

Calling HLA Code From Other Languages

As explained in the HLA manual, calling HLA procedures and functions from other languages is generally easy. Just create an “external” procedure declaration (to make your procedure’s name public), compile the procedure as part of a unit, link it with your other code, and you’re in business (see the Chapter on “Mixed Language Programming” in the Art of Assembly Language) for more details). There is one catch, and I quote from the chapter on Mixed Language Programming from “The Art of Assembly Language”:

A large percentage of the HLA Standard Library routines include exception handling statements or call other routines that use exception handling statements. Unless you’ve set up the HLA exception handling subsystem properly, you should not call any HLA Standard Library Routines from non-HLA programs.

Similarly, you should not use any exception handling statements in code that you call from non-HLA code unless you’ve properly set up the exception handling subsystem.

Until now, that advice has simply meant “*Don’t use exceptions and don’t call any routines that use exceptions (e.g., HLA Standard Library routines) when calling HLA procedures from a non-HLA main program.*” The reason for this tough restriction? Simple, other than myself and perhaps a few hearty programmers who’ve probed the internals of HLA-generated code, very few people have known how to set up the HLA exception handling system properly.

Properly setting up the HLA exception handling system isn’t that complex. In fact, once you know what you’re doing, it’s actually quite easy. However, until now that knowledge hasn’t been publically available, so the best advice has always been “don’t even try it.” The purpose of this white paper is to rectify this situation by describing what you need to do to initialize HLA’s exception handling system.

Before going too much farther, I should point out that the information in this document is specific to Windows. While the same concepts apply to Linux, there are a few differences. If there is demand for such a thing, I’ll be more than happy to create a document such as this one for Linux users. The principle differences have to do with the way x86 CPU exceptions are handled. The general HLA exception handling mechanism is the same under both OSes, it’s just a question of how the HLA exception handling subsystem taps into the OS’ exception system.

When an HLA program first starts running, it executes a (compiler-generated) call to an HLA Standard Library procedure called *BuildExcept*s. BuildExcepts creates a Windows-compatible SEH (Structured Exception Handling) record in the main program’s stack frame. This SEH record becomes the “catch-all” for any exceptions that the program doesn’t specifically handle. Should an exception wind its way down to this particular exception handling record, then the code executes the programs default exception handler, that displays an error message and aborts the program.

The problem with calling HLA code from another language is that this default SEH record has never been built, because there is no HLA main program executing that built this record upon initial execution. When an unhandled exception comes along, the system generally crashes or hangs as there exists no default exception handler to deal with the exception. To avoid this problem (so you can use exceptions and call code that uses exceptions), what you've got to do is manually build that SEH record yourself. Actually, you don't have to build the SEH record yourself - that's exactly what the HLA Standard Library *BuildExcept*s procedure does. What you've got to do is call this procedure so it can build the SEH record for you.

In a normal HLA main program, an application calls *BuildExcept*s exactly once - immediately upon entry into the main program. This creates a single SEH exception handling record that sits around on the stack until the program exits. Unfortunately, when you call HLA code from some other language, you don't get the opportunity to build this SEH record at the beginning of the main program's execution (and even if you did, there is no guarantee that the exception handling system in place in that other language is compatible with HLA's). Therefore, we won't be able to build the SEH record once and forget about it; instead, we'll have to build the SEH record on each call to some HLA procedure from external code, and we'll have to tear down that SEH record before leaving. Yep, this is all overhead that you're going to execute on each call to an HLA function you make from some other language. The good news is that setting up (and tearing down) the SEH record takes less than a dozen instructions, so it's not that big of a deal.

Setting up and tearing down the SEH isn't the only work involved in supporting exceptions in HLA code. There are a couple of routines and a couple of data structures that the HLA compiler automatically generates whenever you write a main program. You'll have to manually supply these routines and data structures yourself.

The data structures exist to support HLA coroutines. Though it's unlikely you'll use coroutines in HLA code you call from C or some other language, you still have to create a coroutine data structure for the "main program" because the HLA exception handling code references this data structure. This is easily achieved with the following HLA code:

```
static
    MainPgmVMT: dword:= &QuitMain;

    // The following comprise the Main Program's coroutine data structure.

    MainPgmCoroutine: dword[ 5 ]; @external( "MainPgmCoroutine__hla_" );
    MainPgmCoroutine: dword; @nostorage;
                        dword &MainPgmVMT, 0, 0;
    SaveSEHPointer:   dword; @nostorage;
                        dword 0, 0;
```

The important field in this structure is the *SaveSEHPointer* field. The exception handling system expects a pointer to the previous SEH record in this field. The *BuildExcept*s stores the old SEH pointer in this field, when your code returns it should restore the SEH pointer from this field. You can ignore the remaining fields in these two data structures, they just exist to keep HLA happy.

The HLA Standard Library provides three routines we'll need to reference in the exception handler code we're setting up. However, the HLA Standard Library header files don't provide prototypes for all of these routines (because it would be unusual for user code to call them), therefore, you'll also have to manually supply prototypes for these routines. The prototypes are

```
procedure BuildExcept; @external("BuildExcept__hla");
procedure HardwareException; @external( "HardwareException__hla" );
procedure DefaultExceptionHandler; @external( "DefaultExceptionHandler__hla"
);
```

BuildExcept we've already discussed. The *HardwareException* procedure is where the system would normally transfer control on a hardware exception. The *DefaultExceptionHandler* is the code that HLA jumps to whenever an exception occurs. The purpose behind these last two procedures is to allow the HLA compiler to link in a separate set of exception handling routines depending on whether you want a "compact" exception handler or the full exception handler (the difference has to do with the size of the string data that HLA would link in). Throughout this paper we'll assume you want to link in the full exception handling package. See the details in the HLA reference manual concerning exceptions (and look at the code HLA emits for short exceptions) if you're interested in linking in the shorter version of the exception handler (with a single generic message rather than exception-specific messages).

In addition to the Standard Library routines given above, the HLA compiler also writes a couple of procedures (and provides program termination code). These procedures take the following form:

```
procedure QuitMain;
begin QuitMain;

    ExitProcess( 1 );

end QuitMain;

procedure HWexcept;
begin HWexcept;

    jmp    HardwareException;

end HWexcept;

procedure DfltExHndlr;
begin DfltExHndlr;

    jmp    DefaultExceptionHandler;

end DfltExHndlr;
```

QuitMain, in the HLA generated code, is really just a label, not a full procedure. HLA transfers control to this label whenever it wants to terminate the program. As some exceptions will transfer control to this label, you must supply this label in your code. All this procedure's body need do is

return control to the operating system. You can actually sneak in anything else you want, but when the procedure completes, it must return control to Windows (e.g., via the `ExitProcess` call).

The *HWexcept* label is where HLA's initialization code points the "hardware exception vector." Specifically, hardware exceptions like divide errors, segmentation faults, bounds violations, etc., first jump to this procedure. This short procedure simply passes control to the routine in the HLA Standard Library that actually handles the hardware exception.

DfltExHndlr is another procedure written by the HLA compiler. The purpose of this routine is to allow HLA code to link with the full exception handler (*DefaultExceptionHandler*) or the short exception handler (see the HLA standard library exception handling code for details). As noted earlier, in this paper we're going to use the full exception handling system.

To explain how to use all these functions and data types, an example is in order. Consider the following C program that will call an HLA procedure named *hlaFunc*:

```
/*
** A demonstration of how you can call HLA code
** that calls the HLA Standard Library from code
** that is not an HLA main program (in this case, it's
** a "C" program).
**
** Note: this program was compiled with Microsoft VC++
** using the following command lines:
**
** c:>vcvars32
** c:>hla -c hlafunc.hla
** c:>cl demo.c hlafunc.obj hlalib.lib kernel32.lib user32.lib
*/

#include <stdio.h>

extern void hlaFunc( int value );

int
main( void )
{
    printf( "Calling HLA code\n" );
    hlaFunc( 10 );
    printf( "Returned from HLA code\n" );

    return 0;
}
```

As usual, we'll place the code we want to call from our C function in an HLA unit and compile this to an `.OBJ` file. Here's the complete HLA procedure (discussion to follow):

```
unit hlaFuncUnit;
```

```

// We want to demonstrate how to call HLA Standard Library
// routines from code that is called from C, so let's include
// the standard library right here.

#include( "stdlib.hhf" )

// Here's the sample function we're going to call from external
// code ("C" in this example) that demonstrates HLA stdlib calls
// and exception handling.

procedure hlaFunc( i:int32 ); @cdecl; @external( "_hlaFunc" );

// These are declarations for procedures that exist in the HLA
// standard library, but are "shrouded" in the sense that there
// aren't corresponding declarations in the stdlib.hhf file (these
// routines generally get called by HLA generated code, and nothing
// else; however, as we have to simulate "HLA generated code" here,
// we have to manually provide these declarations):

procedure BuildExcepts; @external("BuildExcepts__hla_");
procedure HardwareException; @external( "HardwareException__hla_" );
procedure DefaultExceptionHandler; @external( "DefaultExceptionHandler__hla_"
);

// The following are forward/external declarations for procedures
// that are normally created by the HLA compiler when you write
// a "main program." As we are not using an HLA main program here,
// we have to manually create these procedures.

procedure HWexcept; @external( "HWexcept__hla_" );
procedure DfltExHndlr; @external( "DfltExHndlr__hla_" );
procedure QuitMain; @external( "QuitMain__hla_" );

// The following is a Win32 API function this code calls:

procedure ExitProcess( rtnCode:dword ); @external( "_ExitProcess@4" );

// The following are some global, public, variables that the
// HLA exception handling run-time system expect the compiler
// to create for the HLA main program. Once again, as we are not
// writing an HLA main program here, we have to manually supply
// these objects:

static
    MainPgmVMT:dword:= &QuitMain;

    MainPgmCoroutine: dword[ 5 ]; @external( "MainPgmCoroutine__hla_" );
    MainPgmCoroutine: dword; @nostorage;
                        dword &MainPgmVMT, 0, 0;
    SaveSEHPointer:   dword; @nostorage;
                        dword 0, 0;

// HLA main programs provide a "QuitMain" external label that

```

```
// exception handling code can when the exception causes the
// program to abort. This label immediately terminates program
// execution. As we are not writing an HLA main program, the HLA
// compiler does not provide this code for us, we have to supply
// it manually. You can do anything you want here, as long as you
// cause the *whole* program to terminate execution. This particular
// example simply calls ExitProcess and returns a termination code
// of one (which you can change to anything you want; non-zero usually
// indicates successful completion of the application, but this label
// normally gets called when the application aborts because of some
// exception, so returning zero isn't typical in this particular case.
```

```
procedure QuitMain;
```

```
begin QuitMain;
```

```
    ExitProcess( 1 );
```

```
end QuitMain;
```

```
// HWexcept is where the OS would normally transfer control
// when an x86 exception occurs. This procedure is normally
// written by the HLA compiler and simply jumps to an
// appropriate handler in the HLA Standard Library.
```

```
procedure HWexcept;
```

```
begin HWexcept;
```

```
    jmp    HardwareException;
```

```
end HWexcept;
```

```
// DfltExHndlr is where the exception handling code transfers
// control when an HLA exception occurs. This is normally
// written by the compiler (to allow the compiler to choose
// between the full and short forms of the default exception
// handler). NOTE: the following code invokes the *full*
// exception handler (lots of meaningful messages, at the
// expense of the space needed for all those messages).
```

```
procedure DfltExHndlr;
```

```
begin DfltExHndlr;
```

```
    jmp    DefaultExceptionHandler;
```

```
end DfltExHndlr;
```

```
// Here's the HLA code we're going to call from C that
// demonstrates exception handling without an HLA main program.
```

```
procedure hlaFunc( i:int32 );
```

```
var
```

```
    s:string;
```

```

saveSEH:dword;

begin hlaFunc;

    // Before doing anything else, save a copy of the SEH pointer:
    #asm
        mov eax, fs:[0]
    #endasm
    mov( eax, saveSEH );

    // Upon entry into any HLA code that needs exception support,
    // we have to set up the structured exception handling record
    // for HLA:

    call BuildExcepts;

    // Because exception handling code can mess up all the registers,
    // we need to preserve EBX, ESI, and EDI across this call:

    push( esi );
    push( edi );
    push( ebx );

    // Okay, here's the code we're going to execute that uses
    // exceptions, calls HLA stdlib routines, etc., even though
    // caller is not an HLA program:

    try

        stdout.put( "stdout.put called from HLA code, i = ", i, nl );
        raise( 5 );

        exception( 5 );
        stdout.put( "Exception handled by HLA code" nl );

    endtry;

    // One more demonstration, this time with an exception
    // occurring deep down inside an HLA Standard Library routine:

    try

        stralloc( 16 );
        mov( eax, s );
        str.cpy( "Hello World", s );
        stdout.put( "Successfully copied 'Hello World' to s: ", s, nl );
        str.cpy( "0123456789abcdefghijklmnop", s );
        stdout.put( "Shouldn't get here" nl );

    anyexception

        stdout.put( "Exception code: ", eax, nl );
        ex.printStackTrace();

```

```

endtry;
strfree( s );
stdout.put( "Returning to C code" nl );

// Restore the registers we saved earlier:

pop( ebx );
pop( edi );
pop( esi );

// Restore the saved SEH value:

mov( saveSEH, eax );
#asm
    mov fs:[0], eax
#endasm

end hlaFunc;

end hlaFuncUnit;

```

The *hlaFunc* procedure appearing at the end of this source file is of primary interest to us here. The HLA function you call from C (or any other language) must begin by immediately saving the SEH pointer and then calling *BuildExcept*s upon entry into the procedure. This constructs the HLA SEH record and initializes the HLA exception handling system. Just as important, before the procedure returns it must clean up the SEH record; this is accomplished with the last two MOV instructions in this code (including the one appearing in the #asm..#endasm sequence). Everything between those two points is the normal body of your procedure. This code can use the try..endtry statement, raise exceptions, and call external procedures that using try..end and/or raise exceptions. The code appearing in this sample both demonstrates directly raising an exception and calling an HLA Standard Library routine that raises an exception. Also note how this code is free to call HLA Standard Library routines without fear of crashing the system should an exception occur.

It is important to realize that you must call *BuildExcept*s and clean up the SEH record in each HLA procedure you call from some other language. Note, however, that you don't have to do this for HLA procedures that you only call from HLA code (that has already built the SEH record).

This paper should provide you with sufficient information to initialize the HLA exception handling system whenever you call an HLA procedure from some other language (or whenever the HLA exception handling system has not been previously initialized). If you have any questions, feel free to email me at rhyde@cs.ucr.edu.

Randy Hyde