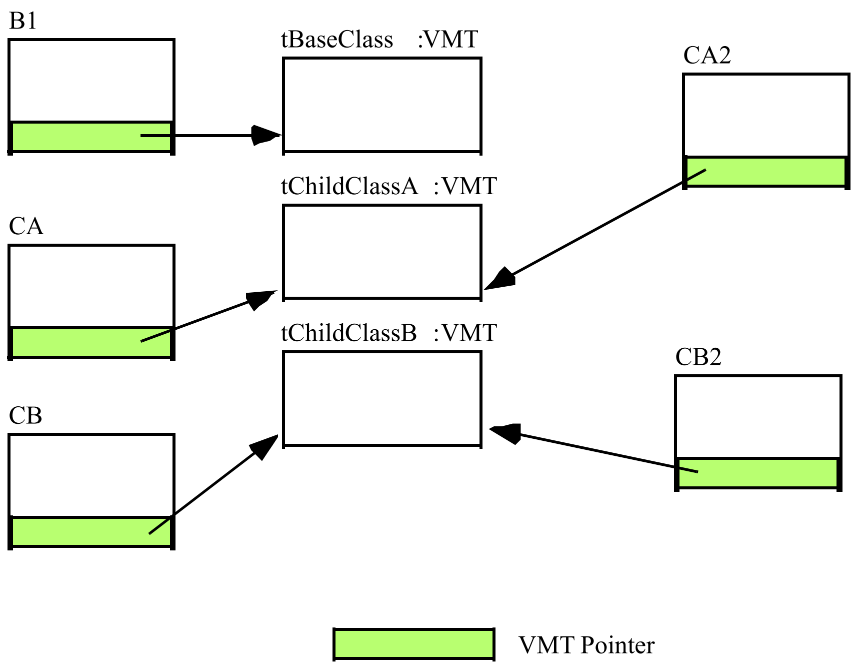
In order for inheritance to work properly, the *i, j,* and *r* fields must appear at the same offsets all child classes as they do in *tBaseClass*. This way, an instruction of the form "mov((type tBaseClass [ebx]).i, eax);" will correct access the i field even if EBX points at an object of type *tChildClassA* or *tChildClassB*. shows the layout of the child and base classes:

Layout of Base and Child Class Objects in Memory

Note that the new fields in the two child classes bear no relation to one another, even if they have the same name (e.g., field *c* in the two child classes does not lie at the same offset). Although the two child classes share the fields they inherit from their common base class, any new fields they add are unique and separate. Two fields in different classes share the same offset only by coincidence.

All classes (even those that aren't related to one another) place the pointer to the virtual method table at offset zero within the object. There is a single VMT associated with each class in a program; even classes that inherit fields from some base class have a VMT that is (generally) different than the base class' VMT. shows how objects of type tBaseClass, tChildClassA and tChildClassB point at their specific VMTs:

```
var
        B1: tBaseClass;
        CA: tChildClassA;
        CB: tChildClassB;
        CB2: tChildClassB;
        CA2: tChildClassA;
```



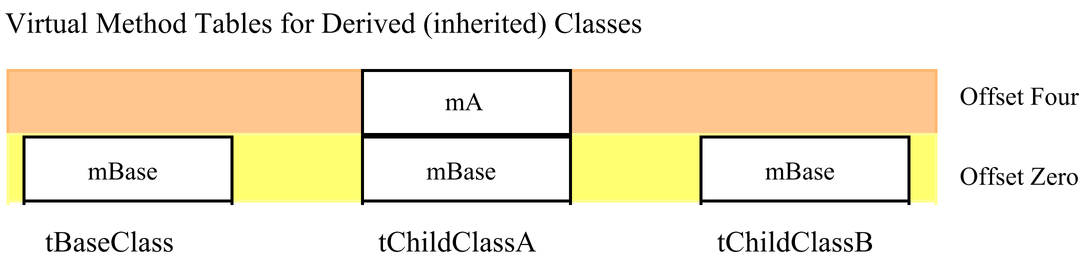
Virtual Method Table References from Objects

A virtual method table is nothing more than an array of pointers to the methods and iterators associated with a class. The address of the first method or iterator appearing in a class is at offset zero, the address of the second appears at offset four, etc. You can determine the offset value for a given iterator or method by using the @offset function. If you want to call a method or iterator directly (using 80x86 syntax rather than HLA's high level syntax), you code use code like the following:

```
var
    sc: tBaseClass;
        .
        .
        .
    lea( esi, sc );             // Get the address of the object (& VMT).
    mov( [esi], edi );          // Put address of VMT into EDI.
    call( (type dword [edi+@offset( tBaseClass.mBase )] );
```

Of course, if the method has any parameters, you must push them onto the stack before executing the code above. Don't forget, when making direct calls to a method, that you must load ESI with the address of the object. Any field references within the method will probably depend upon ESI containing this address. The choice of EDI to contain the VMT address is nearly arbitrary. Unless you're doing something tricky (like using EDI to obtain run-time type information), you could use any register you please here. As a general rule, you should use EDI when simulating class iterator/method calls because this is the convention that HLA employs and most programmers will expect this.

Whenever a child class inherits fields from some base class, the child class' VMT also inherits entries from the base class' VMT. For example, the VMT for class *tBaseClass* contains only a single entry – a pointer to method *tBaseClass.mBase*. The VMT for class *tChildClassA* contains two entries: a pointer to *tBaseClass.mBase* and *tChildClassA.mA*. Since *tChildClassB* doesn't define any new methods or iterators, *tChildClassB's* VMT contains only a single entry, a pointer to the *tBaseClass.mBase* method. Note that *tChildClassB's* VMT is identical to *tBaseClass*' VMT. Nevertheless, HLA produces two distinct VMTs. This is a critical fact that we will make use of a little later. shows the relationship between these VMTs:



Virtual Method Tables for Inherited Classes

Although the VMT always appears at offset zero in an object (and, therefore, you can access the VMT using the address expression "[ESI]" if ESI points at an object), HLA actually inserts a symbol into the symbol table so you may refer to the VMT symbolically. The symbol *_pVMT_* (pointer to Virtual Method Table) provides this capability. So a more readable way to access the VMT pointer (as in the previous code example) is

```
lea( esi, sc );
mov( (type tBaseClass [esi])._pVMT_, edi );
call( (type dword [edi+@offset( tBaseClass.mBase )] );
```

If you need to access the VMT directly, there are a couple ways to do this. Whenever you declare a class object, HLA automatically includes a field named *_VMT_* as part of that class. *_VMT_* is a static array of double word objects. Therefore, you may refer to the VMT using an identifier of the form *classname._VMT_*. Generally, you shouldn't access the VMT directly, but as you'll see shortly, there are some good reasons why you need to know the address of this object in memory.

18.5.11 Constructors and Object Initialization

If you've tried to get a little ahead of the game and write a program that uses objects prior to this point, you've probably discovered that the program inexplicably crashes whenever you attempt to run it. We've covered a lot of material in this chapter thus far, but you are still missing one crucial piece of information – how to properly initialize objects prior to use. This section will put the final piece into the puzzle and allow you to begin writing programs that use classes.

Consider the following object declaration and code fragment:

```
var
   bc: tBaseClass;
       .
       .
       .
   bc.mBase();
```

Remember that variables you declare in the VAR section are uninitialized at run-time. Therefore, when the program containing these statements gets around to executing *bc.mBase*, it executes the three-statement sequence you've seen several times already:

```
    lea( esi, bc);
    mov( [esi], edi );
    call( (type dword [edi+@offset( tBaseClass.mBase )] );
```

The problem with this sequence is that it loads EDI with an undefined value assuming you haven't previously initialized the *bc* object. Since EDI contains a garbage value, attempting to call a subroutine at address "[EDI+@offset(tBaseClass.mBase)]" will likely crash the system. Therefore, before using an object, you must initialize the *_pVMT_* field with the address of that object's VMT. One easy way to do this is with the following statement:

```
mov( &tBaseClass._VMT_, bc._pVMT_ );
```

Always remember, **before using an object, be sure to initialize the virtual method table pointer for that field**.

Although you must initialize the virtual method table pointer for all objects you use, this may not be the only field you need to initialize in those objects. Each specific class may have its own application-specific initialization that is necessary. Although the initialization may vary by class, you need to perform the same initialization on each object of a specific class that you use. If you ever create more than a single object from a given class, it is probably a good idea to create a procedure to do this initialization for you. This is such a common operation that object-oriented programmers have given these initialization procedures a special name: *constructors*.

Some object-oriented languages (e.g., C++) use a special syntax to declare a constructor. Others (e.g., Delphi) simply use existing procedure declarations to define a constructor. One advantage to employing a special syntax is that the language knows when you define a constructor and can automatically generate code to call that constructor for you (whenever you declare an object). Languages, like Delphi, require that you explicitly call the constructor; this can be a minor inconvenience and a source of defects in your programs. HLA does not use a special syntax to declare constructors – you define constructors using standard class procedures. As such, you will need to explicitly call the constructors in your program; however, you'll see an easy method for automating this in a later section of this chapter.

Perhaps the most important fact you must remember is that **constructors must be class procedures**. You must not define constructors as methods (or iterators). The reason is quite simple: one of the tasks of the constructor is to initialize the pointer to the virtual method table and you cannot call a class method or iterator until after you've initialized the VMT pointer. Since class procedures don't use the virtual method table, you can call a class procedure prior to initializing the VMT pointer for an object.

By convention, HLA programmers use the name *Create* for the class constructor. There is no requirement that you use this name, but by doing so you will make your programs easier to read and follow by other programmers.

As you may recall, you can call a class procedure via an object reference or a class reference. E.g., if *clsProc* is a class procedure of class *tClass* and *Obj* is an object of type *tClass*, then the following two class procedure invocations are both legal:

```
    tClass.clsProc();
    Obj.clsProc();
```

There is a big difference between these two calls. The first one calls *clsProc* with ESI containing zero (NULL) while the second invocation loads the address of *Obj* into ESI before the call. We can use this fact to determine within a method the particular calling mechanism.

18.5.12 Dynamic Object Allocation Within the Constructor

As it turns out, most programs allocate objects dynamically using *malloc* and refer to those objects indirectly using pointers. This adds one more step to the initialization process – allocating storage for the object. The constructor is the perfect place to allocate this storage. Since you probably won't need to allocate all objects dynamically, you'll need two types of constructors: one that allocates storage and then initializes the object, and another that simply initializes an object that already has storage.

Another constructor convention is to merge these two constructors into a single constructor and differentiate the type of constructor call by the value in ESI. On entry into the class' *Create* procedure, the program checks the value in ESI to see if it contains NULL (zero). If so, the constructor calls *malloc* to allocate storage for the object and returns a pointer to the object in ESI. If ESI does not contain NULL upon entry into the procedure, then the constructor assumes that ESI points at a valid object and skips over the memory allocation statements. At the very least, a constructor initializes the pointer to the VMT; therefore, the minimalist constructor will look like the following:

```
procedure tBaseClass.mBase; nodisplay;
begin mBase;

    if( ESI = 0 ) then

        push( eax );   // Malloc returns its result here, so save it.
        malloc( @size( tBaseClass ));
        mov( eax, esi );  // Put pointer into ESI;
        pop( eax );

    endif;

    // Initialize the pointer to the VMT:
    // (remember, "this" is shorthand for (type tBaseClass [esi])"

    mov( &tBaseClass._VMT_, this._pVMT_ );

    // Other class initialization would go here.

end mBase;
```

After you write a constructor like the one above, you choose an appropriate calling mechanism based on whether your object's storage is already allocated. For pre-allocated objects (i.e., those you've declared in VAR, STATIC, or STORAGE sections[1] or those you've previously allocated storage for via *malloc*) you simply load the address of the object into ESI and call the constructor. For those objects you declare as a variable, this is very easy – just call the appropriate *Create* constructor:

```
var
    bc0: tBaseClass;
    bcp: pointer to tBaseClass;
        .
        .
        .
    bc0.Create();  // Initializes pre-allocated bc0 object.
        .
        .
        .
    malloc( @size( tBaseClass ));  // Allocate storage for bcp object.
    mov( eax, bcp );
        .
        .
        .
    bcp.Create();  // Initializes pre-allocated bcp object.
```

1. You generally do not declare objects in READONLY sections because you cannot initialize them.

Note that although *bcp* is a pointer to a *tBaseClass* object, the *Create* method does not automatically allocate storage for this object. The program already allocates the storage earlier. Therefore, when the program calls *bcp.Create* it loads ESI with the address contained within *bcp*; since this is not NULL, the *tBaseClass.Create* procedure does not allocate storage for a new object. By the way, the call to *bcp.Create* emits the following sequence of machine instructions:

```
    mov( bcp, esi );
    call tBaseClass.Create;
```

Until now, the code examples for a class procedure call always began with an LEA instruction. This is because all the examples to this point have used object variables rather than pointers to object variables. Remember, a class procedure (method/iterator) call passes the address of the object in the ESI register. For object variables HLA emits an LEA instruction to obtain this address. For pointers to objects, however, the actual object address is the *value* of the pointer variable; therefore, to load the address of the object into ESI, HLA emits a MOV instruction that copies the value of the pointer into the ESI register.

In the example above, the program preallocates the storage for an object prior to calling the object constructor. While there are several reasons for preallocating object storage (e.g., you're creating a dynamic array of objects), you can achieve most simple object allocations like the one above by calling a standard *Create* method (i.e., one that allocates storage for an object if ESI contains NULL). The following example demonstrates this:

```
var
    bcp2: pointer to tBaseClass;
        .
        .
        .
    tBaseClass.Create();   // Calls Create with ESI=NULL.
    mov( esi, bcp2 );      // Save pointer to new class object in bcp2.
```

Remember, a call to a *tBaseClass.Create* constructor returns a pointer to the new object in the ESI register. It is the caller's responsibility to save the pointer this function returns into the appropriate pointer variable; the constructor does not automatically do this for you.

18.6 Compiling Resource Scripts Using HLA

HLA's compile-time language facilities provide the ability to embed domain-specific languages directly in an HLA source file. This paper discusses how to create a domain-specific embedded language that handles Windows Resources. This mini-language not only provides access to these resources in your HLA source files, but it also creates a resource script file (.rc file) that you may compile with the Microsoft Resource Compiler (RC.EXE).

18.6.1 The Motivation

Working with resources when writing Wi32 assembly language programs is usually a two-step process. First, you write some assembly code that requests a resource object from the executable file; then you write a resource script file that matches the resources, via some numeric identifier, with the actual resource file on the disk. The problem with this approach is that you have to maintain (and keep consistent) two sets of source files - an HLA/assembly source file and a resource script (.rc) file. The reason you have to maintain two files is because the assembler associates names with numeric values in a different way than Microsoft's resource compiler. The resource compiler uses C's "#define" syntax, which is not compatible with constant declarations in assembly language. Therefore, you have to create a set of definitions like the following for the resource compiler:

```
#define resource_1 101
#define resource_2 102
#define resource_3 2005
```

When working in assembly language (e.g., HLA), you need to use statements like the following to declare these symbolic names with these values:

```
const
   resource_1 := 101;
   resource_2 := 102;
   resource_3 := 2005;
```

Although entering these two sets of constant defintions twice is a big pain, the real problem comes when you modify either set of definitions and find that you need to edit the other set to keep them consistent. At the very least, we'd like to be able to maintain one set of declarations to avoid consistency problems.

Another problem with using resource scripts is that you have to maintain two separate files - an assembly language source file and a resource script file. While breaking up programs into multiple files isn't always a bad idea, the resource file often contains common things (like strings) that you'd like to find easily when working on small assembly projects. Sometimes, it's just more convenient to put the resources in the same soure file as your assembly source code. One final problem with using a separate resource file is that the resource scripting language is radically different from assembly syntax. It would be nice to be able to declare resources in an assembly language source file like any other object and have the assembler handle the details of creating the resource script (or compiling the resource) for you.

18.6.2 The HLA Solution

Although processing script files is a pipe dream within most assemblers, HLA's compile-time language provides sufficient capability to achieve this. Here are some of the HLA features that give us the capability to create our own language within HLA:

- Context-free macros
- The ability to create (user-defined) output files during assembly
- The ability to execute system commands during assembly
- Conditional assembly and compile-time loops
- Powerful compile-time string processing facilities

The approach we will take here is to define a new "HLA declaration section" using a context-free macro. Within this section a programmer will declare Win32 resource objects. HLA will create a resource script (.rc) file on the basis of the data appearing in this section, and will define a set of symbolic constants by which the rest of the HLA program can refer to those objects. The basic syntax for this new section will be the following:

```
resource( "filename.rc" )
   <<resource definitions>>
endresource;
```

Between the `resource` and `endresource` statements, this code will construct the resource script file using the filename you specify in the `resource` statement. Upon encountering the `endresource` statement, HLA will close the script file and then execute Microsoft's "rc.exe" program to compile the resource code. The declarations between these two statements will also generate symbols that the HLA code can use. In general, there will only be a single resource declaration section in any one given HLA source file; the design of the macros that handle this declaration section will assume that this is the case. In particular, you should avoid nesting `resource/endresource` declaration sections. Though HLA allows this syntax, the efficiency of the macros' execution (at compile-time) is based on the assumption that you've only got one resource/endresource declaration section in an HLA program.

18.6.3 The Resource..Endresource Declaration Section

To Be Written....

18.7 Structures in Assembly Language Programs

Structures, or records, are an abstract data type that allows a programmer to collect different objects together into a single, composite, object. Structures can help make programs easier to read, write, modify, and maintain. Used appropriately, they can also help your programs run faster. Despite the advantages that structures offer, their appearance in assembly language is a relatively recent phenomenon (in the past two decades, or so), and many assemblers still do not support this facility. Furthermore, many "old-timer" assembly language programmers attempt to argue that the appearance of records violates the whole principle of "assembly language programming." This article will certain refute such arguments and describe the benefits of using structures in an assembly language program.

Despite the fact that records have been available in various assembly languages for years (e.g., Microsoft's MASM assembler introduced structures in 80x86 assembly language in the 1980s), the "lack of support for structures" is a common argument against assembly language by HLL programmers who don't know much about assembly. In some respects, their ignorance is justified -- many assemblers don't support structures or records. A second goal of this article is to educate assembly language programmers to counter claims like "assembly language doesn't support structures." Hopefully, that same education will convince those assembly language programmers who've never bothered to use structures, to consider their use.

This article will use the term "record" to denote a structure/record to avoid confusion with the more general term "data structure". Note, however, that the terms "record" and "structure" are synonymous in this article.

18.7.1 What is a Record (Structure)?

The whole purpose of a record is to let you encapsulate different, but logically related, data into a single package. Here is a typical record declaration, in HLA using the RECORD / ENDRECORD declaration:

```
type
   student:
            record
               Name:       string;
               Major:      int16;
               SSN:        char[12];
               Midterm1:   int16;
               Midterm2:   int16;
               Final:      int16;
               Homework:   int16;
               Projects:   int16;
            endrecord;
```

The field names within the record must be unique. That is, the same name may not appear two or more times in the same record. However, in reasonable assemblers (like HLA) that support true structures, all the field names are local to that record. With such assemblers, you may reuse those field names elsewhere in the program.

The RECORD/ENDRECORD type declaration may appear in a variable declaration section (e.g., an HLA STATIC or VAR section) or in a TYPE declaration section. In the previous example the *Student* declaration appears in an HLA TYPE section, so this does not actually allocate any storage for a *Student* variable. Instead, you have to explicitly declare a variable of type *Student*. The following example demonstrates how to do this:

```
var
   John: Student;
```

This allocates 28 bytes of storage: four bytes for the Name field (HLA strings are four-byte pointers to character data found elsewhere in memory), 12 bytes for the SSN field, and two bytes for each of the other six fields.

If the label *John* corresponds to the *base address* of this record, then the *Name* field is at offset *John+0*, the *Major* field is at offset *John+4*, the *SSN* field is at offset *John+6*, etc.

To access an element of a structure you need to know the offset from the beginning of the structure to the desired field. For example, the *Major* field in the variable *John* is at offset 4 from the base address of *John*. Therefore, you could store the value in AX into this field using the instruction

```
mov( ax, (type word John[4]) );
```

Unfortunately, memorizing all the offsets to fields in a record defeats the whole purpose of using them in the first place. After all, if you've got to deal with these numeric offsets why not just use an array of bytes instead of a record?

Well, as it turns out, assemblers like HLA that support true records commonly let you refer to field names in a record using the same mechanism C/C++ and Pascal use: the dot operator. To store AX into the *Major* field, you could use "mov(ax, John.Major);" instead of the previous instruction. This is much more readable and certainly easier to use.

18.7.2 Record Constants

HLA lets you define record constants. In fact, HLA is probably unique among x86 assemblers insofar as it supports both symbolic record constants and literal record constants. Record constants are useful as initializers for static record variables. They are also quite useful as compile-time data structures when using the HLA compile-time language (that is, the macro processor language). This section discusses how to create record constants.

A record literal constant takes the following form:

```
          RecordTypeName:[ List_of_comma_separated_constants ]
```

The *RecordTypeName* is the name of a record data type you've defined in an HLA TYPE section prior to this point. To create a record constant you must have previously defined the record type in a TYPE section of your program.

The constant list appearing between the brackets are the data items for each of the fields in the specified record. The first item in the list corresponds to the first field of the record, the second item in the list corresponds to the second field, etc. The data types of each of the constants appearing in this list must match their respective field types. The following example demonstrates how to use a literal record constant to initialize a record variable:

```
type
   point:
         record
            x:int32;
            y:int32;
            z:int32;
         endrecord;

static
   Vector: point := point:[ 1, -2, 3 ];
```

This declaration initializes *Vector.x* with 1, *Vector.y* with -2, and *Vector.z* with 3.

You can also create symbolic record constants by declaring record objects in the CONST or VAL sections of an HLA program. You access fields of these symbolic record constants just as you would access the field of a record variable, using the dot operator. Since the object is a constant, you can specify the field of a record constant anywhere a constant of that field's type is legal. You can also employ symbolic record constants as record variable initializers. The following example demonstrates this:

```
type
   point:
         record
            x:int32;
            y:int32;
            z:int32;
         endrecord;

const
   PointInSpace: point := point:[ 1, 2, 3 ];

static
   Vector: point := PointInSpace;
   XCoord: int32 := PointInSpace.x;
```

18.7.3 Arrays of Records

It is a perfectly reasonable operation to create an array of records. To do so, you simply create a record type and then use the standard array declaration syntax when declaring an array of that record type. The following example demonstrates how you could do this:

```
type
   recElement:
      record
         << fields for this record >>
      endrecord;
      .
      .
      .
static
   recArray: recElement[4];
```

Naturally, you can create multidimensional arrays of records as well. You would use the standard row or column major order functions to compute the address of an element within such records. The only thing that really changes (from the discussion of arrays) is that the size of each element is the size of the record object.

```
static
   rec2D: recElement[ 4, 6 ];
```

18.7.4 Arrays and Records as Record Fields

Records may contain other records or arrays as fields. Consider the following definition:

```
type
   Pixel:
      record
         Pt:         point;
         color:      dword;
      endrecord;
```

The definition above defines a single point with a 32 bit color component. When initializing an object of type Pixel, the first initializer corresponds to the *Pt* field, *not the x-coordinate field*. **The following definition is incorrect:**

```
static
    ThisPt: Pixel := Pixel:[ 5, 10 ];   // Syntactically incorrect!
```

The value of the first field (“5”) is not an object of type *point*. Therefore, the assembler generates an error when encountering this statement. HLA will allow you to initialize the fields of *Pixel* using declarations like the following:

```
static
    ThisPt: Pixel := Pixel:[ point:[ 1, 2, 3 ], 10 ];
    ThatPt: Pixel := Pixel:[ point:[ 0, 0, 0 ], 5 ];
```

Accessing *Pixel* fields is very easy. Like a high level language you use a single period to reference the *Pt* field and a second period to access the *x*, *y*, and *z* fields of *point*:

```
      stdout.put( "ThisPt.Pt.x = ", ThisPt.Pt.x, nl );
      stdout.put( "ThisPt.Pt.y = ", ThisPt.Pt.y, nl );
      stdout.put( "ThisPt.Pt.z = ", ThisPt.Pt.z, nl );
       .
       .
       .
   mov( eax, ThisPt.Color );
```

You can also declare *arrays* as record fields. The following record creates a data type capable of representing an object with eight points (e.g., a cube):

```
type
   Object8:
      record
         Pts:        point[8];
         Color:      dword;
      endrecord;
```

There are two common ways to nest record definitions. As noted earlier in this section, you can create a record type in a TYPE section and then use that type name as the data type of some field within a record (e.g., the *Pt:point* field in the *Pixel* data type above). It is also possible to declare a record directly within another record without creating a separate data type for that record; the following example demonstrates this:

```
type
   NestedRecs:
      record
         iField: int32;
         sField: string;
         rField:
            record
               i:int32;
               u:uns32;
            endrecord;
         cField:char;
      endrecord;
```

Generally, it's a better idea to create a separate type rather than embed records directly in other records, but nesting them is perfectly legal and a reasonable thing to do on occasion.

18.7.5 Controlling Field Offsets Within a Record

By default, whenever you create a record, most assemblers automatically assign the offset zero to the first field of that record. This corresponds to records in a high level language and is the intuitive default condition. In some instances, however, you may want to assign a different starting offset to the first field of the record. The HLA assembler provides a mechanism that lets you set the starting offset of the first field in the record.

The syntax to set the first offset is

```
name:
    record := startingOffset;
        << Record Field Declarations >>
    endrecord;
```

Using the syntax above, the first field will have the starting offset specified by the *startingOffset int32* constant expression. Since this is an *int32* value, the starting offset value can be positive, zero, or negative.

One circumstance where this feature is invaluable is when you have a record whose base address is actually somewhere within the data structure. The classic example is an HLA string. An HLA string uses a record declaration similar to the following:

```
record
   MaxStrLen: dword;
   length: dword;
   charData: char[xxxx];
endrecord;
```

However, HLA string pointers do not contain the address of the *MaxStrLen* field; they point at the *charData* field. The str.strRec record type found in the HLA Standard Library Strings module uses a record declaration similar to the following:

```
type
   strRec:
      record := -8;
         MaxStrLen: dword;
         length:    dword;
         charData:  char;
      endrecord;
```

The starting offset for the *MaxStrLen* field is -8. Therefore, the offset for the *length* field is -4 (four bytes later) and the offset for the *charData* field is zero. Therefore, if EBX points at some string data, then "(type str.strRec [ebx]).length" is equivalent to "[ebx-4]" since the *length* field has an offset of -4.

18.7.6 Aligning Fields Within a Record

To achieve maximum performance in your programs, or to ensure that your records properly map to records or structures in some high level language, you will often need to be able to control the alignment of fields within a record. For example, you might want to ensure that a *dword* field's offset is an even multiple of four. You use the ALIGN directive in a record declaration to do this. The following example shows how to align some fields on important boundaries:

```
type
   PaddedRecord:
      record
         c: char;
            align(4);
         d: dword;
         b: boolean;
            align(2);
         w: word;
      endrecord;
```

Whenever HLA encounters the ALIGN directive within a record declaration, it automatically adjusts the following field's offset so that it is an even multiple of the value the ALIGN directive specifies. It accomplishes this by increasing the offset of that field, if necessary. In the example above, the fields would have the following offsets: *c*:0, *d*:4, *b*:8, *w*:10.

If you want to ensure that the record's size is a multiple of some value, then simply stick an ALIGN directive as the last item in the record declaration. HLA will emit an appropriate number of bytes of padding at the end of the record to fill it in to the appropriate size. The following example demonstrates how to ensure that the record's size is a multiple of four bytes:

```
type
   PaddedRec:
      record
         << some field declarations >>

         align(4);

      endrecord;
```

Be aware of the fact that the ALIGN directive in a RECORD only aligns fields in memory if the record object itself is aligned on an appropriate boundary. Therefore, you must ensure appropriate alignment of any record variable whose fields you're assuming are aligned.

If you want to ensure that all fields are appropriately aligned on some boundary within a record, but you don't want to have to manually insert ALIGN directives throughout the record, HLA provides a second alignment option to solve your problem. Consider the following syntax:

```
type
  alignedRecord3 :
    record[4]
      << Set of fields >>
    endrecord;
```

The "[4]" immediately following the RECORD reserved word tells HLA to start all fields in the record at offsets that are multiples of four, regardless of the object's size (and the size of the objects preceeding the field). HLA allows any integer expression that produces a value in the range 1..4096 inside these parenthesis. If you specify the value one (which is the default), then all fields are packed (aligned on a byte boundary). For values greater than one, HLA will align each field of the record on the specified boundary. For arrays, HLA will align the field on a boundary that is a multiple of the array element's size. The maximum boundary HLA will round any field to is a multiple of 4096 bytes.

Note that if you set the record alignment using this syntactical form, any ALIGN directive you supply in the record may not produce the desired results. When HLA sees an ALIGN directive in a record that is using field alignment, HLA will first align the current offset to the value specified by ALIGN and then align the next field's offset to the global record align value.

Nested record declarations may specify a different alignment value than the enclosing record, e.g.,

```
type
   alignedRecord4 : record[4]
      a:byte;
      b:byte;
      c:record[8]
         d:byte;
         e:byte;
      endrecord;
      f:byte;
      g:byte;
   endrecord;
```

In this example, HLA aligns fields a, b, f, and g on dword boundaries, it aligns d and e (within c) on eight-byte boundaries. Note that the alignment of the fields in the nested record is true only within that nested record. That is, if c turns out to be aligned on some boundary other than an eight-byte boundary, then d and e will not actually be on eight-byte boundaries; they will, however be on eight-byte boundaries relative to the start of c.

In addition to letting you specify a fixed alignment value, HLA also lets you specify a minimum and maximum alignment value for a record. The syntax for this is the following:

```
type
   recordname : record[maximum : minimum]
      << fields >>
   endrecord;
```

Whenever you specify a maximum and minimum value as above, HLA will align all fields on a boundary that is at least the minimum alignment value. However, if the object's size is greater than the minimum value but less than or equal to the maximum value, then HLA will align that particular field on a boundary that is a multiple of the object's size. If the object's size is greater than the maximum size, then HLA will align the object on a boundary that is a multiple of the maximum size. As an example, consider the following record:

```
type
   r: record[ 4:1 ];
      a:byte;           // offset 0
      b:word;           // offset 2
      c:byte;           // offset 4
      d:dword[2];       // offset 8
      e:byte;           // offset 16
      f:byte;           // offset 17
      g:qword;          // offset 20
   endrecord;
```

Note that HLA aligns g on a dword boundary (not qword, which would be offset 24) since the maximum alignment size is four. Note that since the minimum size is one, HLA allows the f field to be aligned on an odd boundary (since it's a byte).

If an array, record, or union field appears within a record, then HLA uses the size of an array element or the largest field of the record or union to determine the alignment size. That is, HLA will align the field without the outermost record on a boundary that is compatible with the size of the largest element of the nested array, union, or record.

HLA sophisticated record alignment facilities let you specify record field alignments that match that used by most major high level language compilers. This lets you easily access data types used in those HLLs without resorting to inserting lots of ALIGN directives inside the record.

18.7.7 Using Records/Structures in an Assembly Language Program

In the "good old days" assembly language programmers typically ignored records. Records and structures were treated as unwanted stepchildren from high-level languages, that weren't necessary in "real" assembly language programs. Manually counting offsets and hand-coding literal constant offsets from a base address was the way "real" programmers wrote code in early PC applications. Unfortunately for those "real programmers", the advent of sophisticated operating systems like Windows and Linux put an end to that nonsense. Today, it is very difficult to avoid using records in modern applications because too many API functions require their use. If you look at typical Windows and Linux include files for C or assembly language, you'll find hundreds of different structure declarations, many of which have dozens of different members. Attempting to keep track of all the field offsets in all of these structures is out of the question. Worse, between various releases of an operating system (e.g., Linux), some structures have been known to change, thus exacerbating the problem. Today, it's unreasonable to expect an assembly language programmer to manually track such offsets - most programmers have the reasonable expectation that the assembler will provide this facility for them.

18.7.8 Implementing Structures in an Assembler

Unfortunately, properly implementing structures in an assembler takes considerable effort. A large number of the "hobby" (i.e., non-commercial) assemblers were not designed from the start to support sophisticated features such as records/structures. The symbol table management routines in most assemblers use a "flat" layout, with all of the symbols appearing at the same level in the symbol table database. To properly support structures or records, you need a hierarchical structure in your symbol table database. The bad news is that it's quite difficult to retrofit a hierarchical structure over the top of a flat database (i.e., the symbol "hobby assembler" symbol table). Therefore, unless the assembler was originally designed to handle structures properly, the result is usually a major hacked-up kludge.

Four assemblers I'm aware of, MASM, TASM, OPTASM, and HLA, handle structures well. Most other assemblers are still trying to simulate structures using a flat symbol table database, with varying results.

Probably the first attempt people make at records, when their assembler doesn't support them properly, is to create a list of constant symbols that specify the offsets into the record. Returning to our first example (in HLA):

```
type
   student:
            record
               Name:       string;
               Major:      int16;
               SSN:        char[12];
               Midterm1:   int16;
               Midterm2:   int16;
               Final:      int16;
               Homework:   int16;
               Projects:   int16;
            endrecord;
```

One attempt might be the following:

```
const
    Name := 0;
    Major := 4;
    SSN := 6;
    Midterm1 := 18;
    Midterm2 := 20;
    Final := 22;
    Homework := 24;
    Projects := 26;
    size_student := 28;
```

With such a set of declarations, you could reserve space for a student "record" by reserving "size_student" bytes of storage (which almost all assemblers handle okay) and then you can access fields of the record by adding the constant offset to your base address, e.g.,

```
static
    John : byte[ size_student ];
      .
      .
      .
    mov( John[Midterm1], ax );
```

There are several problems with this approach. First of all, the field names are global and must be globally unique. That is, you cannot have two record types that have the same fieldname (as is possible with the assembler supports true records). The second problem, which is fundamentally more problematic, is the fact that you can attach these constant offsets to any object, not just a "student record" type object. For example, suppose "ClassAverage" is an array of words, there is nothing stopping you from writing the following when using constant equate values to simulate record offsets:

```
    mov( ClassAverage[ Midterm1 ], ax );
```

Finally, and probably the most damning criticism of this approach, is that it is very difficult to maintain code that accesses structures in this manner. Inserting fields into the middle of a record, changing data types, and coming up with globally unique names can create all sorts of problems. Many high-level language programmers who've tried to learn assembly language have given up after discovering that they had to maintain records in this fashion in an assembly language program (too bad they didn't start off with a reasonable assembler that properly supports structures).

Manually maintaining all the constant offsets is a maintenance nightmare. So somewhere along the way, some assembly language programmers figured out that they could write macros to handle the declaration of constant offsets for them. For example, here's how you could do this in an HLA program:

```
program t;


#macro struct( _structName_, _dcls_[] ):
    _dcl_, _id_, _type_, _colon_, _offset_;

    ?_offset_ := 0;
    ?_dcl_:string;
    #for( _dcl_ in _dcls_ )

        ?_colon_ := @index( _dcl_ , 0, ":" );
        #if( _colon_ = -1 )

            #error
            (
               "Expected <id>:<type> in struct definition, encountered: ",
                _dcl_
            )

        #else

            ?_id_ := @substr( _dcl_, 0, _colon_ );
            ?_type_ := @substr( _dcl_, _colon_+1, @length( _dcl_ ) -
_colon_ );
            ?@text( _id_ ) := _offset_;
            ?_offset_ := _offset_ + @size( @text( _type_ ));

        #endif;

    #endfor
    ?_structName_:text := "byte[" + @string( _offset_ ) + "]";

#endmacro

struct( threeItems, i:byte, j:word, k:dword )

static
    aStruct: threeItems;
```

```
begin t;

    mov( (type byte aStruct[i]), al );
    mov( (type word aStruct[j]), ax );
    mov( (type dword aStruct[k]), eax );


end t;
```

The "struct" macro expects a set of valid HLA variable declarations supplied as macro arguments. It generates a set of constants using the supplied variable names whose offsets are adjusted according to the size of the objects previously appearing in the list. In this example, HLA creates the following equates:

```
i = 0
j = 1
k = 3
```

This declaration also creates a "data type" named "threeItems" which is equivalent to "byte[7]" (since there are seven bytes in this record) that you may use to create variables of type "threeItems", as is done in this example.

Creating structures with macros solves one of the three major problems: it makes it easier to maintain the constant equates list, as you do not have to manually adjust all the constants when inserting and removing fields in a record. This does not, however, solve the other problems (particularly, the global identifier problem).

While fancier macros could be written, macros that generate identifiers like "objectname_fieldName" that help solve the globally unique problem, the bottom line is that these hacks begin to fail when you attempt to declare nested records, arrays within records, and arrays of records (possibly containing nested records and arrays of records). The bottom line is this: assemblers that don't properly support structures are going to have problems when you've got to work with data structures from high-level languages (e.g., OS API calls, where the OS is written in C, such as Windows and Linux). You're much better off using an assembler that fully supports structures (and other advanced data types) if you need to use structures in your programs.