

A Modified IF..ELSE..ENDIF Statement

The IF statement is another statement that doesn't always do exactly what you want. Like the `_while.._onbreak.._endwhile` example above, it's quite possible to redefine the IF statement so that it behaves the way we want it to. In this section you'll see how to implement a variant of the IF..ELSE..ENDIF statement that nests differently than the standard IF statement.

HLA's particular variant of the IF statement has several limitations. One of the major limitations is the inability to combine logical sub-expressions using logical conjunction (and) and logical disjunction (or). It is possible to simulate conjunction and disjunction if you carefully structure your code. Consider the following example:

```
// "C" code employing logical-AND operator:
```

```
if( expr1 && expr2 )
{
<< statements >>
}
```

```
// Equivalent HLA version:
```

```
if( expr1 ) then

if( expr2 ) then

<< statements >>

endif;

endif;
```

In both cases ("C" and HLA) the `<< statements >>` block executes only if both `expr1` and `expr2` evaluate true. So other than the extra typing involved, it is often very easy to simulate logical conjunction by using two IF statements in HLA.

There is one very big problem with this scheme. Consider what happens if you modify the "C" code to be the following:

```
// "C" code employing logical-AND operator:
```

```
if( expr1 && expr2 )
{
<< 'true' statements >>
}
else
{
<< 'false' statements >>
}
```

The only way to convert this to HLA (using the standard HLA high level control constructs) is by duplicating the 'false' statements. This introduces a bit of inefficiency into your code. As a result, many HLA programmers will switch to low-level control constructs or HLA's hybrid control structures in order to avoid duplicating code. Unfortunately, dropping down into low-level code may make your program harder to read. It would be nice if you could efficiently handle this situation without making your code unreadable. Fortunately, you can do exactly this by creating a new version of the IF statement using HLA's multi-part macro facilities.

Before describing how to create this new type of IF statement, we must digress for a moment and explore an interesting feature of HLA's multi-part macro expansion: KEYWORD macros do not have to use unique names. Whenever you declare an HLA KEYWORD macro, HLA accepts whatever name you choose. If that name happens to be already defined, then the KEYWORD macro name takes precedence as long as the macro is active (that is, from the point you invoke the macro name until HLA encounters the TERMINATOR macro). Therefore, the KEYWORD macro name hides the previous definition of that name until the termination of the macro. This feature applies even to the original macro name; that is, it is possible to define a KEYWORD macro with the same name as the original macro to which the KEYWORD macro belongs. This is a very useful feature because it allows you to change the definition of the macro within the scope of the opening and terminating invocations of the macro.

Although not pertinent to the IF statement we are construction, you should note that parameter and local symbols in a macro also override any previously defined symbols of the same name. So if you use that symbol between the opening macro and the terminating macro, you will get the value of the local symbol, not the global symbol. E.g.,

```
var
i:int32;
j:int32;
.
.
.
macro abc:i;
?i:text := "j";
.
.
.
terminator xyz;
.
.
.
endmacro
.
.
.
mov( 25, i );
mov( 10, j );
```

```

abc
mov( i, eax );    // Loads j's value (10), not 25 into eax.
xyz;

```

The code above loads 10 into EAX because the "mov(i, eax);" instruction appears between the opening and terminating macros *abc..xyz*. Between those two macros the local definition of *i* takes precedence over the global definition. Since *i* is a text constant that expands to *j*, the aforementioned MOV statement is really equivalent to "mov(j, eax);" That statement, of course, loads 10 into EAX. Since this problem is difficult to see while reading your code, you should choose local symbols in multi-part macros very carefully. A good convention to adopt is to combine your local symbol name with the macro name, e.g.,

```
macro abc : i_abc;
```

You may wonder why HLA allows something to crazy to happen in your source code, in a moment you'll see why this behavior is useful (and now, with this brief message out of the way, back to our regularly scheduled discussion).

Before we digressed to discuss this interesting feature in HLA multi-part macros, we were trying to figure out how to efficiently simulate the conjunction and disjunction operators in an IF statement without resorting to low-level code. The problem in the example appearing earlier in this section is that you would have to duplicate some code in order to convert the IF..ELSE statement properly. The following code shows this problem:

```
// "C" code employing logical-AND operator:
```

```

if( expr1 && expr2 )
{
<< 'true' statements >>
}
else
{
<< 'false' statements >>
}

```

```
// Corresponding HLA code using the "nested-IF" algorithm:
```

```

if( expr1 ) then
if( expr2 ) then
<< 'true' statements >>
else
<< 'false' statements >>

```

```
endif;  
  
else  
  
<< 'false' statements >>  
  
endif;
```

Note that this code must duplicate the "<< 'false' statements >>" section if the logic is to exactly match the original "C" code. This means that the program will be larger and harder to read than is absolutely necessary.

One solution to this problem is to create a new kind of IF statement that doesn't nest the same way standard IF statements nest. In particular, if we define the statement such that all IF clauses nested with an outer IF.ENDIF block share the same ELSE and ENDIF clauses. If this were the case, then you could implement the code above as follows:

```
if( expr1 ) then  
  
if( expr2 ) then  
  
<< 'true' statements >>  
  
else  
  
<< 'false' statements >>  
  
endif;
```

If *expr1* is false, control immediately transfers to the ELSE clause. If the value of *expr1* is true, the control falls through to the next IF statement.

If *expr2* evaluates false, then the program jumps to the single ELSE clause that all IFs share in this statement. Notice that a single ELSE clause (and corresponding 'false' statements) appear in this code; hence the code does not necessarily expand in size. If *expr2* evaluates true, then control falls through to the 'true' statements, exactly like a standard IF statement.

Notice that the nested IF statement above does not have a corresponding ENDIF. Like the ELSE clause, all nested IFs in this structure share the same ENDIF. Syntactically, there is no need to end the nested IF statement; the end of the THEN section ends with the ELSE clause, just as the outer IF statement's THEN block ends.

Of course, we can't actually define a new macro named "if" because you cannot redefine HLA reserved words. Nor would it be a good idea to do so even if these were legal (since it would

make your programs very difficult to comprehend if the IF keyword had different semantics in different parts of the program. The following program uses the identifiers "_if", "_then", "_else", and "_endif" instead. It is questionable if these are good identifiers in production code (perhaps something a little more different would be appropriate). The following code example uses these particular identifiers so you can easily correlate them with the corresponding high level statements.

```
/*
 *
 * if.hla
 *
 * This program demonstrates a modification of
 * the IF..ELSE..ENDIF statement using HLA's
 * multi-part macros.
 */

program newIF;
#include( "stdlib.hhf" )

// Macro implementation of new form of if..then..else..endif.
//
// In this version, all nested IF statements transfer control
// to the same ELSE clause if any one of them have a false
// boolean expression. Syntax:
//
// _if( expression ) _then
//
//     <<statements including nested _if clauses>>
//
// _else // this is optional
//
//     <<statements, but _if clauses are not allowed here>>
//
// _endif
//
//
// Note that nested _if clauses do not have a corresponding
// _endif clause. This is because the single _else and/or
// _endif clauses terminate all the nested _if clauses
// including the first one. Of course, once the code
// encounters an _endif another _if statement may begin.
```

```

// Macro to handle the main "_if" clause.
// This code just tests the expression and jumps to the _else
// clause if the expression evaluates false.

macro _if( ifExpr ):elseLbl, hasElse, ifDone;

    ?hasElse := false;
    jf(ifExpr) elseLbl;

// Just ignore the _then keyword.

keyword _then;

// Nested _if clause (yes, HLA lets you replace the main
// macro name with a keyword macro). Identical to the
// above _if implementation except this one does not
// require a matching _endif clause. The single _endif
// (matching the first _if clause) terminates all nested
// _if clauses as well as the main _if clause.

keyword _if( nestedIfExpr );
    jf( nestedIfExpr ) elseLbl;

    // If this appears within the _else section, report
    // an error (we don't allow _if clauses nested in
    // the else section, that would create a loop).

    #if( hasElse )

        #error( "All _if clauses must appear before the _else
clause" )

    #endif

// Handle the _else clause here. All we need to is check to
// see if this is the only _else clause and then emit the
// jmp over the else section and output the elseLbl target.

keyword _else;
    #if( hasElse )

        #error( "Only one _else clause is legal per _if.._endif" )

```

```

#else

    // Set hasElse true so we know that we've seen an _else
    // clause in this statement.

    ?hasElse := true;
    jmp ifDone;
    elseLbl:

#endif

// _endif has two tasks. First, it outputs the "ifDone" label
// that _else uses as the target of its jump to skip over the
// else section. Second, if there was no else section, this
// code must emit the "elseLbl" label so that the false condi-
// tional(s)
// in the _if clause(s) have a legal target label.

terminator _endif;

    ifDone:
    #if( !hasElse )

        elseLbl:

    #endif

endmacro;

static
    tr:boolean := true;
    f:boolean := false;

begin newIF;

    // Real quick demo of the _if statement:

    _if( tr ) _then

        _if( tr ) _then
        _if( f ) _then

            stdout.put( "error" nl );

```

```
    _else  
        stdout.put( "Success" );  
    _endif  
end newIF;
```

Just in case you're wondering, this program prints "Success" and then quits. This is because the nested "_if" statements are equivalent to the expression "true && true && false" which, of course, is false. Therefore, the "_else" portion of this code should execute.

The only surprise in this macro is the fact that it redefines the *_if* macro as a keyword macro upon invocation of the main *_if* macro. The reason this code does this is so that any nested *_if* clauses do not require a corresponding *_endif* and don't support an *_else* clause.

Implementing an ELSEIF clause introduces some difficulties, hence its absence in this example. The design and implementation of an ELSEIF clause is left to the more serious reader¹.

1. I.e., I don't even want to have to think about this problem!