

Creating an Extended Loop in HLA

The HLA WHILE loop uses the following syntax:

```
while( <<expression>> ) do  
  
    <<statements>>  
  
endwhile;
```

As you would expect, as long as the expression evaluates true this control construct repeats the execution of the loop body. When the expression evaluates false, control falls through the the first statement beyond the ENDWHILE clause.

Like C/C++ and many other languages, HLA provides limited forms of the GOTO (or jmp) that let you prematurely exit the loop (that is, exit on some other condition than the loop control expression evaluating false). Specifically, the HLA BREAK and BREAKIF(expression) statements provide this capability. Execution of the BREAK statement (or BREAKIF with an expression that evaluates true) transfers control to the first statement beyond the ENDWHILE clause, just as though the WHILE expression had evaluated false.

One drawback to this scheme, that creates some occasional problems, is that the statement following the ENDWHILE clause cannot determine if the WHILE loop terminated because of the loop control expression being false or because of the execution of a BREAK/BREAKIF statement. The classic example of this need occurs which searching a linked list for some particular item. An iterative solution using a WHILE loop can exit the loop in one of two ways: by finding the item of interest or by reaching the end of the list. A typical example is the following:

```
mov( ListHead, esi );  
while( esi <> NULL ) do  
  
    mov( (type node [esi]).FieldOfInterest, eax );  
    breakif( eax = ValueToFind );  
    mov( (type node [esi]).next, esi );  
  
endwhile;
```

Immediately after the ENDWHILE clause the code has to check the value in the ESI register to see if it is NULL in order to determine if the value appears in the list.

A more satisfactory solution would be to add a clause to the WHILE statement, much like the ELSE clause in an IF statement, that executes only if the loop terminates via BREAK. Such a control structure might use the following syntax:

```
while( << expression >> ) do  
  
    <<statements>>  
  
onbreak
```

```
<< statements >>
```

```
endwhile;
```

In this example, if the loop control expression evaluates false, then control transfers to the first statement following the ENDWHILE clause, exactly as before. However, if a BREAK (or BREAKIF) statement forces the loop to exit, control transfers to the ONBREAK clause. Those statements execute and when complete, control falls through to the statement following the ENDWHILE clause. Note that the loop body consists of the statements between the WHILE and ONBREAK clauses; the statements after the ONBREAK clause do not execute on each iteration of the loop.

Implementing this type of control structure in HLA is very easy using HLA's context-free macros. About the only change to the syntax is that you cannot use WHILE and ENDWHILE for macro names since these are HLA reserved words. So this example will use `_while`, `_do`, `_onbreak`, `_break`, `_breakif`, and `_endwhile` as the new "keywords." Here's the macro that implements the above control structure:

```
macro _while( expr ):falseLbl, breakLbl, topOfLoop, hasOnBreak;

    ?hasOnBreak:boolean:=false;
    topOfLoop:
        jf( expr ) falseLbl;

keyword _do;

keyword _break;
    jmp breakLbl;

keyword _breakif( expr2 );
    jt( expr2 ) breakLbl;

keyword _onbreak;
    #if( hasOnBreak )

        #error( "Extra _onbreak clause encountered" )

    #else

        jmp topOfLoop;
        ?hasOnBreak := true;

    breakLbl:

#endif

terminator _endwhile;

    #if( !hasOnBreak )

        jmp topOfLoop;
```

```
        breakLbl :  
  
    #endif  
    falseLbl :  
  
endmacro ;
```

Note that implementing the converse condition, that is, having a special section for exiting the loop if the loop control expression is false, is a trivial modification to the loop above. Indeed, with a little work, it's possible to have both the “onbreak” and “onfalse” clauses active in the same while loop.