Debugging HLA Programs Appendix J

J.1 The @TRACE Pseudo-Variable

HLA v1.x has a few serious defects in its design. One major issue is debugging support. HLA v1.x emits MASM code that it runs through MASM in order to produce executable object files. Unfortunately, while this scheme makes the development, testing, and debugging of HLA easier, it effectively eliminates the possibility of using existing source level debugging tools to locate defects in an HLA program[1]. Starting with v1.24, HLA began supporting a new feature to help you debug your programs: the "@trace" pseudo-variable. This appendix will explain the purpose of this compile-time variable and how you can use it to locate defects in your programs.

By default, the compile-time variable *@trace* contains false. You can change its value with any variation of the following statement:

```
?@trace := <<boolean constant expression>>;
```

Generally, "<<boolean constant expression>>" is either *true* or *false*, but you could use any compile-time constant expression to set *@trace's* value.

Once you set *@trace* to true, HLA begins generating extra code in your program In fact, before almost every executable statement HLA injects the following code:

```
    _traceLine_( filename, linenumber );
```

The *filename* parameter is a string specifying the name of the current source file, the *linenumber* parameter is an uns32 value that is the current line number of the file. The _traceLine_ procedure uses this information to display an appropriate trace value while the program is running.

HLA will automatically emit the above procedure call in between almost all instructions appearing in your program[2]. Assuming that the *_traceLine_* procedure simply prints the filename and line number, when you run your application it will create a log of each statement it executes.

You can control the emission of the *_traceLine_* procedure calls in your program by alternately setting *@trace* to *true* or *false* throughout your code. This lets you selectively choose which portions of your code will provide trace information during program execution. This feature is very important because if you're displaying the trace information to the console, your program runs thousands, if not millions, of times slower during the trace operation. It wouldn't do to have to trace through a really long loop in order to trace through following code that you're concerned about. By setting *@trace* to *false* prior to the long loop and setting it to true immedately after the loop, you can execute the loop at full speed and then begin tracing the code you want to check once the loop completes execution.

HLA does not supply the *_traceLine_* procedure; it is your responsibility to write the code for this procedure. The following is a typical implementation:

```
procedure trace( filename:string; linenum:uns32 ); external( "_traceLine_" );
procedure trace( filename:string; linenum:uns32 ); nodisplay;
begin trace;

    pushfd();
    stdout.put( filename, ": #", linenum, nl );
    popfd();

end trace;
```

1. Of course, you can also debug the MASM output using a source level debugger, but this is not a very pleasant experience.
2. HLA does not emit this procedure call between statements that are composed and a few other miscellaneous statements. However, your programs probably get better than 95% coverage so this will be sufficient for most purposes.

This particular procedure simply prints the filename and line number each time HLA calls it. Therefore, you'll get a trace sent to the standard output device that looks something like the following:

```
t.hla: #19
t.hla: #20
t.hla: #21
t.hla: #22
etc.
```

A very important thing to note about this sample implementation of *_traceLine_* is that it preserves the FLAGs register. The *_traceLine_* procedure must preserve all registers including the flags. The example code above only preserves the FLAGS because the *stdout.put* macro always preserves all the registers. However, were this code to modify other registers or call some procedure that modifies some registers, you would want to preserve those register values as well. Remember, HLA calls this procedure between most instructions, if you do not preserve the registers and flags within this procedure, it will adversely affect the running of your program.

This is very important: the *@trace* variable must contain false while HLA is compiling your *_traceLine_* procedure. If HLA emits trace code inside the trace procedure, this will create an infinite loop via infinite recursion which will crash your program. Always make sure *@trace* is false across the compilation of your *_traceLine_* procedure.

Here's a sample program using the *@trace* variable and sample output for the program:

```
program t;
#includeOnce( "stdlib.hhf" )


procedure trace( filename:string; linenum:uns32 ); external( "_traceLine_" );

?@trace := false; // Must be false for TRACE routine.

procedure trace( filename:string; linenum:uns32 ); nodisplay;
begin trace;

    stdout.put( filename, ": #", linenum, nl );

end trace;



?@trace := true;

begin t;


    mov( 0, ecx );              // Line 22
    if( ecx == 0 ) then         // Line 23

        mov( 10, ecx );         // Line 25

    else

        mov( 5, ecx );

    endif;
    while( ecx > 0 ) do         // Line 32

        dec( ecx );             // Line 34

    endwhile;
    for( mov( 0, ecx ); ecx < 5; inc( ecx )) do   // Line 37
```



```
        add( 1, eax );                          // Line 39

    endfor;


end t;

Sample Output:

t.hla: #22
t.hla: #23
t.hla: #25
t.hla: #32
t.hla: #34
t.hla: #34
t.hla: #34
t.hla: #34
t.hla: #34
t.hla: #34
t.hla: #34
t.hla: #34
t.hla: #34
t.hla: #34
t.hla: #37
t.hla: #39
t.hla: #39
t.hla: #39
t.hla: #39
t.hla: #39
```

Although the *_traceLine_* procedure in this example is very simple, you can write as sophisticated a procedure as you like. However, do keep in mind that this code executes between almost every pair of statements your program has, so you should attempt to keep this procedure as short and as fast as possible. However, if desired you can stop the program, request user input, and let the user specify how program control is to proceed.

One very useful purpose for HLA's trace capabilities is tracking down bad pointer references in your program. If you program aborts with an "Illegal memory access" exception, you can pinpoint the offending instruction by turning on the trace option and letting the program run until the exception occurs. The last line number in the trace will be (approximately, within one or two statements) the line number of the offending instruction.

You can also display (global) values within the *_traceLine_* procedure. Printing local values, unfortunately, is problematic since external procedures must appear at lex level one in your program (i.e., you cannot nest the function within some other procedure). It is possible to set up a pointer to a local procedure and call that procedure indirectly from within *_traceLine_*, but this effort is worth it only on rare occasions. Usually it's just easier to stick a "stdout.put" statement directly in the code where you wish to view the output. However, if some variable is being overwritten by some unknown statement in your program, and you don't know where the program is modifying the variable, printing the variables value from the *_traceLine_* procedure can help you pinpoint the statement that overwrites the variable.

Of course, another way to debug HLA programs is to stick a whole bunch of "print" statements in your code. The problem with such statements, however, is that you've got to remember to remove them before you ship your final version of the program. There are few things more embarrasing than having your customer ask you why your program is printing debug messages in the middle of the reports your program is producing for them. You can use conditional compilation (#IF statements) to control the compilation of such statements, but conditional compilation statements tend to clutter up your source code (even more so than the debugging print statements). One solution to this problem is to create a macro, let's call it *myDebugMsg*, that hides all the nasty details. Consider the following code:

```
#macro myDebugMsg( runTimeFlag, msg );
```



```
    #if( @defined( DEBUG ))   // If DEBUG is not defined, then don't emit code.

        #if( DEBUG )           // DEBUG has to be a boolean const equal to true.

            #if( @isConst( runTimeFlag ))

                #if( runTimeFlag )

                    stdout.put( msg, nl );

                #endif

            #else  // runTimeFlag is a run-time variable

                if( runTimeFlag ) then

                    stdout.put( msg, nl );

                endif;

            #endif  // runTimeFlag is/is not Constant

        #endif // DEBUG

    #endif // DEBUG is defined

#endmacro
```

With this macro defined, you can write a statement like the following to print/not print a message at run-time:

```
myDebugMsg( BooleanValue, "This is a debug message" );
```

The *BooleanValue* expression is either a boolean constant expression, a boolean variable, or a register. If this value is true, then the program will print the message; if this value is false, the program does not print the message. Since this is a variable, you can control debugging output at run-time by changing the value of the *BooleanValue* parameter.

As *myDebugMsg* is written, you must define the boolean constant *DEBUG* and set it to true or the program will not compile the debugging statements. If DEBUG is a VAL object, you can actually

You can certainly expand upon this macro by providing support for a variable number of parameters. This would allow you to specify an list of values to display in the *stdout.put* macro invocation. See the chapter on Macros for more information about variable parameter lists in macros if you want to do this.

J.2 The Assert Macro

Another tool you may use to help locate defects in your programs is the assert macro. This macro is available in the "excepts.hhf" header file (included by "stdlib.hhf"). An invocation of this macro takes the following form:

```
assert( boolean_expression );
```

Note that you do not preface the macro with "except." since the macro declaration appears outside the "except" namespace in the excepts.hhf header file. The boolean_expression component is any expression that is legal within an HLA HLL control statement like IF, WHILE, or REPEAT..UNTIL.

The assert macro evaluates the expression and simply returns if the expression is true. If the expression evaluates false, then this macro invocation will raise an exception (ex.AssertionFailed). By liberally sprinkling assert invocations through your code, you can test the behavior of your program and stop execution if an assertion fails (thus helping you to pinpoint problems in your code).

One problem with placing a lot of asserts throughout your code is that each assert takes up a small amount of space and execution time. Fortunately, the HLA Standard Library provides a mechanism by which you may control code generation of the assert macros. The excepts.hhf header files defines a VAL object, ex.NDEBUG, that is initialized with the value false. When this compile-time variable is false, HLA will emit the code for the assert macro; however, if you set this constant to true, then HLA will not emit any code for the assert macro. Therefore, you may liberally place assert macro invocations throughout your code and not worry about their effect on the final version of your program you ship; you can easily remove the impact of all assert macros in your program by sticking in a statement of the form "?ex.NDEBUG:=true;" in your source file.

Note that you may selectively turn asserts on or off by alternately placing "ex.NDEBUG:=false;" and "?ex.NDEBUG:=true;" throughout your code. This allows you to leave some important assertions active in your code, even when you ship the final version.

Since assert raises an exception, you may use the HLA try..exception..endtry statement to catch any exceptions that fail. If you do not handle a specific assertion failure, HLA will abort the program with an appropriate message that tells you the (source) file and line number where the assertion failed.

(More to come someday...)

