Tutorial 1: The Basics

This tutorial assumes that the reader knows how to use HLA. If you're not familiar with HLA, download it and study the text inside the package before going on with the tutorial. Good. You're now ready. Let's go!

Theory:

Win32 programs run in protected mode which is available since 80286. But 80286 is now history. So we only have to concern ourselves with 80386 and its descendants. Windows runs each Win32 program in separated virtual space. That means each Win32 program will have its own 4 GB address space. However, this doesn't mean every win32 program has 4GB of physical memory, only that the program can address any address in that range. Windows will do anything necessary to make the memory the program references valid. Of course, the program must adhere to the rules set by Windows, else it will cause the dreaded General Protection Fault. Each program is alone in its address space. This is in contrast to the situation in Win16. All Win16 programs can *see* each other. Not so under Win32. This feature helps reduce the chance of one program writing over other program's code/data.

Memory model is also drastically different from the old days of the 16-bit world. Under Win32, we need not be concerned with memory model or segments anymore! There's only one memory model: Flat memory model. There's no more 64K segments. The memory is a large continuous space of 4 GB. That also means you don't have to play with segment registers. You can use any segment register to address any point in the memory space. That's a GREAT help to programmers. This is what makes Win32 assembly programming as easy as C.

When you program under Win32, you must know some important rules. One such rule is that, Windows uses esi, edi, ebp and ebx internally and it doesn't expect the values in those registers to change. So remember this rule first: if you use any of those four registers in your callback function, don't ever forget to restore them before returning control to Windows. A callback function is your own function which is called by Windows. The obvious example is the windows procedure. This doesn't mean that you cannot use those four registers, you can. Just be sure to restore them back before passing control back to Windows.

Content:

Here's the skeleton program. If you don't understand some of the codes, don't panic. I'll explain each of them later.

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

const
  <Your compile-time constants go here>

type
  <Your type declarations go here>

static
```

```
  <Your initialized data>

storage
  <Your uninitialized data>

readonly
  <run-time constants and read-only data goes here>

<< Your procedures would normally go here >>

begin YourProgramName;

  <Your code goes here>

end YourProgramName;

That's all! Let's analyze this skeleton program.
```

program YourProgramName;

HLA supports two types of source files: programs and units. Programs have a *main program* that executes when the program loads into memory. Units are simply a collection of data and procedures that you must link with other code. An executable file consists of exactly one program and zero or more units linked together. For the time being we will only write HLA code in program files. See the HLA documentation for more details about units.

#include("stdlib.hhf")

This statement tells the HLA compiler to include the HLA Standard Library Header files. Depending on your specific application, you may or may not need this statement. Some applications will require the inclusion of other header files (for example, to define the win32 API functions and constants).

The const Section

The **const** section is where you defined named constants for use in your program. Objects you define in the **const** section do not take up any space at run-time (other than as immediate operands of other machine instructions). HLA supports a second constant declaration section, the **val** section, but we will ignore that declaration section here. See the HLA documentation for more details if you're interested. The **const** section is optional and is absent in many programs.

Example **const** section:

```
const
  MaxCnt := 10;
  pi := 3.14159;
  HW := "Hello World";
  AllOnes:dword := $FFFF_FFFF;
```

The type Section

You may define your own data types in the **type** section. Like the **const** section, the **type** section does not reserve any storage for data in your program. Objects you define in the **type** section are merely templates for the actual variable declarations that come later. Also like the **const** section, the **type** section is optional and may not be present in many HLA applications.

Sample **type** declaration section:

```
type
  integer: int32;
  point:  record
             x:uns16;
             y:uns16;
          endrecord;
```

The static Section

The **static** section is where you declare initialized variable objects. Actually associating an initial value with each variable you declare is optional. If you do not specifically assign a value to an object, HLA will initialize it with zero bits. The **static** section is optional, but this is the most common place where programmer declare their main program variables, so this section commonly appears in HLA programs. Objects you declare in the static section will consume memory throughout the execution lifetime of your program; these objects will also consume space in the executable file that HLA produces.

Sample **static** section:

```
static
  s:string := “Hello World”;
  counter:uns32 := 0;
  index:int32;
  chr:char;
  b:byte := $5F;
```

The storage Section

The **storage** section is where you can declare uninitialize variable objects. Technically, Microsoft’s MASM and LINK programs (which HLA uses to generate actual machine code) don’t support uninitialized data sections. Therefore, HLA simply initializes all objects you declare in a **storage** section to all zero bits. Effectively, the **static** and **storage** sections are identical except that the **storage** section doesn’t let you assign an initial value to a **storage** variable.

Example:

```
storage
  u_str: string;
  i:int32;
  c:char;
  d:dword;
```

The readonly Section

Objects you declare in the **readonly** section must have an initial value. This is because HLA will load all **readonly** objects into a write-protected section of memory when the program runs. Any attempt to write to data you declare in a **readonly** section will generate a memory access exception at run-time. The **readonly** section is a great place to put data tables and other objects whose values do not change at run-time. Like the **const** section, objects you declare in the **readonly** section do not change while the program executes. Unlike the **const** section, however, objects you declare in the **readonly** section do consume memory at run time. Like the other sections, the **readonly** section is optional and most programmers don't bother including a **readonly** section in their programs.

Example:

```
readonly
   ConstStr: string := "Hello World!";
   dataTable: byte[8] := [0,1,2,3,4,5,6,7];
   jmpTable: dword[4] := [&label1, &label2, &label3, &label4 ];
```

The data and var Sections

HLA has two additional variable declaration sections, **var** and **data**. We will not consider these sections in this particular tutorial. See the HLA documentation for more details.

Section Organization

The sections may appear in any order between the **program** and **begin** statements in the source file. All of the sections are optional and, in fact, may appear multiple times. Unless you have good reason to do otherwise, though, you should arrange these section in the order above.

Executable Code

There are two places where actual machine instructions may appear in your programs: in procedures (and other procedure-like modules like iterators and methods) and in the main program. The main program appears between the "begin YourProgramName" and "end YourProgramName" clauses in the program above. Procedures are optional but an HLA program must have a main program section (even if the body of the main program is empty). Note that an empty main program body simply returns control back to windows when the program terminates execution. For the time being we will ignore the presence of procedures in an HLA program and write most of our code in the main program section.

Parameter Passing Conventions under Win32

The C calling convention passes parameters from right to left, that is , the rightmost parameter is pushed first. The caller is responsible for balancing the stack frame after the call. For example,

in order to call a function named foo(int first_param, int second_param, int third_param) in C calling convention the HLA instructions will look like this:

```
push( third_param );
push( second_param );
push( first_param );
call foo;
add( 12, esp );
```

The PASCAL calling convention is the reverse of the C calling convention. It passes parameters from left to right and the callee is responsible for the stack balancing after the call.

Win16 adopts the PASCAL convention because it produces smaller programs. The C convention is useful when you don't know how many parameters will be passed to the function as in the case of **wsprintf**(). In the case of **wsprintf**(), the function has no way to determine beforehand how many parameters will be pushed on the stack, so it cannot do the stack balancing.

STDCALL is the hybrid of the C and PASCAL convention. It passes parameters from right to left but the callee is responsible for stack balancing after the call.Win32 platforms use STDCALL exclusively except in one case: wsprintf(). You must use C calling convention with **wsprintf**().

HLA supports a high-level procedure call syntax. However, HLA's high level procedure call syntax supports only the PASCAL calling convention. This means that HLA pushes the parameters in the reverse order that the win32 API routines expect them. There is a simple solution to this problem: reverse the parameter declarations in the procedure prototypes. For the **foo** procedure mentioned above, you'd use a prototype like the following:

```
procedure foo( third_param:int32; second_param:int32; first_param:int32);
external;
```

Another minor problem between the STDCALL and PASCAL calling sequences is the fact that the STDCALL mechanism inserts an underscore before the procedure name while the PASCAL calling mechanism does not. Fortunately, HLA's **external** directive syntax provides a way around this problem: it lets you specify the exact external name of the routine as a string object:

```
procedure foo( third_param:int32; second_param:int32; first_param:int32);
external( "_foo" );
```

So between reversing the parameters in the procedure's prototype and explicitly specifying the external name, you can overcome the technical problems associated with the win32 API calls. The only remaining problem is a psychological rather than a technical problem. There are a tremendous number of win32 API routines available to you. So many, in fact, that it is unlikely you will memorize them all (or even a fair number of them) and remember the order of the parameters. As such, you will be looking up many functions you call in order to verify the number, type, and order of parameters. Unfortunately, all the documentation you're going to be reading about this will assume that you're using the STDCALL calling convention. Therefore, those examples will have their parameter lists reversed (compared to the way you must pass the parameters in the HLA code). At best, this is annoying; at worst, it will cause you to inject lots of extra defects into your code by inadvertently swapping parameters.

A better solution is to use HLA's macro facilities to reverse the parameters for you. This way you can create a set of procedure/function prototypes that have the same "user interface" as the

standard win32 API routines yet use the HLA high level calling syntax that supports only the Pascal calling convention. Consider the following modification to the **foo** prototype from above:

```
procedure foo_proc( third_param:int32; second_param:int32; first_param:int32);
external( "_foo" );

macro foo( first_param, second_param, third_param );

  foo_proc( third_param, second_param, first_param )

endmacro;
```

Now when you call foo via the invocation "foo(1, 2, 3);" HLA will expand the **foo** macro to produce the actual call "foo_proc(3, 2, 1);" that automatically reverses the parameters for you. By supplying these macros for each win32 API routine you call, you can invoke those routines using the same apparent calling sequence as the STDCALL interface.

Note: you should not use this technique for procedures you write yourself in HLA. If you are the only one who calls your procedures (i.e., you aren't writing Windows callback routines), then stick with the PASCAL calling convention: it's more efficient.

Writing a Simple Console Application

Although this tutorial series is geared towards writing true Windows applications, a good place to start is with a standard console application. HLA generates console applications by default (HLA was intended for teaching assembly language and console applications are the easiest to write, hence HLA's default choice). The following simple program is a complete, standalone, console application that implements the familiar "Hello World" application.

```
program helloWorld;
#include( "stdlib.hhf" )
begin helloWorld;

    stdout.put( "Hello World", nl );

end helloWorld;
```

The logic of this program should be fairly obvious to anyone above the absolute beginner level, so there is no need to discuss it farther.

To compile this program (assuming it's named "hw.hla") you'd use the following command in an win32 command prompt window:

```
c:> hla hw.hla
```

The command above will produce an executable file that displays "Hello World" (and a new line) when you execute the program from a command window prompt. This tutorial will not consider console applications any farther since the HLA documentation covers those in great detail. About the only thing worth noting is the fact that many of the HLA standard library routines (e.g., stdout.put) are only legal in console applications. In particular, you should avoid calling any routine that begins with *stdout*, *stdin*, and *console* in an application that is not a console app. Certain other library functions may also present problems, see the HLA Standard Library documentation

and HLA Standard Library source code if you have any questions about the suitability of routines in the HLA Standard Library for standard Windows applications.