
Exercising Complete Control with HLA

Note: This document was written with Windows and Linux programmers in mind. Most of the examples are Windows examples; this document provides Linux-specific examples only when there is a major difference in the way the compiler operates under Windows versus Linux.

A common complaint I get about HLA is that it "hides the machine from the user and generates tons of code behind the programmer's back." This is usually followed by something along the lines of "true assemblers don't do that." The truth is that the HLA compiler generates very little extraneous code and there is actually only a little bit of overhead in an HLA program. Part of the confusion stems from the fact that many users think of calls to the HLA Standard Library as part of "the HLA language." For example, many programmers who see the ubiquitous "Hello World" program written in HLA automatically assume that the "stdout.put" library call is part of the language and demonstrates the "bloat" that exists in HLA. Obviously, such a belief is erroneous since anyone could write their own I/O routines and replace the call to stdout.put with their code and HLA would be none the wiser.

However, to say that there is no code overhead or to say that HLA doesn't emit code behind the programmers back isn't true either. HLA was designed to make learning assembly language programming easy for beginners. Therefore, HLA does automatically generate some code to help out beginners. Fortunately, it's easy to turn this extra code generation off and have HLA only generate code that you've written. The purpose of this document is to describe how to turn off all extraneous code generation so you, the advanced assembly programmer, can exercise absolute control over the machine code that HLA generates.

By the way, it goes without saying that if you intend to exercise absolute control over your machine code, you won't achieve this if you're using HLA's high-level control statements and certain other high level features that HLA provides. Fortunately, none of those high level features (that generate code behind your back) are necessary in an HLA program. You can easily avoid the extraneous code generation by simply not using those high level control statements in your assembly programs. Since HLA allows you to write "pure assembly code" without any high level features, and there is nothing forcing you to use those statements, using high level control statements as an example of HLA's bloat is illogical. If you don't want such bloat in your assembly programs, don't use those statements!

1.1 Why is "Hello World" So Big?

Whenever someone first installs the HLA system, the first thing they almost always do is write the "Hello World" program and compile and execute this program¹. The second thing they do is take a look at the size of the ".EXE" file that it produces (under Windows). The third thing they do is email me and ask why such a simple program is so large (compiled under HLA v1.38, the typical "Hello World" program that includes and uses "stdlib.hhf" is 16,384 bytes long on the disk²). This is, in fact, larger than the "Hello World" program that some HLL compilers produce. Why is the code so large?

Actually, as it turns out, the code is not large at all. The disk file is large because of the PE/COFF executable format that Windows uses. For performance reasons, the PE/COFF executable format allocates disk space in blocks of 4096 bytes for each *section* in a machine code program. Every executable program starts with a header section. The header section contains information about the rest of the executable file. Windows does not load the header into memory as part of the program. Therefore, this 4,096 byte block on the disk doesn't actually take up space in memory, just on the disk.

If the machine code program makes any Windows API calls (and generally it must make at least one, the call to ExitProcess that returns control to Windows), then the PE/COFF format will require another section

1. The main reason they do this is because the HLA installation instructions use the Hello World program as an example to test out the proper operation of the HLA system.

2. Under version of HLA before v1.38, the file was often 20K or even 24K! There were some optimizations made to the system between HLA v1.37 and HLA v1.38 to reduce the size by 8K.

to hold pointers to the Win32 API functions. This 4,096-byte block contains enough space for up to 1,024 different API functions and other objects. Since most applications typically manipulate fewer than 1,024 system objects, it rarely takes more than one page of data (4,096 bytes) to hold all this information. If you're keeping track, we've just consumed 8,192 bytes and we still haven't actually processed any part of the actual Hello World program.

The next section found in the typical "Hello World" program is the code section. This is the section that contains the actual machine instructions. Even though the actual machine code is only a few hundred bytes, the PE/COFF file format sets aside 4,096 bytes for this block (filling the unused bytes with zeros). So now we're up to 12K.

The last section found in the typical HLA "Hello World" program is the data section. Although the "Hello World" program itself doesn't use any variables, some of the library routines that `stdout.put` winds up calling need static variables. Once again, although there are only a few bytes of actual variable data, the PE/COFF format sets aside 4,096 bytes for the data section. This fourth section (each 4,096 bytes) bumps the total up to 16,384 bytes.

So the truth is, the OS sets aside 8K bytes for its own purposes and the HLA program generates two sections, each 4K long, but only uses a couple of hundred bytes out of this 8K block. Therefore, the size of the executable is strictly due to the PE/COFF format that Windows uses and has little or nothing to do with the HLA language or compiler.

At this point, you're probably protesting "but other assemblers allow you to write short 'Hello World' programs; why isn't this possible in HLA?" Well, the truth is, it is possible to do this in HLA by pulling some very specific tricks. In particular, if you run the linker manually, you can tell it to pack the different sections together, thus producing an executable file that is well under 1K in length. However, these tricks are very specific to the "Hello World" code and may not work in general. Hence, HLA doesn't bother with such trickery and, instead, provides a generic set of linker commands that will work with all programs.

Of course, you're probably from Missouri and you're saying "show me" at this point. It's easy enough to claim that HLA can produce a short "Hello World" program, it's something entirely different to actually write the code. Feat not, we'll get to the example soon enough in this document. In the meantime, I'd like to explain why any attempt to reduce the size of the "Hello World" program from 16K to something under 1K is a complete waste of your time (other than for the satisfaction of knowing you can do it). The argument I will use is not that today's machines have gobs of memory and trying to save memory is not worthwhile. This is absolutely false. It's always a good idea to try and write programs that are as compact as you can reasonably develop. No, the real reason for not going hog-wild when attempting to write the world's shortest "Hello World" program under Windows is that you're not saving anywhere near the memory you think that you are.

Let's take a look at something I said earlier: the typical HLA 'Hello World' program consumes about 16K of space on the disk (which turns out to be about 12K in memory since the PE/COFF header information doesn't remain resident in memory). If we can shrink the executable from 16K (or 12K) down to less than 1K, we're seeing a 16:1 (or 12:1) reduction in space. This seems very impressive. In reality, however, the system always allocates memory in 4K blocks (the x86 MMU page size). Therefore, even if you were to shrink the code for the 'Hello World' program down to one byte, it would still require 4K of memory for the code section since the system always allocates memory in 4K quanta. Since Windows makes the code section read-only, if your program has any variables, the system must put them in their own page of memory (so they have read/write access). This requires a second 4K page, so we're up to at least 8K. So a two-byte program that has one byte of code and one byte of data is going to consume a minimum of 8K of real memory. Therefore, any attempt to reduce the code size below 4K and any attempt to reduce the data requirements below the 4K mark is a complete waste of time since the system will allocate memory pages in 4K blocks³.

Now you might think that reducing 16K or 12K to 4K is still a valid goal to attain. This, however, completely ignores the fact that Windows assigns a minimum of one megabyte of stack space and one megabyte of heap space to a typical application. Plus, of course, Windows maintains lots of other information about

3. It is possible, with a lot of linker tricks, to place the code and data in the same page of memory if you make that page writable. However, it's very dangerous to make a page in memory writable if it contains code. Therefore, this document will ignore that possibility.

your process in memory while the process is running. The bottom line is that if you save 4K or even 8K on your 'Hello World' program, from a percentage point of view you actually wind up saving very little real memory.

The other thing to consider is that the 'Hello World' is a trivial application. How small you can make this application is irrelevant since no one really runs this application anyway. For the reasons outlined above, 'Hello World' makes an incredibly poor benchmark of compiler overhead since the program is so tiny that most of the bytes in the executable file are simply padding. Indeed, an interesting thing to note is that you can make the (16K) 'Hello World' program a whole lot more sophisticated by adding lots of additional features and the size (on the disk) of the program won't grow any larger. The disk file size won't expand until you cross a 4K boundary in the code or data sections of the program (at which point, the size will grow by 4K bytes). So the intuitive conclusion that "if 'Hello World' is 16K, imagine how big a slightly more sophisticated program will be" simply doesn't apply here. You can make the 'Hello World' program quite a bit more sophisticated before it grows by even one byte).

Therefore, the goal of a Windows/HLA programmer should not be "How small can I make this program?" but rather, "how much more functionality can I pack into this program before it crosses a 4K boundary and the executable size grows?"

By the way, don't worry, I'll show you how to write that small 'Hello World' program I promised earlier. We've just got to learn some stuff about HLA first...

1.2 Overhead Present in an HLA Program

Many people naturally assume that the HLA compiler introduces a lot of extra code into the assembly file it produces. They base their beliefs on several things including the sophistication of the HLA Standard Library (the HLA compiler must call some code to do some initialization required by the Standard Library, just like C), the sophistication of the data structures, and because of HLA's support for high level control structures. This, however, is a misconception. Although the HLA compiler does emit some initialization code when it compiles an HLA program, this code is actually quite small; it's probably under a hundred bytes, not thousands of bytes or even hundreds of bytes. So let's get that misconception out of the way real fast; to prove this issue, we'll compile an empty HLA program and take a look at the MASM and Gas code it produces.

1.2.1 The "empty" Program

Conceptually, the simplest program we can write (and execute) is the empty program. The empty program compiles and runs, but just immediately returns to Windows without doing much of anything else. One would hope that the empty program would produce the smallest possible executable file size. Here's the empty program in HLA:

```
program t;
begin t;
end t;
```

Program 2.1 The Canonical Empty Program

Here's the MASM code that HLA emits under Windows for the above program:

```
; Assembly code emitted by HLA compiler
; Version 1.38 build 5950 (prototype)
```

```

; HLA compiler written by Randall Hyde
; MASM compatible output

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

offset32      equ      <offset flat:>

        assume fs:nothing
ExceptionPtr__hla_      equ      <(dword ptr fs:[0])>

        include empty.extpub.inc

; $ignore
;(for test purposes)

        .data
        include empty.data.inc

        .data?
        include empty.bss.inc

        .code
        include empty.consts.inc

        .code
        include empty.ro.inc

        .code

HWexcept__hla_      proc      near32
                    jmp      shorthwExcept__hla_
HWexcept__hla_      endp

DfltExHndlr__hla_      proc      near32
                    jmp      shortDfltExcept__hla_
DfltExHndlr__hla_      endp

_HLAMain          proc      near32

; /* Set up the Structured Exception Handler record */

```

```

; /* for this program. */

        call    BuildExcept__hla_
        pushd  0          ; /* No Dynamic Link. */
        mov    ebp, esp    ; /* Pointer to Main's locals */
        push  ebp          ; /* Main's display. */

QuitMain__hla_::
        pushd  0
        call   dword ptr __imp__ExitProcess@4
_HLMain
        endp

        end

```

Program 2.2 MASM Output Code for the Empty Program

The empty.bss.inc, empty.consts.inc, and empty.ro.inc include files are all empty files. They obviously add nothing to the size of the executable file. The empty.extpub.inc file contains some external declarations, these declarations do not directly affect the size of the file (though any calls to these external routines will affect the size, obviously). Finally, the empty.data.inc file contains the following data:

```

MainPgmCoroutine__hla_ label    byte
        dword  offset32 MainPgmVMT__hla_
        dword  00h             ; /* CurrentSP */
        dword  00h             ; /* Pointer to stack */
        dword  00h             ; /* ExceptionContext */
        dword  00h             ; /* Pointer to last caller */
MainPgmVMT__hla_ label    dword
        dword  offset32 QuitMain__hla_

```

Program 2.3 The empty.data.inc Include File

Consider for a moment, the code appearing just before the main program (_HLMain) in the assembly (MASM) file:

```

HWexcept__hla_ proc    near32
        jmp    shorthwExcept__hla_
HWexcept__hla_ endp

DfltExHndlr__hla_ proc near32
        jmp    shortDfltExcept__hla_
DfltExHndlr__hla_ endp

```

Obviously these two jump instructions don't add much code to the executable, but they do jump to some external table, so it's fair to ask about the code associated with shorthwExcept__hla_ and shortDfltExcept__hla_ (these are two HLA Standard Library modules). These two procedures are actually quite small, their source appears in the HLA Standard Library and is duplicated here:

```

; Special module for the default exception handler.
; This wasn't really written in MASM for any particular

```

```

; reason, it could have been written in HLA, as well.

        .586
        .model flat, syscall
        .code

; Default Hardware Exception Handler to execute if
; we get a Windows Exception. This just passes control
; back to the Windows' handler.

        public shorthwExcept__hla_
shorthwExcept__hla_ proc near32
        mov     eax, 1
        ret
shorthwExcept__hla_ endp
        end

```

Program 2.4 The shorthwExcept__hla_ Procedure

```

; Assembly code emitted by HLA compiler
; Version 1.38 build 5948 (prototype)
; HLA compiler written by Randall Hyde
; MASM compatible output

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

offset32 equ <offset flat:>

        assume fs:nothing
ExceptionPtr__hla_ equ <(dword ptr fs:[0])>

        .data
externdef __imp__MessageBoxA@16:dword
externdef __imp__ExitProcess@4:dword
        .code
externdef HWexcept__hla_:near32
externdef _traceLine_:near32
externdef abstract__hla_:near32
externdef Raise__hla_:near32
        public shortDfltExcept__hla_

        .code
L3_DefaultMessage__hla_label byte
        byte "Unhandled exception error."

```

```

        byte    00h
L4_HLAException__hla_ label byte
        byte    "HLA Exception Handler"
        byte    00h

shortDfltExcept__hla_ proc  near32
        pushd   030h
        push   offset32 [L4_HLAException__hla_+0]
        push   offset32 [L3_DefaultMessage__hla_+0]
        pushd   00h
        call   dword ptr [__imp__MessageBoxA@16+0] ;/* MessageBox */
        pushd   00h
        call   dword ptr [__imp__ExitProcess@4+0] ;/* ExitProcess */
xshortDfltExcept__hla__hla_:
shortDfltExcept__hla_ endp

        end

```

Program 2.5 The `shortDfltExcept__hla_` Procedure

If you know anything about machine code, you'll probably realize real quick that these procedures are very small. In fact, there's probably more bytes required for the two exception strings as the actual object code requires. Although I haven't actually counted the bytes, I'd guess that these two procedures and their data are well under 100 bytes, total.

Returning back to the empty program, the main program (`_HLAMain`) for this file contains the following code:

```

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

        call   BuildExcepts__hla_
        pushd   0                ;/* No Dynamic Link. */
        mov    ebp, esp          ;/* Pointer to Main's locals */
        push   ebp              ;/* Main's display. */

QuitMain__hla__:
        pushd   0
        call   dword ptr __imp__ExitProcess@4

```

The call to `BuildExcepts__hla_` and the three instructions that follow are the "overhead" associated with a typical HLA program. The last two instructions return control to the operating system; It's hard to call these two instructions overhead since every Windows program is going to need something like these two instructions (these would only be overhead if the program returns to Windows somewhere else and these last two instructions never execute).

The three instructions following the call above set up the stack frame for the main program. This provides access to the VAR objects found in the main program (there are none, or there would actually be another SUB instruction present above). In some respect, these instructions are pure overhead since there are no automatic (VAR) objects in this program (and HLA sets up the stack frame in order to access automatic variables from the main program). However, we are talking about *three instructions* here that normally execute only once. My assumption is that this isn't an incredible amount of bloat.

That leaves us with the call to the `BuildExcepts__hla_` procedure. This is another HLA Standard Library module that initializes HLA's exception handling system. Here's what the code to the `BuildExcepts__hla_` procedure looks like:

```

; BuildExcepts-
; This function constructs the initial exception frame
; on the stack for the HLA main program.
; Note that this code does some serious "messing around"
; with the stack.

                .586
                .model flat, syscall
                .code
                assume fs:nothing

offset32       equ    <offset flat:>
ExceptionPtr   equ    <(dword ptr fs:[0])>
                extern MainPgmCoroutine__hla_:dword
                extern DfltExHndlr__hla_:near32
                extern HWexcept__hla_:near32

; Default Hardware Exception Handler to execute if
; we get a Windows Exception. This just passes control
; back to the Windows' handler.

                public BuildExcepts__hla_
BuildExcepts__hla_ proc near32
                pop    eax
                push   offset32 DfltExHndlr__hla_
                push   ebp
                push   offset32 MainPgmCoroutine__hla_
                push   offset32 HWexcept__hla_
                push   ExceptionPtr

                mov    ExceptionPtr, esp
                mov    dword ptr MainPgmCoroutine__hla_+12, esp
                jmp    eax
BuildExcepts__hla_ endp
                end

```

Program 2.6 HLA Standard Library BuildExcepts__hla_ Procedure

Again, as you can see, there's not a whole lot of code here. The vast majority of this code simply initializes HLA's exception handing subsystem. You've just seen all the "bloated" code that HLA emits for most programs. You'll see a little bit later than it's even possible to remove all this code from an HLA output file (assuming you can live without exception support or are willing to write the code to support exceptions yourself).

Note, by the way, that if you compile the empty program under Windows, the .EXE file it produces is 16,384 bytes long (HLA v1.38). As noted earlier, this is due to the PE/COFF and linker options, it has nothing to do with the code that HLA produces. One 4K block is needed for the PE/COFF header information, one 4K block is needed for indirect pointers to Win32 API routines (since the code above makes calls to a couple of different Win32 API functions), one 4K block is needed for the data found in the *empty.data.inc* file, and one 4K block is needed to hold the 100 or so bytes of instruction data associated with this program.

1.2.2 The empty Program, Part II

Although the empty program of the previous section is the smallest program we can write that will compile and run, it's not the smallest program we can create with HLA, assuming we don't care if it doesn't run. The smallest possible program you can write with HLA would consist of a UNIT with a single procedure that has no instructions associated with it. The following is such an empty program:

```
unit empty;

  procedure main; @external( "_HLAMain" );
  procedure main; @noframe;
  begin main;
  end main;

end empty;
```

Program 2.7 The "empty2" Program

To properly link and produce an .EXE file without error, an HLA program must have a procedure named "_HLAMain". The external declaration above and the corresponding procedure declaration for main achieves this. Note the presence of the "@noframe" procedure option. This tells HLA to skip any extra code emission for the procedure. Here's the MASM file that the above produces:

```
; Assembly code emitted by HLA compiler
; Version 1.38 build 5950 (prototype)
; HLA compiler written by Randall Hyde
; MASM compatible output

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

offset32    equ <offset flat:>

    assume fs:nothing
ExceptionPtr_hla_ equ <(dword ptr fs:[0])>

    include empty2.extpub.inc

; $ignore
;(for test purposes)

    .data
```

```

include empty2.data.inc

.data?
include empty2.bss.inc

.code
include empty2.consts.inc

.code
include empty2.ro.inc

.code

_HLAMain      proc    near32
x_HLAMain__hla_ :
_HLAMain      endp

end

```

Program 2.8 MASM Output File for the "empty2" Program

For this program, all of the include files are empty, so there's no need to list them here. If you compile this program to an executable, the resulting file is only 4,096 bytes long. This is because there is no code, so we don't need a 4K block associated with the code; there is no data, so we don't need a 4K block associated with the data segment; there is no Win32 API pointer data because we don't make any Win32 API calls. The PE/COFF header information still requires a 4K block, which is why the file is 4,096 bytes long.

Since this program doesn't make any Win32 API calls, it doesn't properly call the Win32 ExitProcess function to return control to Windows. Therefore, if you run this program, it will probably crash. Note that adding the call to the ExitProcess function will add 8K to the size of the executable file since you'll need a few bytes for the PUSH and CALL instructions and you'll need a few bytes of pointer data (the pointer to the ExitProcess function). Therefore, the smallest practical program you can create, using standard linking facilities, is going to be 12K.

1.2.3 Overhead Associated With Exceptions

As you saw earlier in the "empty" example, there is a bit of overhead associated with HLA's exception support. The empty program requires somewhere around 100 bytes of data and code to support exceptions. In fact, if you're sloppy or unaware, HLA's exception handling facilities can require quite a bit more overhead. Consider the following program:

```

program empty3;
#include( "stdlib.hhf" )
begin empty3;
end empty3;

```

Program 2.9 The "empty3" Program

Here's the MASM code that the HLA compiler produces when you compile empty3:

```
; Assembly code emitted by HLA compiler
; Version 1.38 build 5950 (prototype)
; HLA compiler written by Randall Hyde
; MASM compatible output

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

offset32    equ <offset flat:>

    assume fs:nothing
ExceptionPtr_hla_ equ <(dword ptr fs:[0])>

    include empty3.extpub.inc

; $ignore
; (for test purposes)

    .data
    include empty3.data.inc

    .data?
    include empty3.bss.inc

    .code
    include empty3.consts.inc

    .code
    include empty3.ro.inc

    .code

; /* HWexcept_hla_ gets called when Windows raises the exception. */
HWexcept_hla_ proc    near32
    jmp HardwareException_hla_
HWexcept_hla_ endp

DfltExHndlr_hla_ proc near32
    jmp DefaultExceptionHandler
DfltExHndlr_hla_ endp
```

```

_HLAMain      proc    near32

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

        call    BuildExcepts__hla_
        pushd  0      ; /* No Dynamic Link. */
        mov  ebp, esp  ; /* Pointer to Main's locals */
        push  ebp    ; /* Main's display. */

QuitMain__hla_::
        pushd  0
        call  dword ptr __imp__ExitProcess@4
_HLAMain      endp

        end

```

Program 2.10 The "empty3.asm" Output File

You'll have to look close to see a difference between this MASM file and the one for the original "empty" program. Here are the lines that changed:

```

HWexcept__hla_  proc    near32
                jmp  HardwareException__hla_
HWexcept__hla_  endp

DfltExHndlr__hla_  proc  near32
                jmp  DefaultExceptionHandler
DfltExHndlr__hla_  endp

```

Here's the original code:

```

HWexcept__hla_  proc    near32
                jmp    shorthwExcept__hla_
HWexcept__hla_  endp

DfltExHndlr__hla_  proc  near32
                jmp    shortDfltExcept__hla_
DfltExHndlr__hla_  endp

```

The difference between the two is the standard library routines that they call. By the way, if you compile this code to an .EXE file, you'll discover that the .EXE file is exactly the same size as the original code: 16,384 bytes (with HLA v1.38). However, it turns out that there is over 3K of additional data in the empty3 version of this program. What is it that the #include("stdlib.hhf") has done to this code?

Well, the stdlib.hhf header file includes the excepts.hhf header file and the excepts.hhf header file assigns the value "true" to an HLA compile-time variable (@exceptions) that tells HLA whether you want the full exception handling system or an abbreviated version. When the HLA compiler encounters the begin clause associated with the main program, it checks the value of this compile-time variable. If it contains

true, then HLA emits the `HWexcept__hla_` and `DfltExHndlr__hla_` procedures that transfer control to the full exception handler code (`HardwareException__hla_` and `DefaultExceptionHandler__hla_`). If the `@exceptions` compile-time variable contains false (the default value), then HLA emits these procedures with jumps to the shortened versions of these routines. Now the code for the full routines isn't a whole lot larger than the code for the short routines, the big difference is the amount of data. The short exception handler routines print a very short generic message (the same message for all exceptions) if they wind up getting invoked. The full routines print a descriptive message that varies by the actual exception the system raises. Therefore, the full version of the exception handling code has this really big string array and all the data associated with that array is what consumes the better than 3K of additional space that the *empty3* program requires.

Since the `@exceptions` variable is a compile-time variable you can set during compilation, you can force HLA to use the shortened default exception handlers, even if you've included `stdlib.hhf` or `excepts.hhf`, by simply setting `@exceptions` to false prior to the `begin` clause of the main program, e.g.,

```
program empty3;
#include( "stdlib.hhf" )
?@exceptions := false;
begin empty3;
end empty3;
```

Program 2.11 Modified 'empty3' Program That Uses the Short Exception Code

If you don't really need, or care about, informative exception handling in your code, and you're including the *excepts.hhf* header file (or some other header file that indirectly includes *excepts.hhf*, and this includes many of the Standard Library header files), then you can trim the size of your program down a bit by setting `@exceptions` to false prior to the `begin` clause of your main program. Note, however, that having nice descriptive messages is really great when an exception actually occurs; so it's probably a good idea to use the full exception handling package when you're testing and debugging your code. Then set `@exceptions` to false before creating your production code to shave 3K off the executable's size.

Also note that you cannot trap any hardware exceptions (e.g., divide by zero) when using the short exception handler. If you want to be able to trap hardware exceptions but you don't want the overhead of the exception string messages you've got a couple of choices: (1) implement Windows structured exception handling yourself (difficult) or (2) grab the sources to the exception handling library code and remove all the message strings. Generally, 3K is such a small amount that it isn't worth the effort to try and shave this data from your code.

Later, this document will discuss the overhead associated with HLA's high level control statements. But as long as we're on the subject of exceptions, it's probably worthwhile to explore the cost of the HLA `raise` and `try..endtry` statements. Here's a sample HLA program that exercises these statements and the corresponding MASM code:

```
program ExceptsDemo;
begin ExceptsDemo;

    #asm ;raise stmt #endasm
    raise( 1 );

    #asm ;try stmt #endasm

    try

        mov( 0, al );
```

```

#asm ;unprotected stmt #endasm

unprotected

    mov( 1, al );

#asm ;exception( 1 ) stmt #endasm

exception( 1 )

    mov( 2, al );

#asm ;exception( 2 ) stmt #endasm

exception( 2 )

    mov( 3, al );

#asm ;anyexception stmt #endasm

anyexception

    mov( 4, al );

#asm ;endtry stmt #endasm

endtry;
mov( 5, al );

end ExceptsDemo;

```

Program 2.12 Sample HLA Program to Demonstrate Exceptions

```

; Assembly code emitted by HLA compiler
; Version 1.38 build 5950 (prototype)
; HLA compiler written by Randall Hyde
; MASM compatible output

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

offset32    equ <offset flat:>

    assume fs:nothing
ExceptionPtr__hla_ equ <(dword ptr fs:[0])>

    include t.extpub.inc

```

```

;$ignore
;(for test purposes)

    .data
    include t.data.inc

    .data?
    include t.bss.inc

    .code
    include t.consts.inc

    .code
    include t.ro.inc

    .code

HWexcept__hla_ proc    near32
    jmp shorthwExcept__hla_
HWexcept__hla_ endp

DfltExHndlr__hla_ proc near32
    jmp shortDfltExcept__hla_
DfltExHndlr__hla_ endp

_HLAMain      proc    near32

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

    call    BuildExcepts__hla_
    pushd  0      ; /* No Dynamic Link. */
    mov  ebp, esp ; /* Pointer to Main's locals */
    push  ebp    ; /* Main's display. */

; /*#asm*/

; raise stmt ; /*#endasm*/

    mov  eax, 1
    jmp  Raise__hla_

; /*#asm*/

; try stmt ; /*#endasm*/

    push  offset32 L2_exception__hla_
    push  ebp

```

```

        mov ebp, ExceptionPtr_hla_
        push    dword ptr [ebp+8]
        mov ebp, [esp+4]
        push    offset32 HWexcept_hla_
        push    dword ptr [ExceptionPtr_hla_]
        mov dword ptr [ExceptionPtr_hla_], esp
        mov al, 0

; /*#asm*/

;unprotected stmt ; /*#endasm*/

        mov esp, ExceptionPtr_hla_ ; /* Unwind stack. */
        pop ExceptionPtr_hla_ ; /* Restore previous Excpt Hndlr */
        add esp, 8 ; /* Remove ptr to HW handler and coroutine. */
        pop ebp ; /* Restore context */
        add esp, 4 ; /* Remove ptr to RAISE handler */
        mov al, 1

; /*#asm*/

;exception( 1 ) stmt ; /*#endasm*/

        jmp L1_endtry_hla_
L2_exception_hla_:
        cmp eax, 1
        jne L3_exception_hla_
        mov al, 2

; /*#asm*/

;exception( 2 ) stmt ; /*#endasm*/

        jmp L1_endtry_hla_
L3_exception_hla_:
        cmp eax, 2
        jne L4_exception_hla_
        mov al, 3

; /*#asm*/

;anyexception stmt ; /*#endasm*/

        jmp L1_endtry_hla_
L4_exception_hla_:
        mov al, 4

; /*#asm*/

;endtry stmt ; /*#endasm*/

L1_endtry_hla_:
        mov al, 5
QuitMain_hla__:
        pushd    0
        call    dword ptr __imp__ExitProcess@4
_HLAMain      endp

L5_exception_hla_ equ Raise_hla_
        end

```

Program 2.13 MASM Output File From the Exceptions Source

The purpose of this paper is not to explain how structured exception handling under Windows works (upon which HLA's exception handlers are based). Therefore, I'm not going to bother explaining what any of the statements mean in the code above. Instead, the important thing is to note the amount of code that each statement or clause produces.

The `raise` statement is fairly simple. It loads the value of its argument into EAX and then transfers control to the `Raise__hla_` standard library procedure (see the standard library sources if you're interested, it is a fairly short routine, though). As you can see, the `raise` statement doesn't generate a whole lot of code.

The `try..endtry` statement is at the other extreme. This statement probably generates more code than any other single high level control statement that HLA provides⁴. To get an idea of the amount of code generated for each clause, note that I've used the `#asm..#endasm` directive to inject comments into the MASM output file and I've used instructions of the form `"mov(const, al);"` to help delineate the code that HLA produces for each of the `try..endtry` clauses.

The `try..endtry` statement is very powerful and provides a sophisticated solution to the problem of exception handling. However, as you can see, the `try..endtry` sequence generates quite a bit of code (not a tremendous amount, but it add up if you place a lot of `try..endtry` statements in your program). If you're trying to write code that is as fast and as short as possible, you may produce better quality code by simply returning an error status from your procedures and functions rather than raising exceptions in those functions and relying on a `try..endtry` block to catch the exception. There is no guarantee that the explicit return value approach is faster or shorter, but it usually is shorter and faster (though it's nowhere near as convenient as `raise/try..endtry` and far more error prone). Just something to keep in mind.

1.2.4 Overhead Associated with Procedures, Iterators, and Methods

HLA was designed as a tool to teach assembly language programming to absolute beginners. Therefore, it does a couple of things by default that make it easier on beginners but may produce some excess code that an advanced assembly programmer would never write. One place where this is especially true is in the declaration and invocation of HLA procedures. Fortunately, HLA provides lots of options that let you control the extra code it emits for beginners (including turning off the code generation). This section explores the options you can use to control code generation for procedures and calls to procedures⁵.

By default, HLA automatically generates code at the beginning of a procedure to construct the activation record for that procedure, align the stack to a dword boundary, allocate local variables, and build a display for that procedure⁶. HLA also automatically generates the code to clean up the activation record and return from the procedure (and for the `stdcall` and `pascal` calling sequences, this code also cleans up the parameters on the stack). Sometimes this code is unnecessary (e.g., the procedure doesn't have any stack-based parameters or local variables), slightly less than efficient (e.g., you can access all the parameters and locals off ESP and you don't need to set up a stack frame with EBP), or you want to do things a little differently for some specific reason. Obviously in these situations, HLA's default behavior is not what you want. Fortunately, HLA makes it easy to modify its behavior for a specific procedure or even change the overall default behavior.

To begin with, it's probably a good idea to take a look at the code HLA automatically generates for a procedure. We'll use the following example over and over again with slight modifications in this section:

4. Technically speaking, this is not true. Using the conjunction(&&) and disjunction(||) operators, you can generate some really large if, while, etc., statements. However, anyone who creates a really huge boolean expression is going to expect a bit of code bloat).

5. This section will use the generic term "procedures" to mean any HLA procedure, iterator, or method, unless otherwise noted.

6. Displays are advanced data structures that provide access to non-local automatic variables.

```

program ProcDemo;

    procedure demo( b:byte; w:word; d:dword; var refvar:dword );
    var
        localVar:    dword;

    begin demo;

        nop();

    end demo;

begin ProcDemo;
end ProcDemo;

```

Program 2.14 The Generic HLA Procedure

Here's the assembly output for the demo procedure above (for the sake of brevity, I'll not put the whole MASM output file here - it's roughly the same code you'll find in the empty examples):

```

L1_demo_hla_ proc    near32
    push    ebp      /*Dynamic link*/
    push    dword ptr [ebp-4] /*Display for lex level 0*/
    lea    ebp,[esp+4] /*Get frame ptr*/
    push    ebp      /*Ptr to this proc's A.R.*/
    sub    esp,    4
    and    esp,    0fffffffch
    nop
xL1_demo_hla_hla_:
    mov    esp,    ebp
    pop    ebp
    ret    16
L1_demo_hla_ endp

```

Program 2.15 HLA Code Generation for the 'demo' Procedure

Notice that the original procedure only had one instruction (a NOP). HLA actually generates nine additional instructions inside this procedure. While some of them (e.g., the RET instruction) would have to be present, there is some fat here that can be trimmed, depending on your circumstances.

The first thing that you can almost always trim away is the generation of the code that builds the display. This is the second through fourth instructions above (push, lea, push). Displays are a special data structure that provide access to non-local automatic variables in nested procedures. 99% of the time (or better), most assembly procedures won't need a display. That's because 98% of all assembly language programmers will never nest their procedures and the 2% that do can often pull other tricks to access non-local variables without using a display. Therefore, the vast majority of the time you can eliminate these statements that set up the display from the procedure code. This is easily accomplished by supplying the @nodisplay procedure option, e.g.,

```

program ProcDemo;

```

```

procedure demo( b:byte; w:word; d:dword; var refvar:dword ); @nodisplay;
var
    localVar:    dword;

begin demo;

    nop();

end demo;

begin ProcDemo;
end ProcDemo;

```

Program 2.16 HLA Demo Program with @nodisplay Option

Here's the corresponding code that HLA emits for the program above:

```

L1_demo_hla_ proc    near32
    push    ebp
    mov     ebp, esp
    sub     esp,    4
    and     esp, 0fffffffch
    nop
xL1_demo_hla_hla_:
    mov     esp, ebp
    pop    ebp
    ret    16
L1_demo_hla_ endp

```

Program 2.17 HLA Code Generation for Demo With @nodisplay Option

Well, this code looks a whole lot closer to a procedure with a standard entry/exit sequence. About the only surprising piece of code here is the `and` instruction. HLA automatically emits this code to guarantee that the stack is aligned upon a four-byte boundary upon entering the procedure. If the caller has misaligned the value in `ESP` such that it is not an even multiple of four, certain system calls may fail. The `and` instruction above ensures that `ESP` is dword aligned. Unless you mess with `ESP`'s value (or push word values on the stack), `ESP` is always dword aligned. Note that this is true even if you specify some number of local variables whose aggregate size is not an even multiple of four (the `sub` instruction above reduces `ESP` by the number of bytes of local variables present, but HLA always rounds this value up to the next even multiple of four to keep `ESP` dword-aligned). If you know that `ESP` is dword-aligned (because you've not messed with the stack pointer), then the `and` instruction in the code above is superfluous. You may eliminate this extra instruction by specifying the `@noalignstack` procedure option:

```

procedure demo( b:byte; w:word; d:dword; var refvar:dword );
    @nodisplay;
    @noalignstack;

var
    localVar:    dword;

```

```

begin demo;

    nop();

end demo;

begin ProcDemo;
end ProcDemo;

```

Program 2.18 HLA Demo Code With @noalignstack Option

Here's the corresponding code that HLA emits for the program above:

```

L1_demo_hla_ proc near32
    push ebp
    mov ebp, esp
    sub esp, 4
    nop
xL1_demo_hla_hla_:
    mov esp, ebp
    pop ebp
    ret 16
L1_demo_hla_ endp

```

Program 2.19 HLA Code Generation for Demo With @nodisplay Option

Now we've gotten down to the point where the code looks just like the standard entry/exit sequence you'd expect for a procedure. Of course, there are some changes we could make still. For example, the 80x86 CPU family supports two instructions, `enter` and `leave`, that you may use to build and destroy activation records (including displays, if necessary). While these instructions are typically slower than the discrete instructions that do the same job, they are certainly shorter and, therefore, some programmers prefer to use them. By default, HLA generates discrete instructions to build and destroy activation records. However, by using the `@enter` and `@leave` procedure options, you can tell HLA to use these instructions rather than the discrete instruction sequences:

```

procedure demo( b:byte; w:word; d:dword; var refvar:dword );
    @nodisplay;
    @noalignstack;
    @enter;
    @leave;

    var
        localVar: dword;

begin demo;

    nop();

end demo;

```

```
begin ProcDemo;
end ProcDemo;
```

Program 2.20 HLA Demo Code With @noalignstack Option

Here's the corresponding code that HLA emits for the program above:

```
L1_demo_hla_ proc near32
    enter 4,0
    nop
xL1_demo_hla_hla_:
    leave
    ret 16
L1_demo_hla_ endp
```

Program 2.21 HLA Code Generation for Demo With @nodisplay Option

As you can see, this procedure is starting to get seriously shortened. HLA is emitting only three extra instructions (down from the original nine or so).

Of course, 'real' assembly language programmers want to write all their own code. If HLA is automatically generating anything for them, no matter how convenient, they're going to complain. Well, HLA provides the @noframe procedure option that eliminates all code generation other than the explicit machine instructions the programmer provides. Note that supplying @noframe implicitly supplies @noalignstack and, to a certain extent, @nodisplay since @noframe turns off all extra code generation in a procedure⁷. Here's the examples above specifying @noframe:

```
procedure demo( b:byte; w:word; d:dword; var refvar:dword );
    @nodisplay; // Still should be here, see footnote
    @noframe;

    var
        localVar: dword;

    begin demo;

        nop();

    end demo;

begin ProcDemo;
end ProcDemo;
```

7. Note, however, that if @noframe is not present, HLA will still assume you want to allocate storage for a display and will consider this fact when assigning offsets to local variables found in the procedure. Therefore, it's a good idea to go ahead and specify @nodisplay along with @noframe.

Program 2.22 HLA Demo Code With @noframe Option

Here's the corresponding code that HLA emits for the program above:

```
L1_demo__hla_  proc   near32
                nop
xL1_demo__hla__hla_:
                L1_demo__hla_  endp
```

Program 2.23 HLA Code Generation for Demo With @nodisplay Option

Now, however, we've got a problem. There is no RET instruction to return from this procedure. But that's okay, the "macho" assembly programmer who doesn't want HLA generating any code for them surely wants the program to fall through this procedure to the next instruction in memory, or they wouldn't have left out the RET instruction in the original code. Here's what the procedure would normally look like when the @noframe option is present:

```
procedure demo( b:byte; w:word; d:dword; var refvar:dword );
  @nodisplay; // Still should be here, see footnote
  @noframe;

  var
    localVar:  dword;

  begin demo;

    nop();
    ret( 16 );

  end demo;

begin ProcDemo;
end ProcDemo;
```

Program 2.24 HLA Demo Code With @noframe Option (Part II)

For those who want to write all their own code and not have HLA generate anything extra code in their procedures, constantly attaching @noframe and @nodisplay to every procedure declaration can get old, fast. Fortunately, HLA provides a mechanism that lets you set the default state for all of these procedure options.

As shipped, HLA defaults to the following options: @frame, @display, @alignstack, @noenter, @noleave. You can change the defaults by using these options as compile-time variables and setting them to true or false. Here are the possible options:

Table 1: Procedure Options and Their Effect on Code Generation

Option	Effect if set to true	Effect if set to false
@enter	HLA generates ENTER instruction to build activation records upon procedure entry. Note that @frame must also be true for HLA to emit this code.	HLA generates discrete instructions to build activation records upon procedure entry. Note that @frame must also be true for HLA to emit this code.
@noenter	Same as setting @enter to false.	Same as setting @enter to true.
@leave	HLA emits the LEAVE instruction to clean up the activation record upon exit. Note that @frame must also be true for HLA to emit this code.	HLA emits discrete instructions (mov, pop) to clean up the activation record upon exit. Note that @frame must also be true for HLA to emit this code.
@noleave	Same as setting @leave to false.	Same as setting @leave to true.
@display	HLA emits instructions that allocate storage for and initialize a display structure. If @enter is true, HLA emits an ENTER instruction to accomplish this, otherwise it emits discrete instructions. Note that @frame must also be true for HLA to emit this code.	HLA does not emit any instructions that allocate or initialize the display structure.
@nodisplay	Same as setting @display false.	Same as setting @display true
@alignstack	HLA emits an AND instruction that guarantees ESP is dword-aligned after allocating local variables. Note that @frame must also be true for HLA to emit this code.	HLA does not emit the AND instruction that dword-aligns ESP.
@noalignstack	Same as setting @alignstack to false.	Same as setting @alignstack to true.
@frame	HLA generates code to construct the stack frame and other duties (e.g., align the stack if @alignstack is true, build the display if @display is true).	HLA does not generate any extra code for the procedure. It is the programmer's responsibility to write any necessary code to build the stack frame, if required.
@noframe	Same as setting @frame to false	Same as setting @frame to true.

The "macho" assembly language programmer will probably include the following two statements at the beginning of every HLA program they write:

```
?@noframe := true;
?@nodisplay := true;
```

The inclusion of these two statements tells HLA that the programmer is responsible for writing all the code that appears within the source file. Note that you may re-enable display and frame generation on a procedure by procedure basis by using the @frame and @display procedure options. See the discussion of procedure options in the HLA reference manual for more details.

Note that HLA still makes parameter names and local variable names available to your procedures when you specify the @noframe option. However, the offsets associated with these variables assume that you've built a standard stack frame and that you're going to reference the objects off EBP. If this is not the case, then you should not use the parameter and local variable names in your code; you'll have to use numeric

offsets (say, from ESP) or, better yet, create TEXT constants that provide the necessary offsets from ESP, e.g.,

```
program ProcDemo;

?@noframe := true;
?@nodisplay := true;

procedure demo( _d:dword; var _refvar:dword );
const
    d      :text := "(type dword [esp+12])";
    refvar :text := "(type dword [esp+8])";
    localvar:text := "(type dword [esp])";
begin demo;

    pushd( 0 );      // Allocate _localVar and initialize to zero.
    mov( d, eax );
    mov( eax, localvar );
    mov( refvar, ebx );
    mov( eax, [eax] );
    add( 4, esp ); // Remove localvar from stack.
    ret( 8 );      // Return and pop parameters

end demo;

begin ProcDemo;
end ProcDemo;
```

Program 2.25 Using TEXT Constants to Access Parameters and Local Variables

```
L1_demo_hla_ proc    near32
    pushd    0
    mov eax, dword ptr [esp+12] ;/* (type dword [esp+12]) */
    mov dword ptr [esp+0], eax ;/* (type dword [esp]) */
    mov ebx, dword ptr [esp+8] ;/* (type dword [esp+8]) */
    mov [eax+0], eax ;/* [eax] */
    add esp, 4
    ret 8
xL1_demo_hla__hla_:
L1_demo_hla_ endp
```

Program 2.26 Code Generation for the Above HLA Procedure

1.2.5 Overhead Associated with Procedure Calls

As long as you manually pass the parameters yourself and use the CALL instruction, HLA does not inject any extra instructions into your code. However, if you use HLA's high-level procedure call syntax, HLA may very well emit some extra instructions into the code stream. If this bothers you, well, don't use the high level calling syntax – stick with the manual ("pure assembly") calling syntax.

However, the high level calling syntax is very convenient, it is far more readable and maintainable, and most of the time it generates exactly the same code you're going to write by hand. Therefore, it makes sense to use it as often as you can and understand the degenerate cases (where HLA emits some bad code) so you can code those by hand when efficiency is a prime concern.

First of all, HLA does a great job with "pass by value" parameters when the size of the value is four, eight, or 16 bytes. Such parameters generally require only a single instruction per double-word to push on the stack prior to the call⁸. As the objects get larger, passing them by value gets very expensive. At some point, HLA doesn't bother trying to push the data on the stack, instead, it uses a MOVSD instruction to copy the data onto the stack. The following code shows what happens when you try to pass a 256-byte variable by value:

```

program ProcDemo;

type
    b256:byte[256];

    procedure demo( b:b256 );
    begin demo;
    end demo;

static
    c:b256;

begin ProcDemo;

    demo( c );

end ProcDemo;

```

Program 2.27 Code That Passes a 256-byte Array by Value

```

        sub     esp, 256
        push   esi
        push   edi
        push   ecx
        pushfd
        cld
        lea   esi, [L2_c__hla_+0] ;/* c */
        mov   ecx, 64
        lea   edi, dword ptr [esp+16]
        rep   movsd
        popfd
        pop   ecx
        pop   edi
        pop   esi
        call  L1_demo__hla_ ;/* demo*/

```

Program 2.28 MASM Code HLA Emits for the Call to 'demo' Above

8. Assuming of course, you're passing the parameters on the stack and not in a register.

This isn't an example of HLA generating bloated code. HLA is doing a reasonable job given the request of the source code. However, HLA makes it so easy to write code that blows up like this that you can often make a mistake and pass a large data structure by value, causing HLA to generate a fair amount of slowing executing code (actually, once you get above 64 bytes, HLA usually generates a sequence like the one above (with possibly one or two additional instructions if the object's size is not an even multiple of four bytes). So the size won't change too much as the object gets larger, but the execution time required by the "rep movsd" instruction goes up linearly with the size of the object. Moral of the story: unless there are good semantic reasons for doing so, always pass large objects by reference rather than by value. Watch out for this, because HLA will gladly emit the code to pass it by value without complaining.

Note that for parameters up to 64 bytes in size, HLA will actually emit a series of discrete push instructions. For parameters that are 16 bytes or less, this is no big deal (it only takes four push instructions to pass a 16-bit parameter by value). However, it's going to take 16 push instructions to pass a single 64-bit parameter by value to a procedure. This can cause some serious code bloat if you're doing this a lot. Moral: same as before, pass large objects by reference rather than by value (large is probably anything greater than 16 bytes in size).

HLA can go through some real gymnastics attempting to pass small parameters by value, as well. Because most modern (32-bit) operating systems always expect the stack to be dword aligned, HLA (like most languages and OS API functions) always passes a parameter using a multiple of four bytes to hold that value. So if you're passing an object that's one, two, or three bytes in size, HLA will pass four bytes as the actual parameter. The procedure (generally, this is actually up to the programmer) ignores the extra bytes. This creates a problem when attempting to pass certain parameters on the stack; HLA solves these problems at the expense of greater code. Consider the following HLA program that has a one byte parameter and calls the procedure several different ways:

```
program smallParmDemo;

procedure byteParm( b:byte );
begin byteParm;
end byteParm;

static
    b:byte;

begin smallParmDemo;

    byteParm( b );
    byteParm( al );
    byteParm( ah );
    byteParm( (type byte [eax]) );

end smallParmDemo;
```

Program 2.29 Procedure with a One-Byte Parameter

Here's the MASM code HLA generates for each of the calls to byteParm:

```
    pushd    00h
    push    eax
    mov al, [L2_b_hla_+0] ;/* b */
    mov byte ptr [esp+4], al
    pop    eax
    call    L1_byteParm_hla_ ;/* byteParm*/
```

```

push    eax
call    L1_byteParm_hla_    ;/* byteParm*/

sub esp, 4
mov byte ptr [esp], ah
call    L1_byteParm_hla_    ;/* byteParm*/

pushd   00h
push    eax
mov al, [eax+0] ;/* (type byte [eax]) */
mov byte ptr [esp+4], al
pop    eax
call    L1_byteParm_hla_    ;/* byteParm*/

```

Program 2.30 MASM Code HLA Generates for the Calls to byteParm

Many of these calls have an incredible amount of bloat! Any mediocre assembly programmer can probably do a better job than this! Why is HLA so bad? The reason HLA generates some ugly code here is because HLA makes a promise that it won't change any register values when passing parameters to a procedure (just in case you're passing some additional parameters in some registers). This promise severely impacts HLA's options when it comes to copying parameter data to the stack⁹. Indeed, about the only option HLA has when it needs a register is to preserve that register's contents while copying the parameter data. Consider the first call to `byteParm` above (passing the byte variable `b`). HLA first makes room for `b` on the stack by pushing a doubleword zero value. The HLA emits code to push the value of `EAX`, copy `b`'s value into `AL`, store `AL` into the stack location allocated earlier, and then restore `EAX`'s original value.

Now the clever assembly programmer might claim that this could be done far more efficiently with a single instruction, as follows:

```
push( (type dword b) );
```

99.999% of the time, that programmer would be right; this is a much better way to pass a single byte parameter in a dword slot on the stack (this instruction pushes the value of the three bytes the follow `b` in memory, but since the procedure will ignore those three bytes anyway, who cares?). Unfortunately, this trick fails spectacularly in one very special (and, admittedly, rare) case. Consider what happens when `b` is allocated as the 4096th byte in a page and the next page in memory is not read-enabled. This is cause the program to crash. Granted, it's incredibly unlikely that this will ever happen in an HLA program. However, HLA's design can't make the assumption that it won't ever happen. So HLA has to generate safe, but ugly, code.

Of course, there's nothing preventing you from recognizing this problem and manually pushing `b`'s value as a dword yourself. E.g., either of the following will work:

```
push( (type dword b) );
call byteParm;
```

-or-

```
byteParm( #{ push( (type dword b) ); }# );
```

As long as you can ensure that there are three reasonable bytes following `b`, this scheme is quite a bit more efficient than the default code HLA generates.

The second and third calls to `byteParm` in the example above are the ones where HLA actually generates half-way decent code. If the byte parameter falls in the L.O. byte of a 32-bit register, HLA will simply push

9. It is interesting to note that MASM does not make this same promise. It will happily wipe out the `EAX` register if it needs a scratch-pad register while passing parameter data to a procedure via the `INVOKE` statement. I like to believe that HLA is a bit more "civilized" in this regard.

the contents of that 32-bit register onto the stack. You aren't going to do any better than this (short of passing the parameter in a register, rather than on the stack). The second call, passing the byte parameter in AH (or any other byte register that is not the L.O. byte of a 32-bit register) needs two instructions: one to allocate storage on the stack (PUSH) and another to copy the register's value onto the stack. An expert assembly language programmer, if they know they've got a register to play around with, can, perhaps, generate slightly better code by copying the eight-bit value to the L.O. byte of that register and then pushing the full register, e.g.,

```
mov bl, ah
push ebx
```

This sequence is slightly shorter, though probably not any faster, than the code that HLA generates.

The fourth example above is really just a special case of the first example. If you look at the two code sequences, you'll notice that they are equivalent.

HLA generates less than stellar code for some of these sequences because it assumes that all registers are in use and it shouldn't modify any register values. Obviously, this is not always the case when you're calling a procedure. However, it's a rather difficult problem for HLA to automatically determine if there is a free register available that it can use while passing parameters. Fortunately, HLA provides a way for you to tell it that it can freely use one register (which is all it needs) for processing parameters: the `@use reg` procedure option. Consider the following modification of the previous program:

```
program smallParmDemo;

procedure byteParm( b:byte ); @use ebx;
begin byteParm;
end byteParm;

static
  b:byte;

begin smallParmDemo;

  byteParm( b );
  byteParm( al );
  byteParm( ah );
  byteParm( (type byte [eax]) );

end smallParmDemo;
```

Program 2.31 byteParm and @use ebx

The `@use ebx` option tells HLA that it can freely use the EBX, BX, BL, and BH registers when generating the code to pass parameters to this procedure. Here's the code HLA generates when you allow it to use the EBX register in this capacity:

```
mov bl, byte ptr [L2_b_hla_+0] ;/* b */
push ebx
call L1_byteParm_hla_ ;/* byteParm*/
push eax
call L1_byteParm_hla_ ;/* byteParm*/
sub esp, 4
mov byte ptr [esp], ah
call L1_byteParm_hla_ ;/* byteParm*/
mov bl, byte ptr [eax+0] ;/* (type byte [eax]) */
```

```

push    ebx
call    L1_byteParm_hla_    ;/* byteParm*/

```

Program 2.32 HLA Generated Code for the Above Calls to byteParm

As you can see, the code is much better than before (not quite as good since it still doesn't assume it can push `b` directly onto the stack, but much better nonetheless). Of course, if you want to take absolute control, you can always push the parameter manually.

Of course, the stack isn't the most efficient place to pass parameters. The x86 registers are the best place to pass parameters (subject to the constraint that they fit in the registers). Note that HLA will allow you to pass parameters in register using a high level calling syntax as follows:

```

procedure parmsInRegs( a: dword in eax; var b: byte in ebx );
.
.
.

```

Of course, there's nothing stopping you in HLA from simply loading a register with some value prior to a call and referencing that register inside the procedure without declaring any formal parameters. The nice thing about using the high level declaration and calling syntax is that HLA will automatically move the value into the register for you if you specify an actual parameter other than the register for that parameter. However, since there's not much in the way of bloat here, there's really no sense in discussing it farther in this document. See the HLA reference manual for more details.

Reference parameters have their own special problems. As long as you're passing a non-indexed static address (that is, the address of a static, readonly, or storage) object by reference, HLA generates really good code (a single push instruction). However, once you throw in an index register or specify an automatic variable (whose offsets are indexed off `EBP`), HLA has to emit an `LEA` instruction to compute the effective address of the operand. Since the `LEA` instruction requires a register, we're back to the same problem we had with the byte-sized operand earlier. Well, the solution is the same: if you want decent code, either pass the address manually or specify an `@use` procedure option to tell HLA that it can use a register for computing effective addresses.

HLA supports several other parameter passing mechanisms. This document won't cover them for two reasons: (1) 99% of the assembly language programmers out there have probably never heard of these parameter passing mechanisms, and (2) the 1% of them who have, know that they're usually inefficient anyway (and fast/short code avoids them like the plague).

1.2.6 Bloat in the HLA Standard Library

It goes without saying that if you want to understand the purpose of every byte in your HLA programs, you don't call HLA Standard Library routines. It's not that they're incredibly poorly written, but they're "black boxes" and unless you sit down and study their source code, you have no idea what (private) data they declare, what routines they call, or anything else about their efficiency.

The HLA Standard Library routines were not written to be the fastest nor the shortest examples of HLA code. They were written to be easy to read, understand, and maintain. Furthermore, many of the routines build upon other routines. A classic example is the `stdout.puti8` routine. This procedure takes a single byte parameter. It calls the `conv.i8ToStr` procedure to convert the value to a string, then calls the `fileio.puts` function to actually print the string (specifying the standard output file handle as the "file"). The `conv.i8ToStr` function zero extends the eight-bit value to 16 bits and calls the `conv.i16ToStr` function. The `conv.i16ToStr` function zero extends its 16-bit value to 32 bits and calls the `conv.i32ToStr` function. The `conv.i32ToStr` function zero extends its value to 64 bits and calls the `conv.i64ToStr` function. The `conv.i64ToStr` function actually converts its 64-bit value (include 56 bits of zeros at this point) to a string and the chain of calls pass the string back to the original call from `stdout.puti8`. Each of these routines (except `conv.i64ToStr`, which

does all the real work) is very short and fairly trivial. If your program winds up calling all of these routines, this is probably the most compact representation you could come up with. However, this obviously requires a lot more code than had the standard library simply provided a `conv.i8ToStr` function that did the conversion directly. Furthermore, all those extra calls, plus the fact that converting a 64-bit value to a string is more expensive than converting an eight-bit value to a string, means the code is going to run a bit slower. Therefore, if speed and/or space are prime considerations in your program, avoid the HLA Standard Library (or, always start with the source code to the routine you want to call and clean it up so avoid long call chains like the one in the above example).

There is another source of bloat that is indirectly related to the HLA Standard Library. The HLA Standard Library was modelled after the standard libraries found in C, C++, and other high level languages. As a result, calling these library routines causes you to "think" like a C programmer. And as any expert assembly programmer can tell you, "thinking in assembly" is the only way to write efficient assembly programs. Even if all the routines in the HLA Standard Library were written as efficiently as possible, the mindset they leave you in is not conducive to writing efficient code. Therefore, take care when using the HLA Standard Library because it can cause you to write sloppy code if you're not carefully considering what you're doing at each step in your code.

1.3 Taking Control with HLA Units

Reading the HLA reference manual, you might get the impression that HLA applications are written as PROGRAMs and separately compiled modules that you link with HLA or applications in other languages are written using UNITs. HLA units are actually a bit more flexible than this, if you're willing to play some games. In particular, HLA units can completely free you from the yoke of HLA compiler-generated code and give you an environment where the only instructions that appear in your executable file are those instructions you write. This section will describe how to use HLA units to achieve this.

Fundamentally, there are only a couple of differences between HLA units and HLA programs. HLA programs allow you to declare automatic variables in a global VAR section, units do not¹⁰. The major difference, of course, is that HLA units don't have a "main program" associated with them as HLA programs do. If you take a look at the code that HLA generates for units and programs, you see only a couple of differences between the output files. Specifically, HLA collects all the code from the main program and creates a MASM (Gas) procedure named `_HLAMain` (`_start`). Also, HLA emits some support code to initialize the exception handling system for programs, none of this code appears in the assembly output file for a unit. Other than these two issues, HLA units and programs are semantically equivalent.

To prove this point, the following is an HLA unit that compiles to the same exact code as the standard Hello World program.

```
unit unitAsPgm;
#include( "stdout.hhf" )

?@nodisplay := true;
?@noframe := true;

// Make these names public so the library routines
// and linker can find them.

procedure _HLAMain; @external;
procedure Hwexcept__hla_; @external;
procedure DfltExHndlr__hla_; @external;
```

10. Which makes sense because VAR objects are always associated with a procedure or the HLA main program. In a unit, there is no main program with which you can associate automatic variables.

```

// The following are HLA Standard Library procedures.
// Just make 'em labels rather than procs because we
// just JMP to these labels.

label
    shorthwExcept__hla_ ; @external;
    shortDfltExcept__hla_ ; @external;
    BuildExcepts__hla_ ; @external;
    QuitMain; @external( "QuitMain__hla_" );

static

    // The following is the link to the Win32 API ExitProcess procedure
    // address.

    __imp__ExitProcess :dword; @external( "__imp__ExitProcess@4" );

    // The main program needs a coroutine object for
    // use by the exception handling subsystem:

    MainPgmCoroutine__hla_ : dword; @external;
    MainPgmCoroutine__hla_ : dword; @nostorage;
        dword &MainPgmVMT__hla_ ;
        dword 0,0,0,0;

    MainPgmVMT__hla_ : dword := &QuitMain;

// The following are needed to provide linkage to
// the HLA exception handling routines.

procedure HWexcept__hla_ ;
begin HWexcept__hla_ ;
    jmp shorthwExcept__hla_ ;
end HWexcept__hla_ ;

procedure DfltExHndlr__hla_ ;
begin DfltExHndlr__hla_ ;
    jmp shortDfltExcept__hla_ ;
end DfltExHndlr__hla_ ;

procedure _HLAMain;
begin _HLAMain;

    call    BuildExcepts__hla_ ;
    pushd( 0 );           // no dynamic link (previous proc's EBP).
    mov( esp, ebp );     // Set up our stack frame.
    push( ebp );         // Main's display.

    // << put main program code here >>

```

```

        stdout.put( "Hello World" nl );

end _HLAMain;

// Fall through from the above and return to Windows.
// (this needs to be outside _HLAMain because QuitMain_hla_
// needs to be a public name).

procedure QuitMain;
begin QuitMain;

    pushd(0);
    call( __imp__ExitProcess );

end QuitMain;

end unitAsPgm;

```

Program 2.33 Hello World Program Written as a Unit

The HLA compiler instructs the linker to start program execution at the label `_HLAMain`. By writing a procedure named `_HLAMain` and making this name public (via the `@external` directive), this unit provides an HLA "main program" that the OS will invoke immediately after loading the program into memory. This main program explicitly contains the instructions that the HLA compiler would normally emit for a program (the call to `BuildExcepts__HLA_` and setting up the activation record). Following the initialization code is the invocation of the `stdout.put` macro that prints "Hello World" to the standard output. One unusual feature of this code is that the `QuitMain` label has to be global and public (i.e., we can't simply put the code that returns to Windows inside the `_HLAMain` procedure because external code references this label and you can't reference local labels from outside a procedure). The alternative would be to duplicate the code, but then we wouldn't have the semantic equivalent of the original Hello World program. If you compare the assembly output of this code with the assembly output of the standard Hello World program, you'll find that the code is nearly identical (about the only real difference is the extra procedure surrounding the code that returns to the OS; of course, this does not change the executable file one byte).

Of course, it doesn't make any sense to simply duplicate the effects of an HLA program within a unit (other than to prove it can be done). The real reason for using units in this fashion is to gain complete control over the code appearing in the executable file. Specifically, I'm assuming you want to dump some of the initialization code, data structures, and support code that exist primarily for the benefit of the HLA run-time system and exception handling subsystem. Here's the bottom line, if you want to take full responsibility for all the code appearing in your HLA program, write it as a unit and create an `_HLAMain` procedure to serve as your main program (note: Linux users need to name their main program `"_start"`). Here's the template you should use:

```

unit barebones;

?@nodisplay := true;
?@noframe := true;

procedure _HLAMain; @external;

procedure _HLAMain;
begin _HLAMain;

```

```

    // Put the code for your main program here.

end _HLAMain;

end barebones;

```

Program 2.34 Bare Bones HLA Program Implemented via a Unit

If you write your code using this "barebones" unit as a template, you're going to be in complete control of the code in your program. Do keep in mind that unless you initialize the exception handling system using the code given earlier (`BuildExcepts__HLA_`, etc.), you'll not be able to use HLA exceptions and that pretty much means you can't call any HLA Standard Library routines (since a large percentage of those can raise an exception). However, you will have completely escaped HLA's interference with your code and the only machine instructions that will find their way into your programs are the ones you write (or the code associated with any external routines you call).

I've made a big deal about using HLA units to give you complete control over the code HLA emits. Throughout this document, I've given the impression that only hard-core, die-hard, macho, assembly language programmers would want to do this. Actually, there are many real-world applications where the code that HLA emits for programs would be inappropriate. A classic example is the need to write dynamic link libraries. Such code has to be implemented as a unit, you cannot use an HLA procedure for such code.

1.4 Hello World, Revisited

This document began with a lament about the size of a typical Hello World program and mentioned that it's possible to write a shorter version of the program using HLA. In this section we'll explore how to write a short version of this program. Actually, let's forget exploring and jump right into things.

Based on what I've said about the HLA Standard Library, it should come as no surprise that the smallest Hello World program is not going to call any Standard Library routines. The most compact Hello World program is going to make direct OS API calls. Well, without further ado, here are the compact versions of the Hello World program for Windows and Linux (different versions are necessary since the OS APIs are different).

```

unit HelloWorld;

?@noframe := true;

procedure main; @external( "_HLAMain" );

static

    WriteFile:procedure
    (
        Handle:          dword;
        var buffer:      var;
        len:             dword;
        var bytesWritten: dword;
        overlapped:      dword
    );
    @use edx;
    @stdcall;
    @external( "__imp__WriteFile@20" );

```

```

    GetStdHandle:procedure
    (
        WhichHandle:int32
    );
    @stdcall;
    @external( "__imp__GetStdHandle@4" );

    ExitProcess:procedure( exitcode:dword );
    @stdcall;
    @external( "__imp__ExitProcess@4" );

procedure main;
var
    BytesWritten    :dword;
begin main;

    GetStdHandle( -11 );
    WriteFile( eax, &hwString, 13, BytesWritten, 0 );
    ExitProcess( 0 );

    hwString:    byte    "Hello World", $d, $a;

end main;

end HelloWorld;

```

Program 2.35 Windows Version of the Short Hello World Program

If you compile this program to an executable, guess what? It's going to need 12K for the .EXE file (4K for the PE/COFF header file, 4K for the Win32 API pointers, and 4K for the code and string data (which is merged into the code above, explaining why the program is 12K rather than 16K). Again, this is a PE/COFF and linker issue. We'll see in a moment how to rectify this problem. In the meantime, here's the Linux version of this program:

```

unit hw;

procedure main; @external( "_start" );

procedure main; @noframe;
begin main;

    // Print Hello World:

    mov( 4, eax );
    mov( 1, ebx );
    lea( ecx, helloWorld );
    mov( 12, edx );
    int( $80 );

    // return to Linux:

    mov( 1, eax );
    mov( 0, ebx );
    int( $80 );

```

```

    helloWorld: byte "Hello World", $a;

end main;

end hw;

```

Program 2.36 Linux Version of the Short Hello World Program

Compiling this program under Linux produces a much smaller executable file. In fact, it's under 1K. This is largely due to the more efficient ELF executable format.

Now there is a trick you can play with the Microsoft linker to reduce the size of the Hello World program to less than 1K under Windows. The Microsoft linker normally aligns all sections on 4K boundaries. There is an option you can specify to change this alignment ("`/align:value`"). When you use this linker option, the linker will complain about the fact that you should only use it for DLLs and VxDs. However, the Hello World program still seems to run fine when you use this option. However, I'd take that warning seriously and not use it for run-of-the-mill application development (this is one of the reasons, for example, that HLA doesn't specify this option). To test this option out, compile the Windows Hello World program above to an object file using the HLA "`-s`" command line option. Compilation of the source file under Windows produces a special linker response file (e.g., "`hw.link`"). Edit this file; it should look like the following (HLA v1.38):

```

/heap:0x1000000,0x1000000
/stack:0x1000000,0x1000000
/BASE:0x4000000
/machine:IX86
-entry:HLAMain
/section:.text,ER
/section:.data,RW
/section:.bss,RW
kernel32.lib
user32.lib
gdi32.lib

```

Just add the line "`/align:16`" somewhere in this file and run HLA with the following command line:

```
hla hw.asm
```

Note that the command line specifies the `hw.asm` file, not `hw.hla`. If you recompile `hw.hla`, it will overwrite the change you just made to the `hw.link` linker response file and you'll have to redo the change. After the compile operation, check out the size of the Hello World program. By the way, you should try this same trick with the typical HLA Hello World program (that calls the Standard Library) and compare the file sizes.

1.5 Okay, You're on Your Own!

This paper has discussed how to take complete control over the assembly code that HLA produces. Armed with this information, advanced assembly language programmers can no longer complain that HLA mucks around with their code and this is sufficient reason to avoid HLA. Have fun and good luck with this information.

