Macros Chapter Eight

8.1 Chapter Overview

This chapter continues where the previous chapter left off – continuing to discuss the HLA compile time language. This chapter discusses what is, perhaps, the most important component of the HLA compile-time language, macros. Many people judge the power of an assembler by the power of its macro processing capabilities. If you happen to be one of these people, you'll probably agree that HLA is one of the more powerful assemblers on the planet after reading this chapter; because HLA has one of the most powerful macro processing facilities of any computer language processing system.

8.2 Macros (Compile-Time Procedures)

Macros are symbols that a language processor replaces with other text during compilation. Macros are great devices for replacing long repetitive sequences of text with much shorter sequences of text. In additional to the traditional role that macros play (e.g., "#define" in C/C++), HLA's macros also serve as the equivalent of a compile-time language procedure or function. Therefore, macros are very important in HLA's compile-time language; just as important as functions and procedures are in other high level languages.

Although macros are nothing new, HLA's implementation of macros far exceeds the macro processing capabilities of most other programming languages (high level or low level). The following sections explore HLA's macro processing facilities and the relationship between macros and other HLA CTL control constructs.

8.2.1 Standard Macros

HLA supports a straight-forward macro facility that lets you define macros in a manner that is similar to declaring a procedure. A typical, simple, macro declaration takes the following form:

```
#macro macroname;

    << macro body >>

#endmacro;
```

Although macro and procedure declarations are similar, there are several immediate differences between the two that are obvious from this example. First, of course, macro declarations use the reserved word #MACRO rather than PROCEDURE. Second, you do not begin the body of the macro with a "BEGIN *macroname*;" clause. This is because macros don't have a declaration section like procedures so there is no need for a keyword that separates the macro declarations from the macro body. Finally, you will note that macros end with the "#ENDMACRO" clause rather than "END *macroname*;" The following is a concrete example of a macro declaration:

```
#macro neg64;

    neg( edx );
    neg( eax );
    sbb( 0, edx );

#endmacro;
```


Execution of this macro's code will compute the two's complement of the 64-bit value in EDX:EAX (see "Extended Precision NEG Operations" on page 872).

To execute the code associated with *neg64*, you simply specify the macro's name at the point you want to execute these instructions, e.g.,

```
mov( (type dword i64), eax );
mov( (type dword i64+4), edx );
neg64;
```

Note that you do *not* follow the macro's name with a pair of empty parentheses as you would a procedure call (the reason for this will become clear a little later).

Other than the lack of parentheses following *neg64*'s invocation[1] this looks just like a procedure call. You could implement this simple macro as a procedure using the following procedure declaration:

```
procedure neg64p;
begin neg64p;

    neg( edx );
    neg( eax );
    sbb( 0, edx );

end neg64p;
```

Note that the following two statements will both negate the value in EDX:EAX:

```
neg64;              neg64p();
```

The difference between these two (i.e., the macro invocation versus the procedure call) is the fact that macros expand their text in-line whereas a procedure call emits a call to the associate procedure elsewhere in the text. That is, HLA replaces the invocation "neg64;" directly with the following text:

```
neg( edx );
neg( eax );
sbb( 0, edx );
```

On the other hand, HLA replaces the procedure call "neg64p();" with the single call instruction:

```
call neg64p;
```

Presumably, you've defined the *neg64p* procedure earlier in the program.

You should make the choice of macro versus procedure call on the basis of efficiency. Macros are slightly faster than procedure calls because you don't execute the CALL and corresponding RET instructions. On the other hand, the use of macros can make your program larger because a macro invocation expands to the text of the macro's body on each invocation. Procedure calls jump to a single instance of the procedure's body. Therefore, if the macro body is large and you invoke the macro several times throughout your program, it will make your final executable much larger. Also, if the body of your macro executes more than a few simple instructions, the overhead of a CALL/RET sequence has little impact on the overall execution time of the code, so the execution time savings are nearly negligible. On the other hand, if the body of a procedure is very short (like the *neg64* example above), you'll discover that the macro implementation is much faster and doesn't expand the size of your program by much. Therefore, a good rule of thumb is

- Use macros for short, time-critical program units. Use procedures for longer blocks of code and when execution time is not as critical.

Macros have many other disadvantages over procedures. Macros cannot have local (automatic) variables, macro parameters work differently than procedure parameters, macros don't support (run-time) recur-

1. To differentiate macros and procedures, this text will use the term *invocation* when describing the use of a macro and *call* when describing the use of a procedure.

sion, and macros are a little more difficult to debug than procedures (just to name a few disadvantages). Therefore, you shouldn't really use macros as a substitute for procedures except in some rare situations.

8.2.2 Macro Parameters

Like procedures, macros allow you to define parameters that let you supply different data on each macro invocation. This lets you write generic macros whose behavior can vary depending on the parameters you supply. By processing these macro parameters at compile-time, you can write very sophisticated macros.

Macro parameter declaration syntax is very straight-forward. You simply supply a list of parameter names within parentheses in a macro declaration:

```
#macro neg64( reg32HO, reg32LO );

    neg( reg32HO );
    neg( reg32LO );
    sbb( 0, reg32HO );

#endmacro;
```

Note that you do not associate a data type with a macro parameter like you do procedural parameters. This is because HLA macros are always *text* objects. The next section will explain the exact mechanism HLA uses to substitute an actual parameter for a formal parameter.

When you invoke a macro, you simply supply the actual parameters the same way you would for a procedure call:

```
neg64( edx, eax );
```

Note that a macro invocation that requires parameters expects you to enclose the parameter list within parentheses.

8.2.2.1 Standard Macro Parameter Expansion

As the previous section explains, HLA automatically associates the type *text* with macro parameters. This means that during a macro expansion, HLA substitutes the text you supply as the actual parameter everywhere the formal parameter name appears. The semantics of "pass by textual substitution" are a little different than "pass by value" or "pass by reference" so it is worthwhile exploring those differences here.

Consider the following macro invocations, using the *neg64* macro from the previous section:

```
neg64( edx, eax );
neg64( ebx, ecx );
```

These two invocations expand into the following code:

```
// neg64(edx, eax );

    neg( edx );
    neg( eax );
    sbb( 0, edx );

// neg64( ebx, ecx );

    neg( ebx );
    neg( ecx );
    sbb( 0, ebx );
```


Note that macro invocations do not make a local copy of the parameters (as pass by value does) nor do they pass the address of the actual parameter to the macro. Instead, a macro invocation of the form "neg64(edx, eax);" is equivalent to the following:

```
?reg32HO: text := "edx";
?reg32LO: text := "eax";

neg( reg32HO );
neg( reg32LO );
sbb( 0, reg32HO );
```

Of course, the text objects immediately expand their string values in-line, producing the former expansion for "neg64(edx, eax);".

Note that macro parameters are not limited to memory, register, or constant operands as are instruction or procedure operands. Any text is fine as long as its expansion is legal wherever you use the formal parameter. Similarly, formal parameters may appear anywhere in the macro body, not just where memory, register, or constant operands are legal. Consider the following macro declaration and sample invocations:

```
#macro chkError( instr, jump, target );

    instr;
    jump target;

#endmacro;


    chkError( cmp( eax, 0 ), jnl, RangeError );    // Example 1
        ...
    chkError( test( 1, bl ), jnz, ParityError );   // Example 2


// Example 1 expands to

    cmp( eax, 0 );
    jnl RangeError;

// Example 2 expands to

    test( 1, bl );
    jnz ParityError;
```

In general, HLA assumes that all text between commas constitutes a single macro parameter. If HLA encounters any opening "bracketing" symbols (left parentheses, left braces, or left brackets) then it will include all text up to the appropriate closing symbol, ignoring any commas that may appear within the bracketing symbols. This is why the *chkError* invocations above treat "cmp(eax, 0)" and "test(1, bl)" as single parameters rather than as a pair of parameters. Of course, HLA does not consider commas (and bracketing symbols) within a string constant as the end of an actual parameter. So the following macro and invocation is perfectly legal:

```
#macro print( strToPrint );

    stdout.out( strToPrint );

#endmacro;
    .
    .
    .
    print( "Hello, world!" );
```

HLA treats the string "Hello, world!" as a single parameter since the comma appears inside a literal string constant, just as your intuition suggests.

If you are unfamiliar with textual macro parameter expansion in other languages, you should be aware that there are some problems you can run into when HLA expands your actual macro parameters. Consider the following macro declaration an invocation:

```
#macro Echo2nTimes( n, theStr );

    ?echoCnt: uns32 := 0;
    #while( echoCnt < n*2 )

        #print( theStr )
        ?echoCnt := echoCnt + 1;

    #endwhile

#endmacro;

    .
    .
    .
    Echo2nTimes( 3+1, "Hello" );
```

This example displays "Hello" five times during compilation rather than the eight times you might intuitively expect. This is because the #WHILE statement above expands to

```
    #while( echoCnt < 3+1*2 )
```

The actual parameter for *n* is "3+1", since HLA expands this text directly in place of *n*, you get the text above. Of course, at compile time HLA computes "3+1*2" as the value five rather than as the value eight (which you would get had HLA passed this parameter by value rather than by textual substitution).

The common solution to this problem, when passing numeric parameters that may contain compile-time expressions, is to surround the formal parameter in the macro with parentheses. E.g., you would rewrite the macro above as follows:

```
#macro Echo2nTimes( n, theStr );

    ?echoCnt: uns32 := 0;
    #while( echoCnt < (n)*2 )

        #print( theStr )
        ?echoCnt := echoCnt + 1;

    #endwhile

#endmacro;
```

The previous invocation would expand to the following code:

```
    ?echoCnt: uns32 := 0;
    #while( echoCnt < (3+1)*2 )

        #print( theStr )
        ?echoCnt := echoCnt + 1;

    #endwhile
```

This version of the macro produces the intuitive result.

If the number of actual parameters does not match the number of formal parameters, HLA will generate a diagnostic message during compilation. Like procedures, the number of actual parameters must agree with the number of formal parameters. If you would like to have optional macro parameters, then keep reading...

8.2.2.2 Macros with a Variable Number of Parameters

You may have noticed by now that some HLA macros don't require a fixed number of parameters. For example, the *stdout.put* macro in the HLA Standard Library allows one or more actual parameters. HLA uses a special array syntax to tell the compiler that you wish to allow a variable number of parameters in a macro parameter list. If you follow the last macro parameter in the formal parameter list with "[ ]" then HLA will allow a variable number of actual parameters (zero or more) in place of that formal parameter. E.g.,

```
#macro varParms( varying[] );

    << macro body >>

#endmacro;
    .
    .
    .
    varParms( 1 );
    varParms( 1, 2 );
    varParms( 1, 2, 3 );
    varParms();
```

Note, especially, the last example above. If a macro has any formal parameters, you must supply parentheses with the macro list after the macro invocation. This is true even if you supply zero actual parameters to a macro with a varying parameter list. Keep in mind this important difference between a macro with no parameters and a macro with a varying parameter list but no actual parameters.

When HLA encounters a formal macro parameter with the "[ ]" suffix (which must be the last parameter in the formal parameter list), HLA creates a constant string array and initializes that array with the text associated with the remaining actual parameters in the macro invocation. You can determine the number of actual parameters assigned to this array using the @ELEMENTS compile-time function. For example, "@elements(varying)" will return some value, zero or greater, that specifies the total number of parameters associated with that parameter. The following declaration for *varParms* demonstrates how you might use this:

```
#macro varParms( varying[] );

    ?vpCnt := 0;
    #while( vpCnt < @elements( varying ))

        #print( varying[ vpCnt ] )
        ?vpCnt := vpCnt + 1;

    #endwhile

#endmacro;
    .
    .
    .
    varParms( 1 );          // Prints "1" during compilation.
    varParms( 1, 2 );       // Prints "1" and "2" on separate lines.
    varParms( 1, 2, 3 );    // Prints "1", "2", and "3" on separate lines.
    varParms();             // Doesn't print anything.
```

Since HLA doesn't allow arrays of *text* objects, the varying parameter must be an array of strings. This, unfortunately, means you must treat the varying parameters differently than you handle standard macro parameters. If you want some element of the varying string array to expand as text within the macro body, you can always use the @TEXT function to achieve this. Conversely, if you want to use a non-varying for-

mal parameter as a string object, you can always use the @STRING:name operator. The following example demonstrates this:

```
#macro ReqAndOpt( Required, optional[] );

    ?@text( optional[0] ) := @string:ReqAndOpt;
    #print( @text( optional[0] ))

#endmacro;

    .
    .
    .
    ReqAndOpt( i, j );

// The macro invocation above expands to

    ?@text( "j" ) := @string:i;
    #print( "j" )

// The above further expands to

    j := "i";
    #print( j )

// The above simply prints "i" during compilation.
```

Of course, it would be a good idea, in a macro like the above, to verify that there are at least two parameters before attempting to reference element zero of the *optional* parameter. You can easily do this as follows:

```
#macro ReqAndOpt( Required, optional[] );

    #if( @elements( optional ) > 0 )

        ?@text( optional[0] ) := @string:ReqAndOpt;
        #print( @text( optional[0] ))

    #else

        #error( "ReqAndOpt must have at least two parameters" )

    #endif

#endmacro;
```

8.2.2.3 Required Versus Optional Macro Parameters

As noted in the previous section, HLA requires exactly one actual parameter for each non-varying formal macro parameter. If there is no varying macro parameter (and there can be at most one) then the number of actual parameters must exactly match the number of formal parameters. If a varying formal parameter is present, then there must be at least as many actual macro parameters as there are non-varying (or required) formal macro parameters. If there is a single, varying, actual parameter, then a macro invocation may have zero or more actual parameters.

There is one big difference between a macro invocation of a macro with no parameters and a macro invocation of a macro with a single, varying, parameter that has no actual parameters: the macro with the

varying parameter list must have an empty set of parentheses after it while the macro invocation of the macro without any parameters does not allow this. You can use this fact to your advantage if you wish to write a macro that doesn't have any parameters but you want to follow the macro invocation with "( )" so that it matches the syntax of a procedure call with no parameters. Consider the following macro:

```
#macro neg64( JustForTheParens[] );

    #if( @elements( JustForTheParens ) = 0 )

        neg( edx );
        neg( eax );
        sbb( 0, edx );

    #else

        #error( "Unexpected operand(s)" )

    #endif

#endmacro;
```

The macro above allows invocations of the form "neg64();" using the same syntax you would use for a procedure call. This feature is useful if you want the syntax of your parameterless macro invocations to match the syntax of a parameterless procedure call. It's not a bad idea to do this, just in the off chance you need to convert the macro to a procedure at some point (or vice versa, for that matter).

If, for some reason, it is more convenient to operate on your macro parameters as *string* objects rather than *text* objects, you can specify a single varying parameter for the macro and then use #IF and @ELEMENTS to enforce the number of required actual parameters.

8.2.2.4 The "#(" and ")#" Macro Parameter Brackets

Once in a (really) great while, you may want to include arbitrary commas (i.e., outside a bracketing pair) within a macro parameter. Or, perhaps, you might want to include other text as part of a macro expansion that HLA would normally process before storing away the text as the value for the formal parameter[2]. The "#(" and ")#" bracketing symbols tell HLA to collect all text, except for surrounding whitespace, between these two symbols and treat that text as a single parameter. Consider the following macro:

```
#macro PrintName( theParm );

    ?theName := @string:theParm;
    #print( theName )

#endmacro;
```

Normally, this macro will simply print the text of the actual parameter you pass to it. So were you to invoke the macro with "PrintName(j);" HLA would simply print "j" during compilation. This occurs because HLA associates the parameter data ("j") with the string value for the text object *theParm*. The macro converts this text data to a string, puts the string data in *theName*, and then prints this string.

Now consider the following statements:

```
?tt:text := "j";
PrintName( tt );
```

This macro invocation will also print "j". The reason is that HLA expands text constants immediately upon encountering them. So after this expansion, the invocation above is equivalent to

2. For example, HLA will normally expand all *text* objects prior to the creation of the data for the formal parameter. You might not want this expansion to occur.


```
PrintName( j );
```

So this macro invocation prints "j" for the same reason the last example did.

What if you want the macro to print "tt" rather than "j"? Unfortunately, HLA's *eager* evaluation of the text constant gets in the way here. However, if you bracket "tt" with the "#(" and ")#" brackets, you can instruct HLA to *defer* the expansion of this text constant until it actually expands the macro. I.e., the following macro invocation prints "tt" during compilation:

```
PrintName( #( tt )# );
```

Note that HLA allows any amount of arbitrary text within the "#(" and ")#" brackets. This can include commas and other arbitrary text. The following macro invocation prints "Hello, World!" during compilation:

```
PrintName( #( Hello, World! )# );
```

Normally, HLA would complain about the mismatched number of parameters since the comma would suggest that there are two parameters here. However, the deferred evaluation brackets tell HLA to consider all the text between the "#(" and ")#" symbols as a single parameter.

8.2.2.5 Eager vs. Deferred Macro Parameter Evaluation

HLA uses two schemes to process macro parameters. As you saw in the previous section, HLA uses *eager* evaluation when processing text constants appearing in a macro parameter list. You can force *deferred* evaluation of the text constant by surrounding the text with the "#(" and ")#" brackets. For other types of operands, HLA uses deferred macro parameter evaluation. This section discusses the difference between these two forms and describes how to force eager evaluation if necessary.

Eager evaluation occurs while HLA is collecting the text associated with each macro parameter. For example, if "T" is a text constant containing the string "U" and "M" is a macro, then when HLA encounters "M(T)" it will first expand "T" to "U". Then HLA processes the macro invocation "M(U)" as though you had supplied the text "U" as the parameter to begin with.

Deferred evaluation of macro parameters means that HLA does not process the parameter(s), but rather passes the text unchanged to the macro. Any expansion of the text associated with macro parameters occurs within the macro itself. For example, if M and N are both macros accepting a single parameter, then the invocation "M(N(0))" defers the evaluation of "N(0)" until HLA processes the macro body. It does not evaluate "N(0)" first and pass this expansion as a parameter to the macro. The following program demonstrates eager and deferred evaluation:

```
// This program demonstrates the difference
// between deferred and eager macro parameter
// processing.


program EagerVsDeferredEvaluation;

macro ToDefer( tdParm );

    #print( "ToDefer: ", @string:tdParm )
    @string:tdParm

endmacro;

macro testEVD( theParm );

    #print( "testEVD:'", @string:theParm, "'" )
```



```
endmacro;

const

    txt:text := "Hello";
    str:string := "there";


begin EagerVsDeferredEvaluation;

    testEVD( str );                 // Deferred evaluation.
    testEVD( txt );                 // Eager evaluation.
    testEVD( ToDefer( World ) );    //Deferred evaluation.

end EagerVsDeferredEvaluation;
```

Program 8.1 Eager vs. Deferred Macro Parameter Evaluation

Note that the macro *testEVD* outputs the text associated with the formal parameter as a string during compilation. When you compile Program 8.1 it produces the following output:

```
testEVD:'Hello'
testEVD:'Hello'
testEVD:'ToDefer( World )'
```

The first line prints 'Hello' because this is the text supplied as a parameter for the first call to *testEVD*. Since this is a string constant, not a text constant, HLA uses deferred evaluation. This means that it passes the text appearing between the parentheses unchanged to the *testEVD* macro. That text is "Hello" hence the same output as the parameter text.

The second *testEVD* invocation prints 'Hello'. This is because the macro parameter, *txt*, is a text object. HLA eagerly processes text constants before invoking the macro. Therefore, HLA translates "testEVD(txt)" to "testEVD(Hello)" prior to invoking the macro. Since the macro parameter text is now "Hello", that's what HLA prints during compilation while processing this macro.

The third invocation of testEVD above is semantically identical to the first. It is present just to demonstrate that HLA defers processing macros just like it defers the processing of everything else except text constants.

Although the code in Program 8.1 does not actually evaluate the *ToDefer* macro invocation, this is only because the body of *testEVD* does not directly use the parameter. Instead, it converts *theParm* to a string and prints its value. Had this code actually referred to *theParm* in an expression (or as a statement), then HLA would have invoked *ToDefer* and let it do its job. Consider the following modification to the above program:

```
// This program demonstrates the difference
// between deferred and eager macro parameter
// processing.


program DeferredEvaluation;

macro ToDefer( tdParm );

    @string:tdParm

endmacro;
```



```
macro testEVD( theParm );

    #print( "Hello ", theParm )

endmacro;


begin DeferredEvaluation;

    testEVD( ToDefer( World ) );

end DeferredEvaluation;
```

Program 8.2 Deferred Macro Parameter Expansion

The macro invocation "testEVD(ToDefer(World));" defers the evaluation of its parameter. Therefore, the actual parameter *theParm* is a text object containing the string "ToDefer(World)". Inside the testEVD macro, HLA encounters theParm and expands it to this string, i.e.,

```
#print( "Hello ", theParm )
```

expands to

```
#print( "Hello ", ToDefer( World ) )
```

When HLA processes the #PRINT statement, it eagerly processes all parameters. Therefore, HLA expands the statement above to

```
#print( "Hello ", "World" )
```

since "ToDefer(World)" expands to *@string:tdParm* and that expands to "World".

Most of the time, the choice between deferred and eager evaluation produces the same result. In Program 8.2, for example, it doesn't matter whether the *ToDefer* macro expansion is eager (thus passing the string "World" as the parameter to *testEVD*) or deferred. Either mechanism produces the same output.

There are situations where deferred evaluation is not interchangeable with eager evaluation. The following program demonstrates a problem that can occur when you use deferred evaluation rather than eager evaluation. In this example the program attempts to pass the current line number in the source file as a parameter to a macro. This does not work because HLA expands (and evaluates) the @LINENUMBER function call inside the macro on every invocation. Therefore, this program always prints the same line number (eight) regardless of the invocation line number:

```
// This program a situation where deferred
// evaluation fails to work properly.

program DeferredFails;

macro printAt( where );

    #print( "at line ", where )

endmacro;
```



```
begin DeferredFails;

    printAt( @linenumber );
    printAt( @lineNumber );

end DeferredFails;
```

Program 8.3 An Example Where Deferred Evaluation Fails to Work Properly

Intuitively, this program should print:

```
at line 14
at line 15
```

Unfortunately, because of deferred evaluation, the two *printAt* invocations simply pass the text "@linenumber" as the actual parameter value rather than the string representing the line numbers of these two statements in the program. Since the formal parameter always expands to @LINENUMBER on the same line (line eight), this program always prints the same line number regardless of the line number of the macro invocation.

If you need an eager evaluation of a macro parameter there are three ways to achieve this. First of all, of course, you can specify a *text* object as a macro parameter and HLA will immediately expand that object prior to passing it as the macro parameter. The second option is to use the @TEXT function (with a string parameter). HLA will also immediately process this object, expanding it to the appropriate text, prior to processing that text as a macro parameter. The third option is to use the @EVAL pseudo-function. Within a macro invocation's parameter list, the @EVAL function instructs HLA to evaluate the @EVAL parameter prior to passing the text to the macro. Therefore, you can correct the problem in Program 8.3 by using the following code (which properly prints at "at line 14" and "at line 15"):

```
// This program a situation where deferred
// evaluation fails to work properly.

program EvalSucceeds;

macro printAt( where );

    #print( "at line ", where )

endmacro;

begin EvalSucceeds;

    printAt( @eval( @linenumber ));
    printAt( @eval( @lineNumber ));

end EvalSucceeds;
```

Program 8.4 Demonstration of @EVAL Compile-time Function

In addition to immediately processing built-in compiler functions like @LINENUMBER, the @EVAL pseudo-function will also invoke any macros appearing in the @EVAL parameter. @EVAL usually leaves other values unchanged.

8.2.3 Local Symbols in a Macro

Consider the following macro declaration:

```
macro JZC( target );

        jnz NotTarget;
        jc  target;
    NotTarget:

endmacro;
```

The purpose of this macro is to simulate an instruction that jumps to the specified target location if the zero flag is set *and* the carry flag is set. Conversely, if either the zero flag is clear or the carry flag is clear this macro transfers control to the instruction immediately following the macro invocation.

There is a serious problem with this macro. Consider what happens if you use this macro more than once in your program:

```
        JZC( Dest1 );
            .
            .
            .
        JZC( Dest2 );
            .
            .
            .
```

The macro invocations above expand to the following code:

```
        jnz NotTarget;
        jc Dest1;
NotTarget:
            .
            .
            .
        jnz NotTarget;
        jc Dest2;
NotTarget:
            .
            .
            .
```

The problem with the expansion of these two macro invocations is that they both emit the same label, *NotTarget*, during macro expansion. When HLA processes this code it will complain about a duplicate symbol definition. Therefore, you must take care when defining symbols inside a macro because multiple invocations of that macro may lead to multiple definitions of that symbol.

HLA's solution to this problem is to allow the use of *local symbols* within a macro. Local macro symbols are unique to a specific invocation of a macro. For example, had *NotTarget* been a local symbol in the *JZC* macro invocations above, the program would have compiled properly since HLA treats each occurrence of *NotTarget* as a unique symbol.

HLA does not automatically make internal macro symbol definitions local to that macro[3]. Instead, you must explicitly tell HLA which symbols must be local. You do this in a macro declaration using the following generic syntax:

```
#macro macroname ( optional_parameters ) : optional_list_of_local_names ;
    << macro body >>
#endmacro;
```

3. Sometimes you actually want the symbols to be global.

The list of local names is a sequence of one or more HLA identifiers separated by commas. Whenever HLA encounters this name in a particular macro invocation it automatically substitutes some unique name for that identifier. For each macro invocation, HLA substitutes a different name for the local symbol.

You can correct the problem with the *JZC* macro by using the following macro code:

```
#macro JZC( target ):NotTarget;

        jnz NotTarget;
        jc  target;
    NotTarget:

#endmacro
;
```

Now whenever HLA processes this macro it will automatically associate a unique symbol with each occurrence of *NotTarget*. This will prevent the duplicate symbol error that occurs if you do not declare *NotTarget* as a local symbol.

HLA implements local symbols by substituting a symbol of the form "*_nnnn_*" (where *nnnn* is a four-digit hexadecimal number) wherever the local symbol appears in a macro invocation. For example, a macro invocation of the form "JZC(SomeLabel);" might expand to

```
       jnz _010A_;
       jc SomeLabel;
_010A_:
```

For each local symbol appearing within a macro expansion HLA will generate a unique temporary identifier by simply incrementing this numeric value for each new local symbol it needs. As long as you do not explicitly create labels of the form "*_nnnn_*" (where *nnnn* is a hexadecimal value) there will never be a conflict in your program. HLA explicitly reserves all symbols that begin and end with a single underscore for its own private use (and for use by the HLA Standard Library). As long as you honor this restriction, there should be no conflicts between HLA local symbol generation and labels in your own programs since all HLA-generated symbols begin and end with a single underscore.

HLA implements local symbols by effectively converting that local symbol to a text constant that expands to the unique symbol HLA generates for the local label. That is, HLA effectively treats local symbol declarations as indicated by the following example:

```
#macro JZC( target );
    ?NotTarget:text := "_010A_";

        jnz NotTarget;
        jc  target;
    NotTarget:

#endmacro;
```

Whenever HLA expands this macro it will substitute "_010A_" for each occurrence of *NotTarget* it encounters in the expansion. This analogy isn't perfect because the text symbol *NotTarget* in this example is still accessible after the macro expansion whereas this is not the case when defining local symbols within a macro. But this does give you an idea of how HLA implements local symbols.

One important consequence of HLA's implementation of local symbols within a macro is that HLA will produce some puzzling error messages if an error occurs on a line that uses a local symbol. Consider the following (incorrect) macro declaration:

```
#macro LoopZC( TopOfLoop ): ExitLocation;

       jnz ExitLocation;
       jc TopOfLoop;

#endmacro;
```


Note that in this example the macro does not define the *ExitLocation* symbol even though there is a jump (JNZ) to this label. If you attempt to compile this program, HLA will complain about an undefined statement label and it will state that the symbol is something like "_010A_" rather than *ExitLocation*.

Locating the exact source of this problem can be challenging since HLA cannot report this error until the end of the procedure or program in which *LoopZC* appears (long after you've invoked the macro). If you have lots of macros with lots of local symbols, locating the exact problem is going to be a lot of work; your only option is to carefully analyze the macros you do call (perhaps by commenting them out of your program one by one until the error goes away) to discover the source of the problem. Once you determine the offending macro, the next step is to determine which local symbol is the culprit (if the macro contains more than one local symbol). Because tracking down bugs associated with local symbols can be tough, you should be especially careful when using local symbols within a macro.

Because local symbols are effectively text constants, don't forget that HLA eagerly processes any local symbols you pass as parameters to other macros. To see this effect, consider the following sample program:

```
// LocalDemo.HLA
//
// This program demonstrates the effect
// of passing a local macro symbol as a
// parameter to another macro.  Remember,
// local macro symbols are text constants
// so HLA eager evaluates them when they
// appear as macro parameters.

program LocalExpansionDemo;

macro printIt( what );

    #print( @string:what )
    #print( what )

endmacro;

macro LocalDemo:local;

    ?local:string := "localStr";

    printIt( local );       // Eager evaluation, passes "_nnnn_".
    printIt( #( local )# )  // Force deferred evaluation, passes "local".

endmacro;

begin LocalExpansionDemo;

    LocalDemo;

end LocalExpansionDemo;
```

Program 8.5 Local Macro Symbols as Macro Parameters

Inside *LocalDemo* HLA associates the unique symbol "_0001_" (or something similar) with the local symbol *local*. Next, HLA defines "_0001_" to be a string constant and associates the text "localStr" with this constant.

The first *printIt* macro invocation expands to "printIt(_0001_)" because HLA eagerly processes text constants in macro parameter lists (remember, local symbols are, effectively, text constants). Therefore, *printIt*'s *what* parameter contains the text "_0001_" for this first invocation. Therefore, the first #PRINT statement prints this textual data ("_0001_") and the second print statement prints the value associated with "_0001_" which is "localStr".

The second *printIt* macro invocation inside the *LocalDemo* macro explicitly forces HLA to use deferred evaluation since it surrounds *local* with the "#(" and ")#" bracketing symbols. Therefore, HLA associates the text "local" with *printIt*'s formal parameter rather than the expansion "_0001_". Inside *printIt*, the first #PRINT statement displays the text associated with the *what* parameter (which is "local" at this point). The second #PRINT statement expands *what* to produce "local". Since *local* is a currently defined text constant (defined within *LocalDemo* that invokes *printIt*), HLA expands this text constant to produce "_0001_". Since "_0001_" is a string constant, HLA prints the specified string ("localStr") during compilation. The complete output during compilation is

```
_0001_
localStr
local
localStr
```

Discussing the expansion of local symbols may seem like a lot of unnecessary detail. However, as your macros become more complex you may run into difficulties with your code based on the way HLA expands local symbols. Hence it is necessary to have a good grasp on how HLA processes these symbols.

Quick tip: if you ever need to generate a unique label in your program, you can use HLA local symbol facility to achieve this. Normally, you can only reference HLA's local symbols within the macro that defines the symbol. However, you can convert that local symbol to a string and process that string in your program as the following simple program demonstrates:

```
// UniqueSymbols.HLA
//
// This program demonstrates how to generate
// unique symbols in a program.

program UniqueSymsDemo;

macro unique:theSym;

    @string:theSym

endmacro;

begin UniqueSymsDemo;

    ?lbl:text := unique;

    jmp lbl;

lbl:


    ?@tostring:lbl :text := unique;
    jmp lbl;

lbl:

end UniqueSymsDemo;
```


Program 8.6 A Macro That Generates Unique Symbols for a Program

The first instance of *label:* in this program expands to "_0001_:" while the second instance of *label:* in this program expands to "_0003_:". Of course, reusing symbols in this manner is horrible programming style (it's very confusing), but there are some cases you'll encounter when writing advanced macros where you will want to generate a unique symbol for use in your program. The *unique* macro in this program demonstrates exactly how to do this.

8.2.4 Macros as Compile-Time Procedures

Although programmers typically use macros to expand to some sequence of machine instructions, there is absolutely no requirement that a macro body contain any executable instructions. Indeed, many macros contain only compile-time language statements (e.g., #IF, #WHILE, "?" assignments, etc.). By placing only compile-time language statements in the body of a macro, you can effectively write compile-time procedures and functions using macros.

The *unique* macro from the previous section is a good example of a compile-time function that returns a string result. Consider, again, the definition of this macro:

```
#macro unique:theSym;

    @string:theSym

#endmacro;
```

Whenever your code references this macro, HLA replaces the macro invocation with the text "@string:theSym" which, of course, expands to some string like "_021F_". Therefore, you can think of this macro as a compile-time function that returns a string result.

Be careful that you don't take the function analogy too far. Remember, macros always expand to their body text at the point of invocation. Some expansions may not be legal at any arbitrary point in your programs. Fortunately, most compile-time statements are legal anywhere whitespace is legal in your programs. Therefore, macros generally behave as you would expect functions or procedures to behave during the execution of your compile-time programs.

Of course, the only difference between a procedure and a function is that a function returns some explicit value while procedures simply do some activity. There is no special syntax for specifying a compile-time function return value. As the example above indicates, simply specifying the value you wish to return as a statement in the macro body suffices. A compile-time procedure, on the other hand, would not contain any non-compile-time language statements that expand into some sort of data during macro invocation.

8.2.5 Multi-part (Context-Free) Macros

HLA's macro facilities, as described up to this point, are not particularly amazing. Indeed, most assemblers provide macro facilities very similar to those this chapter presents up to this point. Earlier, this chapter made the claim that HLA's macro facilities are quite a bit more powerful than those found in other assembly languages (or any programming language for that matter). Part of this power comes from the synergy that exists between the HLA compile-time language and HLA's macros. However, the one feature that sets HLA's macro facilities apart from all others is HLA's ability to handle multi-part, or context-free[4], macros. This section describes this powerful feature.

4. The term "context-free" is an automata theory term used to describe constructs, like programming language control structures, that allow nesting.

The best way to introduce HLA's context-free macro facilities is via an example. Suppose you wanted to create a macro to define a new high level language statement in HLA (a very common use for macros). Let's say you wanted to create a statement like the following:

```
nLoop( 10 )

    << body >>

endloop;
```

The basic idea is that this code would execute the body of the loop ten times (or however many times the *nLoop* parameter specifies). A typical low-level implementation of this control structure might take the following form:

```
        mov( 10, ecx );
UniqueLabel:

        << body >>

        dec( ecx );
        jne UniqueLabel;
```

Clearly it will require two macros (*nLoop* and *endloop*) to implement this control structure. The first attempt a beginner might try is doomed to failure:

```
#macro nLoop( cnt );
        mov( cnt, ecx );
UniqueLabel:

#endmacro;

#macro endloop;
    dec( ecx );
    jne UniqueLabel;
#endmacro;
```

You've already seen the problem with this pair of macros: they use a global target label. Any attempt to use the *nLoop* macro more than once will result in a duplicate symbol error. Previously, we utilized HLA's local symbol facilities to overcome this problem. However, that approach will not work here because local symbols are local to a specific macro invocation; unfortunately, the *endloop* macro needs to reference *UniqueLabel* inside the *nLoop* invocation, so *UniqueLabel* cannot be a local symbol in this example.

A quick and dirty solution might be to take advantage of the trick employed by the *unique* macro appearing in previous sections. By utilizing a global text constant, you can share the label information across two macros using an implementation like the following:

```
#macro nLoop( cnt ):UniqueLabel;

    ?nLoop_target:string := @string:UniqueLabel;
    mov( cnt, ecx );
    UniqueLabel:

#endmacro;

#macro endloop;

    dec( ecx );
    jnz @text( nLoop_target );

#endmacro;
```


Using this definition, you can have multiple calls to the *nLoop* and *endloop* macros and HLA will not generate a duplicate symbol error:

```
    nLoop( 10 )

        stdout.put( "Loop counter = ", ecx, nl );

    endloop;

    nLoop( 5 )

        stdout.put( "Second Loop Counter = ", ecx, nl );

    endloop;
```

The macro invocations above produce something like the following (reasonably correct) expansion:

```
    mov( 10, ecx );
_023A_:                 //UniqueLabel, first invocation

        stdout.put( "Loop counter = ", ecx, nl );

        dec( ecx );
        jne _023A_;      // Expansion of nLoop_target becomes _023A_.

    mov( 5, ecx );
_023B_:                 // UniqueLabel, second invocation.

        stdout.put( "Second Loop Counter = ", ecx, nl );

        dec( ecx );
        jnz _023B_;      // Expansion of nLoop_target becomes _023B_.
```

This scheme looks like it's working properly. However, this implementation suffers from a big drawback- it fails if you attempt to nest the *nLoop..endloop* control structure:

```
    nLoop( 10 )

        push( ecx ); // Must preserve outer loop counter.
        nLoop( 5 )

            stdout.put( "ecx=", ecx, " [esp]=", (type dword [esp]), nl );

        endloop;
        pop( ecx );  // Restore outer loop counter.

    endloop;
```

You would expect to see this code print its message 50 times. However, the macro invocations above produce code like the following:

```
    mov( 10, ecx );
_0321_:                 //UniqueLabel, first invocation

            push( ecx );
            mov( 5, ecx );
_0322_:

            stdout.put( "ecx=", ecx, " [esp]=", (type dword [esp]), nl );

        dec( ecx );
        jne _0322_;      // Expansion of nLoop_target becomes _0322_.
```



```
        pop( ecx );
        dec( ecx );
        jne _0322_;      // Expansion of nLoop_target incorrectly becomes _0322_.
```

Note that the last JNE should jump to label "_0321_" rather than "_0322_". Unfortunately, the nested invocation of the *nLoop* macro overwrites the value of the global string constant *nLoop_target* thus the last JNE transfers control to the wrong label.

It is possible to correct this problem using an array of strings and another compile-time constant to create a *stack* of labels. By pushing and popping these labels as you encounter *nLoop* and *endloop* you can emit the correct code. However, this is a lot of work, is very inelegant, and you must repeat this process for every nestable control structure you dream up. In other words, it's a total kludge. Fortunately, HLA provides a better solution: multi-part macros.

Multi-part macros let you define a set of macros that work together. The *nLoop* and the *endloop* macros in this section are a good example of a pair of macros that work intimately together. By defining *nLoop* and *endloop* within a multi-part macro definition, the problems with communicating the target label between the two macros goes away because multi-part macros share parameters and local symbols. This provides a much more elegant solution to this problem than using global constants to hold target label information.

As its name suggests, a multi-part macro consists of a sequence of statements containing two matched macro names (e.g., *nLoop* and *endloop*). Multi-part macro invocations always consist of at least two macro invocations: a *beginning* invocation (e.g., *nLoop*) and a *terminating* invocation (e.g., *endloop*). Some number of unrelated (to the macro bodies) instructions may appear between the two invocations. To declare a multi-part macro, you use the following syntax:

```
#macro beginningMacro (optional_parameters) : optional_local_symbols;

    << beginningMacro body >>

#terminator terminatingMacro (optional_parameters) : optional_local_symbols;

    << terminatingMacro body >>

#endmacro;
```

The presence of the #TERMINATOR section in the macro declaration tells HLA that this is a multi-part macro declaration. It also ends the macro declaration of the beginning macro and begins the declaration of the terminating macro (i.e., the invocation of *beginningMacro* does not emit the code associated with the #TERMINATOR macro). As you would expect, parameters and local symbols are optional in both declarations and the associated glue characters (parentheses and colons) are not present if the parameters and local symbol lists are not present.

Now let's look at the multi-part macro declaration for the *nLoop..endloop* macro pair:

```
#macro nLoop( cnt ):TopOfLoop;

        mov( cnt, ecx );
    TopOfLoop:

#terminator endloop;

        dec( ecx );
        jne TopOfLoop;

#endmacro;
```

As you can see in this example, the definition of the *nLoop..endloop* control structure is much simpler when using multi-part macros; better still, multi-part macro declarations work even if you nest the invocations.

The most notable thing in this particular macro declaration is that the *endloop* macro has access to *nLoop*'s parameters and local symbols (in this example the *endloop* macro does not reference *cnt*, but it could if this was necessary). This makes communication between the two macros trivial.

Multi-part macro invocations must always occur in pairs. If the beginning macro appears in the text, the terminating macro must follow at some point. A terminating macro may never appear in the source file without a previous, matching, instance of the beginning macro. These semantics are identical to many of the HLA high level control structures; i.e., you cannot have an ENDIF without having a corresponding IF clause earlier in the source file.

When you nest multi-part macro invocations, HLA "magically" keeps track of local symbols and always emits the appropriate local label value. The nested macros appearing earlier are no problem for multi-part macros:

```
nLoop( 10 )

    push( ecx ); // Must preserve outer loop counter.
    nLoop( 5 )

        stdout.put( "ecx=", ecx, " [esp]=", (type dword [esp]), nl );

    endloop;
    pop( ecx );  // Restore outer loop counter.

endloop;
```

The above code properly compiles to something like:

```
       mov( 10, ecx );
_01FE_:

           push( ecx );
           mov( 5, ecx );
_01FF_:

           stdout.put( "ecx=", ecx, " [esp]=", (type dword [esp]), nl );

           dec( ecx );
           jne _01FF_;

           pop( ecx );

       dec( ecx );
       jne _01FE_;     // Note the correct label here.
```

In addition to terminating macros, HLA's multi-part macro facilities also provide an option for introducing additional macro declarations associated with the beginning/terminating macro pair: #KEYWORD macros. #KEYWORD macros are macros that are active only between a specific beginning and terminating macro pair. The classic use for #KEYWORD macros is to allow the introduction of context-sensitive keywords into the macro (context-sensitive, in this case, meaning that the terms are only active within the context of the body of statements between the beginning and terminating macros). Classic examples of statements that could employ these types of macros include the BREAK and CONTINUE statements within a loop body and the CASE clause within a SWITCH..ENDSWITCH statement.

The syntax for a multi-part macro declaration that includes one or more #KEYWORD macros is the following:

```
#macro beginningMacro( optional_parameters ): optional_local_labels;

    << beginningMacro Body >>

#keyword keywordMacro( optional_parameters ): optional_local_labels;
```



```
    << keywordMacro Body >>

#terminator terminatingMacro( optional_parameters ): optional_local_labels;

    << terminatingMacro Body >>

#endmacro;
```

If a #KEYWORD macro is present in a macro declaration there must also be a terminating macro declaration. You cannot have a #KEYWORD macro without a corresponding #TERMINATOR macro. The #TERMINATOR macro declaration is always last in a multi-part macro declaration.

The syntax example above specifies only a single #KEYWORD macro. HLA, however, allows zero or more #KEYWORD macro declarations in a multi-part macro. The HLA SWITCH statement, for example, defines two #KEYWORD macros, *case* and *default*.

#KEYWORD and #TERMINATOR macros may refer to the parameters and local symbols defined in the beginning macro, but they may not refer to locals and parameters in other #KEYWORD macros. Parameters and local symbols in #KEYWORD macro declarations are local to that specific macro. If you really need to communicate information between #KEYWORD and #TERMINATOR macros, define some local symbols in the beginning macro and assign these local symbols the parameter (or local symbol) values in the affected #KEYWORD macro. Then refer to this beginning macro local symbol in other parts of the macro. The following is a trivial example of this:

```
#macro ShareParameter:parmValue;

    << beginning macro body >>

#keyword ParmToShare( p );

    ?parmValue:text := @string:p;

    << keyword macro body >>

#terminator UsesSharedParm;

    mov( parmValue, ecx );

    << terminator macro body >>

#endmacro;
```

By assigning *ParmToShare*'s parameter value to the beginning macro's *parmValue* local symbol, this code makes the value of *p* accessible by the *UsesSharedParm* terminating macro.

This section only touches on the capabilities of HLA's multi-part macro facilities. Additional examples appear later in this chapter in the section on Domain Specific Embedded Languages (see "Domain Specific Embedded Languages" on page 1003). This text will make use of HLA's multi-part macros in later chapters as well. For more information on multi-part macros, see these sections in this text or check out the HLA documentation.

8.2.6 Simulating Function Overloading with Macros

The C++ language supports a nifty feature known as *function overloading*. Function overloading lets you write several different functions or procedures that all have the same name. The difference between these functions is the types of their parameters or the number of parameters. A procedure declaration is said to be unique if it has a different number of parameters than other functions with the same name or if the types of its parameters differs from another function with the same name. HLA does not directly support

procedure overloading but you can use macros to achieve the same result. This section explains how to use HLA's macros and the compile-time language to achieve function/procedure overloading.

One good use for procedure overloading is to reduce the number of standard library routines you must remember how to use. For example, the HLA Standard Library provides four different "puti" routines that output an integer value: *stdout.puti64*, *stdout.puti32*, *stdout.puti16*, and *stdout.puti8*. The different routines, as their name suggests, output integer values according to the size of their integer parameter. In the C++ language (or another other language supporting procedure/function overloading) the engineer designing the input routines would probably have chosen to name them all *stdout.puti* and leave it up to the compiler to select the appropriate one based on the operand size[5]. The following macro demonstrates how to do this in HLA using the compile-time language to figure out the size of the parameter operand:

```
// Puti.hla
//
// This program demonstrates procedure overloading via macros.
//
// It defines a "puti" macro that calls stdout.puti8, stdout.puti16,
// stdout.puti32, or stdout.puti64 depending on the size of the operand.


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


// puti-
//
// Automatically decides whether we have a 64, 32, 16, or 8-bit
// operand and calls the appropriate stdout.putiX routine to
// output this value.

macro puti( operand );

    // If we have an eight-byte operand, call puti64:

    #if( @size( operand ) = 8 )

        stdout.puti64( operand );


    // If we have a four-byte operand, call puti32:

    #elseif( @size( operand ) = 4 )

        stdout.puti32( operand );


    // If we have a two-byte operand, call puti16:

    #elseif( @size( operand ) = 2 )

        stdout.puti16( operand );


    // If we have a one-byte operand, call puti8:

    #elseif( @size( operand ) = 1 )
```

5. By the way, the HLA Standard Library does this as well. Although it doesn't provide *stdout.puti*, it does provide *stdout.put* that will choose an appropriate output routine based upon the parameter's type. This is a bit more flexible than a *puti* routine.


```
            stdout.puti8( operand );


        // If it's not an eight, four, two, or one-byte operand,
        // then print an error message:

        #else

            #error( "Expected a 64, 32, 16, or 8-bit operand" )

        #endif

    endmacro;


    // Some sample variable declarations so we can test the macro above.

    static
        i8:     int8    := -8;
        i16:    int16   := -16;
        i32:    int32   := -32;
        i64:    qword;


    begin putiDemo;

        // Initialize i64 since we can't do this in the static section.

        mov( -64, (type dword i64 ));
        mov( $FFFF_FFFF, (type dword i64[4]));

        // Demo the puti macro:

        puti( i8  );  stdout.newln();
        puti( i16 );  stdout.newln();
        puti( i32 );  stdout.newln();
        puti( i64 );  stdout.newln();

    end putiDemo;
```

Program 8.7 Simple Procedure Overloading Based on Operand Size

The example above simply tests the size of the operand to determine which output routine to use. You can use other HLA compile-time functions, like @TYPENAME, to do more sophisticated processing. Consider the following program that demonstrates a macro that overloads stdout.puti32, stdout.putu32, and stdout.putd depending on the type of the operand:

```
// put32.hla
//
// This program demonstrates procedure overloading via macros.
//
// It defines a "put32" macro that calls stdout.puti32, stdout.putu32,
// or stdout.putdw depending on the type of the operand.
```



```
program put32Demo;
#include( "stdlib.hhf" )


// put32-
//
// Automatically decides whether we have an int32, uns32, or dword
// operand and calls the appropriate stdout.putX routine to
// output this value.

macro put32( operand );

    // If we have an int32 operand, call puti32:

    #if( @typename( operand ) = "int32" )

        stdout.puti32( operand );


    // If we have an uns32 operand, call putu32:

    #elseif( @typename( operand ) = "uns32" )

        stdout.putu32( operand );


    // If we have a dword operand, call putidw:

    #elseif( @typename( operand ) = "dword" )

        stdout.putd( operand );


    // If it's not a 32-bit integer value, report an error:

    #else

        #error( "Expected an int32, uns32, or dword operand" )

    #endif

endmacro;


// Some sample variable declarations so we can test the macro above.

static
    i32:    int32   := -32;
    u32:    uns32   := 32;
    d32:    dword   := $32;


begin put32Demo;


    // Demo the put32 macro:

    put32( d32 );  stdout.newln();
    put32( u32 );  stdout.newln();
    put32( i32 );  stdout.newln();
```



```
end put32Demo;
```

Program 8.8 Procedure Overloading Based on Operand Type

You can easily extend the macro above to output eight and sixteen-bit operands as well as 32-bit operands. That is left as an exercise.

The number of actual parameters is another way to resolve which overloaded procedure to call. If you specify a variable number of macro parameters (using the "[ ]" syntax, see “Macros with a Variable Number of Parameters” on page 974) you can use the @ELEMENTS compile-time function to determine exactly how many parameters are present and call the appropriate routine. The following sample program uses this trick to determine whether it should call *stdout.puti32* or *stdout.puti32Size*:

```
// puti32.hla
//
// This program demonstrates procedure overloading via macros.
//
// It defines a "puti32" macro that calls stdout.puti32 or stdout.puti32size
// depending on the number of parameters present.


program puti32Demo;
#include( "stdlib.hhf" )


// puti32-
//
// Automatically decides whether we have an int32, uns32, or dword
// operand and calls the appropriate stdout.putX routine to
// output this value.

macro puti32( operand[] );

    // If we have a single operand, call stdout.puti32:

    #if( @elements( operand ) = 1 )

        stdout.puti32( @text(operand[0]) );


    // If we have two operands, call stdout.puti32size and
    // supply a default value of ' ' for the padding character:

    #elseif( @elements( operand ) = 2 )

        stdout.puti32Size( @text(operand[0]), @text(operand[1]), ' ' );


    // If we have three parameters, then pass all three of them
    // along to puti32size:

    #elseif( @elements( operand ) = 3 )

        stdout.puti32Size
        (
            @text(operand[0]),
```



```
            @text(operand[1]),
            @text(operand[2])
        );


    // If we don't have one, two, or three operands, report an error:

    #else

        #error( "Expected one, two, or three operands" )

    #endif

endmacro;


// A sample variable declaration so we can test the macro above.

static
    i32:    int32   := -32;


begin puti32Demo;


    // Demo the put32 macro:

    puti32( i32 );  stdout.newln();
    puti32( i32, 5 );  stdout.newln();
    puti32( i32, 5, '*' );  stdout.newln();

end puti32Demo;
```

Program 8.9 Using the Number of Parameters to Resolve Overloaded Procedures

All the examples up to this point provide procedure overloading for Standard Library routines (specifically, the integer output routines). Of course, you are not limited to overloading procedures in the HLA Standard Library. You can create your own overloaded procedures as well. All you've got to do is write a set of procedures, all with unique names, and then use a single macro to decide which routine to actually call based on the macro's parameters. Rather than call the individual routines, invoke the common macro and let it decide which procedure to actually call.

8.3 Writing Compile-Time "Programs"

The HLA compile-time language provides a powerful facility with which to write "programs" that execute while HLA is compiling your assembly language programs. Although it is possible to write some general purpose programs using the HLA compile-time language, the real purpose of the HLA compile-time language is to allow you to write short programs *that write other programs*. In particular, the primary purpose of the HLA compile-time language is to automate the creation of large or complex assembly language sequences. The following subsections provide some simple examples of such compile-time programs.

8.3.1 Constructing Data Tables at Compile Time

Earlier, this text suggested that you could write programs to generate large, complex, lookup tables for your assembly language programs (see “Generating Tables” on page 651). That chapter provided examples in HLA but suggested that writing a separate program was unnecessary. This is true, you can generate most look-up tables you’ll need using nothing more than the HLA compile-time language facilities. Indeed, filling in table entries is one of the principle uses of the HLA compile-time language. In this section we will take a look at using the HLA compile-time language to construct data tables during compilation.

In the section on generating tables, this text gave an example of an HLA program that writes a text file containing a lookup table for the trigonometric *sine* function. The table contains 360 entries with the index into the table specifying an angle in degrees. Each *int32* entry in the table contained the value *sin(angle)*1000* where *angle* is equal to the index into the table. The section on generating tables suggested running this program and then including the text output from that program into the actual program that used the resulting table. You can avoid much of this work by using the compile-time language. The following HLA program includes a short compile-time code fragment that constructs this table of sines directly.

```
// demoSines.hla
//
// This program demonstrates how to create a lookup table
// of sine values using the HLA compile-time language.

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

const
    pi :real80 := 3.1415926535897;

readonly
    sines:  int32[ 360 ] :=
            [
                // The following compile-time program generates
                // 359 entries (out of 360).  For each entry
                // it computes the sine of the index into the
                // table and multiplies this result by 1000
                // in order to get a reasonable integer value.

                ?angle := 0;
                #while( angle < 359 )

                    // Note: HLA's @sin function expects angles
                    // in radians.  radians = degrees*pi/180.
                    // the "int32" function truncates its result,
                    // so this function adds 1/2 as a weak attempt
                    // to round the value up.

                    int32( @sin( angle * pi / 180.0 ) * 1000 + 0.5 ),
                    ?angle := angle + 1;

                #endwhile

                // Here's the 360th entry in the table.  This code
                // handles the last entry specially because a comma
                // does not follow this entry in the table.

                int32( @sin( 359 * pi / 180.0 ) * 1000 + 0.5 )
            ];
```



```
begin demoSines;

    // Simple demo program that displays all the values in the table.

    for( mov( 0, ebx); ebx<360; inc( ebx )) do

        mov( sines[ ebx*4 ], eax );
        stdout.put
        (
            "sin( ",
            (type uns32 ebx ),
            " )*1000 = ",
            (type int32 eax ),
            nl
        );

    endfor;


end demoSines;
```

Program 8.10 Generating a SINE Lookup Table with the Compile-time Language

Another common use for the compile-time language is to build ASCII character lookup tables for use by the XLAT instruction at run-time. Common examples include lookup tables for alphabetic case manipulation. The following program demonstrates how to construct an upper case conversion table and a lower case conversion table[6]. Note the use of a macro as a compile-time procedure to reduce the complexity of the table generating code:

```
// demoCase.hla
//
// This program demonstrates how to create a lookup table
// of alphabetic case conversion values using the HLA
// compile-time language.

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

const
    pi :real80 := 3.1415926535897;


// emitCharRange-
//
//  This macro emits a set of character entries
// for an array of characters.  It emits a list
// of values (with a comma suffix on each value)
// from the starting value up to, but not including,
// the ending value.
```

6. Note that on modern processors, using a lookup table is probably not the most efficient way to convert between alphabetic cases. However, this is just an example of filling in the table using the compile-time language. The principles are correct even if the code is not exactly the best it could be.


```
        macro emitCharRange( start, last ): index;

            ?index:uns8 := start;
            #while( index < last )

                char( index ),
                ?index := index + 1;

            #endwhile

        endmacro;

        readonly

            // toUC:
            // The entries in this table contain the value of the index
            // into the table except for indicies #$61..#$7A (those entries
            // whose indicies are the ASCII codes for the lower case
            // characters).  Those particular table entries contain the
            // codes for the corresponding upper case alphabetic characters.
            // If you use an ASCII character as an index into this table and
            // fetch the specified byte at that location, you will effectively
            // translate lower case characters to upper case characters and
            // leave all other characters unaffected.

            toUC:   char[ 256 ] :=
                    [
                        // The following compile-time program generates
                        // 255 entries (out of 256).  For each entry
                        // it computes toupper( index ) where index is
                        // the character whose ASCII code is an index
                        // into the table.

                        emitCharRange( 0, uns8('a') )

                        // Okay, we've generated all the entries up to
                        // the start of the lower case characters.  Output
                        // Upper Case characters in place of the lower
                        // case characters here.

                        emitCharRange( uns8('A'), uns8('Z') + 1 )


                        // Okay, emit the non-alphabetic characters
                        // through to byte code #$FE:

                        emitCharRange( uns8('z') + 1, $FF )

                        // Here's the last entry in the table.  This code
                        // handles the last entry specially because a comma
                        // does not follow this entry in the table.

                        #$FF
                    ];


            // The following table is very similar to the one above.
            // You would use this one, however, to translate upper case
            // characters to lower case while leaving everything else alone.
            // See the comments in the previous table for more details.
```



```
    TOlc:   char[ 256 ] :=
            [
                emitCharRange( 0, uns8('A') )
                emitCharRange( uns8('a'), uns8('z') + 1 )
                emitCharRange( uns8('Z') + 1, $FF )

                #$FF
            ];


begin demoCase;

    for( mov( uns32( ' ' ), eax ); eax <= $FF; inc( eax )) do

        mov( toUC[ eax ], bl );
        mov( TOlc[ eax ], bh );
        stdout.put
        (
            "toupper( '",
            (type char al),
            "' ) = '",
            (type char bl),
            "'        tolower( '",
            (type char al),
            "' ) = '",
            (type char bh),
            "'",
            nl
        );

    endfor;


end demoCase;
```

Program 8.11 Generating Case Conversion Tables with the Compile-Time Language

One important thing to note about this sample is the fact that a semicolon does not follow the *emitCharRange* macro invocations. Macro invocations do not require a closing semicolon. Often, it is legal to go ahead and add one to the end of the macro invocation because HLA is normally very forgiving about having extra semicolons inserted into the code. In this case, however, the extra semicolons are illegal because they would appear between adjacent entries in the *TOlc* and *toUC* tables. Keep in mind that macro invocations don't require a semicolon, especially when using macro invocations as compile-time procedures.

8.3.2 Unrolling Loops

In the chapter on Low-Level Control Structures (see "Unraveling Loops" on page 800) this text points out that you can unravel loops to improve the performance of certain assembly language programs. One problem with unravelling, or unrolling, loops is that you may need to do a lot of extra typing, especially if many iterations are necessary. Fortunately, HLA's compile-time language facilities, especially the #WHILE loop, comes to the rescue. With a small amount of extra typing plus one copy of the loop body, you can unroll a loop as many times as you please.

If you simply want to repeat the same exact code sequence some number of times, unrolling the code is especially trivial. All you've got to do is wrap an HLA #WHILE..#ENDWHILE loop around the sequence

and count down a VAL object the specified number of times. For example, if you wanted to print "Hello World" ten times, you could encode this as follows:

```
?count := 0;
#while( count < 10 )

    stdout.put( "Hello World", nl );
    ?count := count + 1;

#endwhile
```

Although the code above looks very similar to a WHILE (or FOR) loop you could write in your program, remember the fundamental difference: the code above simply consists of ten straight *stdout.put* calls in the program. Were you to encode this using a FOR loop, there would be only one call to *stdout.put* and lots of additional logic to loop back and execute that single call ten times.

Unrolling loops becomes slightly more complicated if any instructions in that loop refer to the value of a loop control variable or other value that changes with each iteration of the loop. A typical example is a loop that zeros the elements of an integer array:

```
mov( 0, eax  );
for( mov( 0, ebx ); ebx < 20; inc( ebx )) do

    mov( eax, array[ ebx*4 ] );

endfor;
```

In this code fragment the loop uses the value of the loop control variable (in EBX) to index into *array*. Simply copying "mov(eax, array[ebx*4]);" twenty times is not the proper way to unroll this loop. You must substitute an appropriate constant index in the range 0..76 (the corresponding loop indices, times four) in place of "EBX*4" in this example. Correctly unrolling this loop should produce the following code sequence:

```
    mov( eax, array[ 0*4 ] );
    mov( eax, array[ 1*4 ] );
    mov( eax, array[ 2*4 ] );
    mov( eax, array[ 3*4 ] );
    mov( eax, array[ 4*4 ] );
    mov( eax, array[ 5*4 ] );
    mov( eax, array[ 6*4 ] );
    mov( eax, array[ 7*4 ] );
    mov( eax, array[ 8*4 ] );
    mov( eax, array[ 9*4 ] );
    mov( eax, array[ 10*4 ] );
    mov( eax, array[ 11*4 ] );
    mov( eax, array[ 12*4 ] );
    mov( eax, array[ 13*4 ] );
    mov( eax, array[ 14*4 ] );
    mov( eax, array[ 15*4 ] );
    mov( eax, array[ 16*4 ] );
    mov( eax, array[ 17*4 ] );
    mov( eax, array[ 18*4 ] );
    mov( eax, array[ 19*4 ] );
```

You can do this more efficiently using the following compile-time code sequence:

```
?iteration := 0;
#while( iteration < 20 )

    mov( eax, array[ iteration*4 ] );
    ?iteration := iteration+1;
```



```
#endwhile
```

If the statements in a loop make use of the loop control variable's value, it is only possible to unroll such loops if those values are known at compile time. You cannot unroll loops when user input (or other run-time information) controls the number of iterations.

8.4 Using Macros in Different Source Files

Unlike procedures, macros do not have a fixed piece of code at some address in memory. Therefore, you cannot create "external" macros and link them with other modules in your program. However, it is very easy to share macros with different source files – just put the macros you wish to reuse in a header file and include that file using the #include directive. You can make the macro will be available to any source file you choose using this simple trick.

8.5 Putting It All Together

This chapter has barely touched on the capabilities of the HLA macro processor and compile-time language. The HLA language has one of the most powerful macro processors around. None of the other 80x86 assemblers even come close to HLA's capabilities with regard to macros. Indeed, if you could say just one thing about HLA in relation to other assemblers, it would have to be that HLA's macro facilities are, by far, the best.

The combination of the HLA compile-time language and the macro processor give HLA users the ability to extend the HLA language in many ways. In the chapter on Domain Specific Languages, you'll get the opportunity to see how to create your own specialized languages using HLA's macro facilities.

Even if you don't do exotic things like creating your own languages, HLA's macro facilities and compile-time language are really great for automating code generation in your programs. The HLA Standard Library, for example, makes heavy use of HLA's macro facilities; "procedures" like *stdout.put* and *stdin.get* would be very difficult (if not impossible) to create without the power of HLA macro facilities and the compile-time language. For some good examples of the possible complexity one can achieve with HLA's macros, you should scan through the #include files in the HLA Standard Library and look at some of the macros appearing therein.

This chapter serves as a basic introduction to HLA's macro facilities. As you use macros in your own programs you will gain even more insight into their power. So by all means, use macros as much as you can – they can help reduce the effort needed to develop programs.

