
The HLA Compile-Time Language

Chapter Seven

7.1 Chapter Overview

Now we come to the fun part. For the past nine chapters this text has been molding and conforming you to deal with the HLA language and assembly language programming in general. In this chapter you get to turn the tables; you'll learn how to force HLA to conform to your desires. This chapter will teach you how to extend the HLA language using HLA's *compile-time language*. By the time you are through with this chapter, you should have a healthy appreciation for the power of the HLA compile-time language. You will be able to write short compile-time programs. You will also be able to add new statements, of your own choosing, to the HLA language.

7.2 Introduction to the Compile-Time Language (CTL)

HLA is actually two languages rolled into a single program. The *run-time language* is the standard 80x86/HLA assembly language you've been reading about in all the past chapters. This is called the run-time language because the programs you write execute when you run the executable file. HLA contains an interpreter for a second language, the HLA Compile-Time Language (or CTL) that executes programs while HLA is compiling a program. The source code for the CTL program is embedded in an HLA assembly language source file; that is, HLA source files contain instructions for both the HLA CTL and the run-time program. HLA executes the CTL program during compilation. Once HLA completes compilation, the CTL program terminates; the CTL application is not a part of the run-time executable that HLA emits, although the CTL application can *write* part of the run-time program for you and, in fact, this is the major purpose of the CTL.

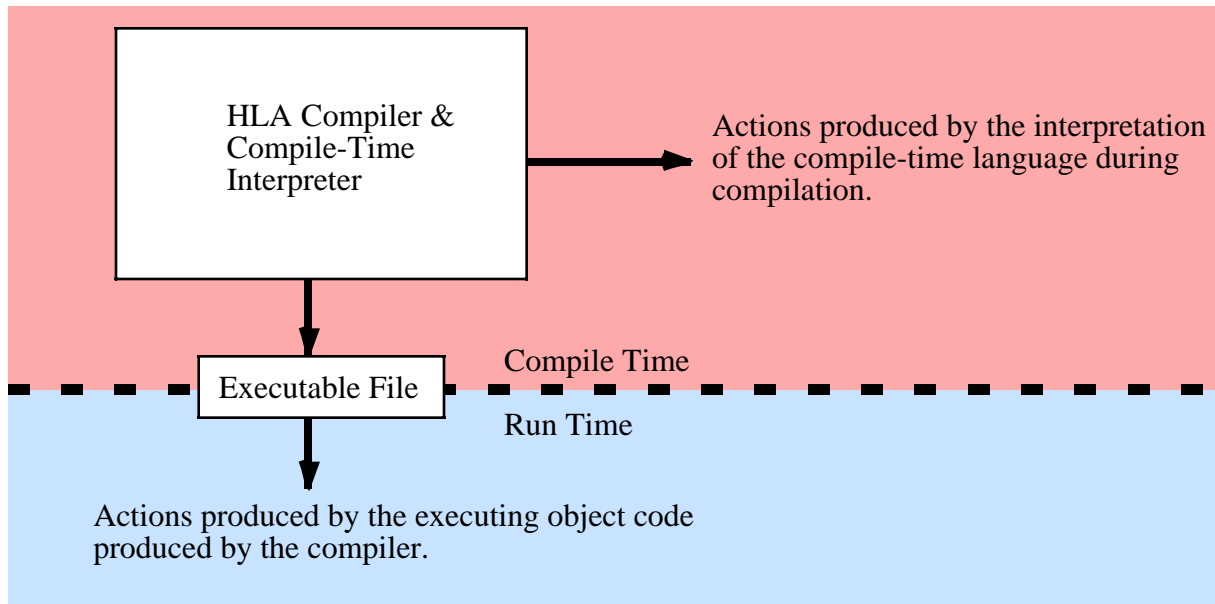


Figure 7.1 Compile-Time vs. Run-Time Execution

It may seem confusing to have two separate languages built into the same compiler. Perhaps you're even questioning why anyone would need a compile time language. To understand the benefits of a compile time language, consider the following statement that you should be very comfortable with at this point:

```
stdout.put( "i32=", i32, " strVar=", strVar, " charVar=", charVar, nl );
```

This statement is neither a statement in the HLA language nor a call to some HLA Standard Library procedure. Instead, *stdout.put* is actually a statement in a CTL application provided by the HLA Standard Library. The *stdout.put* "application" processes a list of objects (the parameter list) and makes calls to various other Standard Library procedures; it chooses the procedure to call based on the type of the object it is currently processing. For example, the *stdout.put* "application" above will emit the following statements to the run-time executable:

```
stdout.puts( "i32=" );
stdout.puti32( i32 );
stdout.puts( " strVar=" );
stdout.puts( strVar );
stdout.puts( " charVar=" );
stdout.putc( charVar );
stdout.newln();
```

Clearly the *stdout.put* statement is much easier to read and write than the sequence of statements that *stdout.put* emits in response to its parameter list. This is one of the more powerful capabilities of the HLA programming language: the ability to modify the language to simplify common programming tasks. Printing lots of different data objects in a sequential fashion is a common task; the *stdout.put* "application" greatly simplifies this process.

The HLA Standard Library is *loaded* with lots of HLA CTL examples. In addition to standard library usage, the HLA CTL is quite adept at handling "one-off" or "one-use" applications. A classic example is filling in the data for a lookup table. An earlier chapter in this text noted that it is possible to construct look-up tables using the HLA CTL (see "Tables" on page 647 and "Generating Tables" on page 651). Not only is this possible, but it is often far less work to use the HLA CTL to construct these look-up tables. This chapter abounds with examples of exactly this application of the CTL.

Although the CTL itself is relatively inefficient and you would not use it to write end-user applications, it does maximize the use of that one precious commodity of which there is so little available: your time. By learning how to use the HLA CTL, and applying it properly, you can develop assembly language applications as rapidly as high level language applications (even faster since HLA's CTL lets you create *very* high level language constructs).

7.3 The #PRINT and #ERROR Statements

Chapter One of this textbook began with the typical first program most people write when learning a new language; the "Hello World" program. It is only fitting for this chapter to present that same program when discussing the second language of this text. So here it is, the basic "Hello World" program written in the HLA compile time language:

```

program ctlHelloWorld;
begin ctlHelloWorld;

    #print( "Hello, World of HLA/CTL" )

end ctlHelloWorld;

```

Program 7.1 The CTL "Hello World" Program

The only CTL statement in this program is the "#print" statement. The remaining lines are needed just to keep the compiler happy (though we could have reduced the overhead to two lines by using a UNIT rather than a PROGRAM declaration).

The #PRINT statement displays the textual representation of its argument list during the compilation of an HLA program. Therefore, if you compile the program above with the command "hla ctlHW.hla" the HLA compiler will immediately print, before returning control to the command line, the text:

```

Hello, World of HLA/CTL

```

Note that there is a big difference between the following two statements in an HLA source file:

```

#print( "Hello World" )
stdout.puts( "Hello World" nl );

```

The first statement prints "Hello World" (and a newline) during the compilation process. This first statement does not have any effect on the executable program. The second line doesn't affect the compilation process (other than the emission of code to the executable file). However, when you run the executable file, the second statement prints the string "Hello World" followed by a new line sequence.

The HLA/CTL #PRINT statement uses the following basic syntax:

```

#print( list_of_comma_separated_constants )

```

Note that a semicolon does not terminate this statement. Semicolons terminate run-time statements, they generally do not terminate compile-time statements (there is one big exception, as you will see a little later).

The #PRINT statement must have at least one operand; if multiple operands appear in the parameter list, you must separate each operand with a comma (just like *stdout.put*). If a particular operand is not a string constant, HLA will translate that constant to its corresponding string representation and print that string. Example:

```

#print( "A string Constant ", 45, ' ', 54.9, ' ', true )

```

You may specify named symbolic constants and constant expressions. However, all #PRINT operands must be constants (either literal constants or constants you define in the CONST or VAL sections) and those constants must be defined before you use them in the #PRINT statement. Example:

```
const
  pi := 3.14159;
  charConst := 'c';

#print( "PI = ", pi, " CharVal=", CharConst )
```

The HLA #PRINT statement is particularly invaluable for debugging CTL programs (since there is no debugger available for CTL code). This statement is also useful for displaying the progress of the compilation and displaying assumptions and default actions that take place during compilation. Other than displaying the text associated with the #PRINT parameter list, the #PRINT statement does not have any affect on the compilation of the program.

The #ERROR statement allows a single string constant operand. Like #PRINT this statement will display the string to the console during compilation. However, the #ERROR statement treats the string as an error message and displays the string as part of an HLA error diagnostic. Further, the #ERROR statement increments the error count and this will cause HLA to stop the compilation (without assembly or linking) at the conclusion of the source file. You would normally use the #ERROR statement to display an error message during compilation if your CTL code discovers something that prevents it from creating valid code. Example:

```
#error( "Statement must have exactly one operand" )
```

Like the #PRINT statement, the #ERROR statement does not end with a semicolon. Although #ERROR only allows a string operand, it's very easy to print other values by using the string (constant) concatenation operator and several of the HLA built-in compile-time functions (see "Compile-Time Constants and Variables" on page 952 and "Compile-Time Functions" on page 956) for more details).

7.4 Compile-Time Constants and Variables

Just as the run-time language supports constants and variables, so does the compile-time language. You declare compile-time constants in the CONST section, the same as for the run-time language. You declare compile-time variables in the VAL section. Objects you declare in the VAL section are constants as far as the run-time language is concerned, but remember that you can change the value of an object you declare in the VAL section throughout the source file. Hence the term "compile-time variable." See "HLA Constant and Value Declarations" on page 397 for more details.

The CTL assignment statement ("?") computes the value of the constant expression to the right of the assignment operator (":=") and stores the result into the VAL object name appearing immediately to the left of the assignment operator¹. The following example is a rework of the example above; this example, however, may appear anywhere in your HLA source file, not just in the VAL section of the program.

```
?ConstToPrint := 25;
#print( "ConstToPrint = ", ConstToPrint )
?ConstToPrint := ConstToPrint + 5;
#print( "Now ConstToPrint = ", ConstToPrint )
```

Note that HLA's CTL ignores the distinction between the different sizes of numeric objects. HLA always reserves storage for the largest possible object of a given type, so HLA merges the following types:

```
byte, word, dword -> dword
uns8, uns16, uns32 -> uns32
int8, int16, int32 -> int32
```

1. If the identifier to the left of the assignment operator is undefined, HLA will automatically declare this object at the current scope level.

```
real32, real64, real80 -> real80
```

For most practical applications of the CTL, this shouldn't make a difference in the operation of the program.

7.5 Compile-Time Expressions and Operators

As the previous section states, the HLA CTL supports constant expressions in the CTL assignment statement. Unlike the run-time language (where you have to translate algebraic notation into a sequence of machine instructions), the HLA CTL allows a full set of arithmetic operations using familiar expression syntax. This gives the HLA CTL considerable power, especially when combined with the built-in compile-time functions the next section discusses.

HLA's CTL supports the following operators in compile-time expressions:

Table 1: Compile-Time Operators

Operator(s)	Operand Types ^a	Description
- (unary)	numeric	Negates the specific numeric value (int, uns, real).
	cset	Returns the complement of the specified character set.
! (unary)	integer	Inverts all the bits in the operand (bitwise not).
	boolean	Boolean NOT of the operand.
*	numericL * numericR	Multiplies the two operands.
	csetL * csetR	Computes the intersection of the two sets
div	integerL div integerR	Computes the integer quotient of the two integer (int/uns/dword) operands.
mod	integerL mod integerR	Computes the remainder of the division of the two integer (int/uns/dword) operands.
/	numericL / numericR	Computes the real quotient of the two numeric operands. Returns a real result even if both operands are integers.
<<	integerL << integerR	Shifts integerL operand to the left the number of bits specified by the integerR operand.
>>	integerL >> integerR	Shifts integerL operand to the right the number of bits specified by the integerR operand.
+	numericL + numericR	Adds the two numeric operands.
	csetL + csetR	Computes the union of the two sets.
	strL + strR	Concatenates the two strings.

Table 1: Compile-Time Operators

Operator(s)	Operand Types ^a	Description
-	numericL - numericR	Computes the difference between numericL and numericR.
	csetL - csetR	Computes the set difference of csetL-csetR.
= or ==	numericL = numericR	Returns true if the two operands have the same value.
	csetL = csetR	Returns true if the two sets are equal.
	strL = strR	Returns true if the two strings/chars are equal.
	typeL = typeR	Returns true if the two values are equal. They must be the same type.
<> or !=	typeL <> typeR (Same as =)	Returns false if the two (compatible) operands are not equal to one another.
<	numericL < numericR	Returns true if numericL is less than numericR.
	csetL < csetR	Returns true if csetL is a proper subset of csetR.
	strL < strR	Returns true if strL is less than strR
	booleanL < booleanR	Returns true if left operand is less than right operand (note: false < true).
	enumL < enumR	Returns true if enumL appears in the same enum list as enumR and enumL appears first.
<=	Same as <	Returns true if the left operand is less than or equal to the right operand. For character sets, this means that the left operand is a subset of the right operand.
>	Same as <	Returns true if the left operand is greater than the right operand. For character sets, this means that the left operand is a proper superset of the right operand.
>=	Same as <=	Returns true if the left operand is greater than or equal to the right operand. For character sets, this means that the left operand is a superset of the right operand.
&	integerL & integerR	Computes the bitwise AND of the two operands.
	booleanL & booleanR	Computes the logical AND of the two operands.

Table 1: Compile-Time Operators

Operator(s)	Operand Types ^a	Description
	integerL integerR	Computes the bitwise OR of the two operands.
	booleanL booleanR	Computes the logical OR of the two operands.
^	integerL ^ integerR	Computes the bitwise XOR of the two operands.
	booleanL ^ booleanR	Computes the logical XOR of the two operands. Note that this is equivalent to "booleanL <> booleanR".
in	charL in csetR	Returns true if charL is a member of csetR.

a. Numeric is {intXX, unsXX, byte, word, dword, and realXX} values. Cset is a character set operand. Type integer is { intXX, unsXX, byte, word, dword }. Type str is any string or character value. "TYPE" indicates an arbitrary HLA type. Other types specify an explicit HLA data type.

Table 2: Operator Precedence and Associativity

Associativity	Precedence (Highest to Lowest)	Operator
Right-to-left	6	! (unary)
		- (unary)
Left to right	5	*
		div
		mod
		/
		>>
		<<
Left to right	4	+
		-
Left to right	3	= or ==
		<> or !=
		<
		<=
		>
		>=

Table 2: Operator Precedence and Associativity

Associativity	Precedence (Highest to Lowest)	Operator
Left to right	2	&
		^
Nonassociative	1	in

Of course, you can always override the default precedence and associativity of an operator by using parentheses in an expression.

7.6 Compile-Time Functions

HLA provides a wide range of compile-time functions you can use. These functions compute values during compilation the same way a high level language function computes values at run-time. The HLA compile-time language includes a wide variety of numeric, string, and symbol table functions that help you write sophisticated compile-time programs.

Most of the names of the built-in compile-time functions begin with the special symbol "@" and have names like `@sin` or `@length`. The use of these special identifiers prevents conflicts with common names you might want to use in your own programs (like `length`). The remaining compile-time functions (those that do not begin with "@") are typically data conversion functions that use type names like `int8` and `real64`. You can even create your own compile-time functions using macros (see "Macros" on page 969).

HLA organizes the compile-time functions into various classes depending on the type of operation. For example, there are functions that convert constants from one form to another (e.g., string to integer conversion), there are many useful string functions, and HLA provides a full set of compile-time numeric functions.

The complete list of HLA compile-time functions is too lengthy to present here. Instead, a complete description of each of the compile-time objects and functions appears in Appendix H (see "HLA Compile-Time Functions" on page 1493); this section will highlight a few of the functions in order to demonstrate their use. Later sections in this chapter, as well as future chapters, will make extensive use of the various compile-time functions.

Perhaps the most important concept to understand about the compile-time functions is that they are equivalent to constants in your assembly language code (i.e., the run-time program). For example, the compile-time function invocation `@sin(3.1415265358979328)` is roughly equivalent to specifying "0.0" at that point in your program². A function invocation like `@sin(x)` is legal only if `x` is a constant with a previous declaration at the point of the function call in the source file. In particular, `x` cannot be a run-time variable or other object whose value exists at run-time rather than compile-time. Since HLA replaces compile-time function calls with their constant result, you may ask why you should even bother with compile time functions. After all, it's probably more convenient to type "0.0" than it is to type `@sin(3.1415265358979328)` in your program. However, compile-time functions are really handy for generating lookup tables (see "Generating Tables" on page 651) and other mathematical results that may

2. Actually, since `@sin`'s parameter in this example is not exactly π , you will get a small positive number instead of zero as the function result, but in theory you should get zero.

change whenever you change a `CONST` value in your program. Later sections in this chapter will explore these ideas farther.

7.6.1 Type Conversion Compile-time Functions

One set of commonly used compile-time functions are the type conversion functions. These functions take a single parameter of one type and convert that information to some specified type. These functions use several of the HLA built-in data type names as the function names. Functions in this category are

- `boolean`
- `int8`, `int16`, and `int32`
- `uns8`, `uns16`, and `uns32`
- `byte`, `word`, `dword` (these are effectively equivalent to `uns8`, `uns16`, and `uns32`)
- `real32`, `real64`, and `real80`
- `char`
- `string`
- `cset`
- `text`

These functions accept a single constant expression parameter and, if at all reasonable, convert that expression's value to the type specified by the type name. For example, the following function call returns the value `-128` since it converts the string constant to the corresponding integer value:

```
int8( "-128" )
```

Certain conversions don't make sense or have restrictions associated with them. For example, the *boolean* function will accept a string parameter, but that string must be "true" or "false" or the function will generate a compile-time error. Likewise, the numeric conversion functions (e.g., *int8*) allow a string operand but the string operand must represent a legal numeric value. Some conversions (e.g., *int8* with a character set parameter) simply don't make sense and are always illegal.

One of the most useful functions in this category is the *string* function. This function accepts nearly all constant expression types and it generates a string that represents the parameter's data. For example, the invocation `"string(128)"` produces the string `"128"` as the return result. This function is real handy when you have a value that you wish to use where HLA requires a string. For example, the `#ERROR` compile-time statement only allows a single string operand. You can use the *string* function and the string concatenation operator (`"+"`) to easily get around this limitation, e.g.,

```
#error( "Value ( " + string( Value ) + " ) is out of range" )
```

7.6.2 Numeric Compile-Time Functions

The functions in this category perform standard mathematical operations at compile time. These functions are real handy for generating lookup tables and "parameterizing" your source code by recalculating functions on constants defined at the beginning of your program. Functions in this category include the following:

- `@abs`
- `@ceil`, `@floor`
- `@sin`, `@cos`, `@tan`
- `@exp`, `@log`, `@log10`
- `@min`, `@max`
- `@random`, `@randomize`
- `@sqrt`

See Appendix H for more details on these functions.

7.6.3 Character Classification Compile-Time Functions

The functions in this group all return a boolean result. They test a character (or all the characters in a string) to see if it belongs to a certain class of characters. The functions in this category include

- @isAlpha, @isAlphanumeric
- @isDigit, @isxDigit
- @isLower, @isUpper
- @isSpace

In addition to these character classification functions, the HLA language provides a set of pattern matching functions that you can also use to classify character and string data. See the appropriate sections a little later for the discussion of these routines.

7.6.4 Compile-Time String Functions

The functions in this category operate on string parameters. Most return a string result although a few (e.g., @length and @index) return integer results. These functions do not directly affect the values of their parameters; instead, they return an appropriate result that you can assign back to the parameter if you wish to do so.

- @delete, @insert
- @index, @rindex
- @length
- @lowercase, @uppercase
- @strbrk, @strspan
- @strset
- @substr, @tokenize, @trim

For specific details concerning these functions and their parameters and their types, see Appendix H. Combined with the pattern matching functions, the string handling functions give you the ability to extend the HLA language by processing textual data that appears in your source files. Later sections appearing in this chapter will discuss ways to do this.

The @length function deserves a special discussion because it is probably the most popular function in this category. It returns an *uns32* constant specifying the number of characters found in its string parameter. The syntax is the following:

```
@length( string_expression )
```

Where *string_expression* represents any compile-time string expression. As noted above, this function returns the length, in characters, of the specified expression.

7.6.5 Compile-Time Pattern Matching Functions

HLA provides a very rich set of string/pattern matching functions that let you test a string to see if it begins with certain types of characters or strings. Along with the string processing functions, the pattern matching functions let you extend the HLA language and provide several other benefits as well. There are far too many pattern matching functions to list here (see Appendix H for complete details). However, a few examples will demonstrate the power and convenience of these routines.

The pattern matching functions all return a boolean true/false result. If a function returns true, we say that the function *succeeds* in matching its operand. If the function returns false, then we say it *fails* to match its operand. An important feature of the pattern matching functions is that they do not have to match the entire string you supply as a parameter, these patterns will (usually) succeed as long as they match a prefix of

the string parameter. The `@matchStr` function is a good example, the following function invocation always returns true:

```
@matchStr( "Hello World", "Hello" )
```

The first parameter of all the pattern matching functions ("Hello World" in this example) is the string to match. The matching functions will attempt to match the characters at the beginning of the string with the other parameters supplied for that particular function. In the `@matchStr` example above, the function succeeds if the first parameter begins with the string specified as the second parameter (which it does). The fact that the "Hello World" string contains additional characters beyond "Hello" is irrelevant; it only needs to begin with the string "Hello" is doesn't require equality with "Hello".

Most of the compile-time pattern matching functions support two optional parameters. The functions store additional data into the VAL objects specified by these two parameters if the function is successful (conversely, if the function fails, it does not modify these objects). The first parameter is where the function stores the *remainder*. The remainder after the execution of a pattern matching function is those characters that follow the matched characters in the string. In the example above, the remainder would be " World". If you wanted to capture this remainder data, you would add a third parameter to the `@matchStr` function invocation:

```
@matchStr( "Hello World", "Hello", World )
```

This function invocation would leave " World" sitting in the *World* VAL object. Note that *World* must be pre-declared as a string in the VAL section (or via the "?" statement) prior to the invocation of this function.

By using the conjunction operator ("&") you can combine several pattern matching functions into a single expression, e.g.,

```
@matchStr( "Hello There World", "Hello ", tw ) & @matchStr( tw, "There ", World )
```

This full expression returns true and leaves "World" sitting in the *World* variable. It also leaves "There World" sitting in *tw*, although *tw* is probably a temporary object whose value has no meaning beyond this expression. Of course, the above could be more efficiently implemented as follows:

```
@matchStr( "Hello There World", "Hello There", World )
```

However, keep in mind that you can combine different pattern matching functions using conjunction, they needn't all be calls to `@matchStr`.

The second optional parameter to most pattern matching functions holds a copy of the text that the function matched. E.g., the following call to `@matchStr` returns "Hello" in the Hello VAL object³

```
@matchStr( "Hello World", "Hello", World, Hello )
```

For more information on these pattern matching functions please see Appendix H. The chapter on Domain Specific Languages (see "Domain Specific Embedded Languages" on page 1003) and several other sections in this chapter will also make further use of these functions.

7.6.6 Compile-Time Symbol Information

During compilation HLA maintains an internal database known as the *symbol table*. The symbol table contains lots of useful information concerning all the identifiers you've defined up to a given point in the program. In order to generate machine code output, HLA needs to query this database to determine how to treat certain symbols. In your compile-time programs, it is often necessary to query the symbol table to determine how to handle an identifier or expression in your code. The HLA compile-time symbol information functions handle this task.

3. Strictly speaking, this example is rather contrived since we generally know the string that `@matchStr` matches. However, for other pattern matching functions this is not the case.

Many of the compile-time symbol information functions are well beyond the scope of this chapter (and, in some cases, beyond the scope of this text). This chapter will present a few of the functions and later chapters will add to this list. For a complete list of the compile-time symbol table functions, see Appendix H. The functions we will consider in this chapter include the following:

- @size
- @defined
- @typeName
- @elements
- @elementSize

Without question, the @size function is probably the most important function in this group. Indeed, previous chapters have made use of this function already. The @size function accepts a single HLA identifier or constant expression as a parameter. It returns the size, in bytes, of the data type of that object (or expression). If you supply an identifier, it can be a constant, type, or variable identifier. HLA returns the size of the type of the object. As you've seen in previous chapters, this function is invaluable for allocating storage via *malloc* and allocating arrays.

Another very useful function in this group is the @defined function. This function accepts a single HLA identifier as a parameter, e.g.,

```
@defined( MyIdentifier )
```

This function returns true if the identifier is defined at that point in the program, it returns false otherwise.

The @typeName function returns a string specifying the type name of the identifier or expression you supply as a parameter. For example, if *i32* is an *int32* object, then "@typeName(i32)" returns the string "int32". This function is useful for testing the types of objects you are processing in your compile-time programs.

The @elements function requires an array identifier or expression. It returns the total number of array elements as the function result. Note that for multi-dimensional arrays this function returns the product of all the array dimensions⁴.

The @elementSize function returns the size, in bytes, of an element of an array whose name you pass as a parameter. This function is extremely valuable for computing indices into an array (i.e., this function computes the *element_size* component of the array index calculation, see "Accessing Elements of a Single Dimension Array" on page 465).

7.6.7 Compile-Time Expression Classification Functions

The HLA compile-time language provides functions that will classify some arbitrary text and determine if that text is a constant expression, a register, a memory operand, a type identifier, and more. Some of the more common functions are

- @isConst
- @isReg, @isReg8, @isReg16, @isReg32, @isFReg
- @isMem
- @isType

Except for @isType, which requires an HLA identifier as a parameter, these functions all take some arbitrary text as their parameter. These functions return true or false depending upon whether that parameter satisfies the function requirements (e.g., @isConst returns true if its parameter is a constant identifier or expression). The @isType function returns true if its parameter is a type identifier.

The HLA compile-time language includes several other classification functions that are beyond the scope of this chapter. See Appendix H for details on those functions.

4. There is an @dim function that returns an array specifying the bounds on each dimension of a multidimensional array. See the appendices for more details if you're interested in this function.

7.6.8 Miscellaneous Compile-Time Functions

The HLA compile-time language contains several additional functions that don't fall into one of the categories above. Some of the more useful miscellaneous functions include

- @odd
- @lineNumber
- @text

The @odd function takes an ordinal value (i.e., non-real numeric or character) as a parameter and returns true if the value is odd, false if it is even. The @lineNumber function requires no parameters, it returns the current line number in the source file. This function is quite useful for debugging compile-time (and run-time!) programs.

The @text function is probably the most useful function in this group. It requires a single string parameter. It expands that string as text in place of the @text function call. This function is quite useful in conjunction with the compile-time string processing functions. You can build an instruction (or a portion of an instruction) using the string manipulation functions and then convert that string to program source code using the @text function. The following is a trivial example of this function in operation:

```
?id1:string := "eax";
?id2:string := "i32";
@text( "mov( " + id1 + ", " + id2 + ");" )
```

The sequence above compiles to

```
mov( eax, i32 );
```

7.6.9 Predefined Compile-Time Variables

In addition to functions, HLA also includes several predefined compile-time variables. The use of most of HLA's compile time variables is beyond the scope of this text. However, the following you've already seen:

- @bound
- @into

Volume Three (see "Some Additional Instructions: INTMUL, BOUND, INTO" on page 393) discusses the use of these objects to control the emission of the INTO and BOUND instructions. These two boolean pseudo-variables determine whether HLA will compile the BOUND (@bound) and INTO (@into) instructions or treat them as comments. By default, these two variables contain true and HLA will compile these instructions to machine code. However, if you set these values to false, using one or both of the following statements then HLA will not compile the associated statement:

```
?@bound := false;
?@into := false;
```

If you set @BOUND to false, then HLA treats BOUND instructions as though they were comments. If you set @INTO to false, then HLA treats INTO instructions as comments. You can control the emission of these statements throughout your program by selectively setting these pseudo-variables to true or false at different points in your code.

7.6.10 Compile-Time Type Conversions of TEXT Objects

Once you create a text constant in your program, it's difficult to manipulate that object. The following example demonstrates a programmer's desire to change the definition of a text symbol within a program:

```
val
  t:text := "stdout.put";
```

```

.
.
.
?t:text := "fileio.put";

```

The basic idea in this example is that *t* expands to "stdout.put" in the first half of the code and it expands to "fileio.put" in the second half of the program. Unfortunately, this simple example will not work. The problem is that HLA will expand a text symbol in place almost anywhere it finds that symbol. This includes occurrences of *t* within the "?" statement above. Therefore, the code above expands to the following (incorrect) text:

```

val
  t:text := "stdout.put";
  .
  .
  .
  ?stdout.put:text := "fileio.put";

```

HLA doesn't know how to deal with the "?" statement above, so it generates a syntax error.

At times you may not want HLA to expand a text object. Your code may want to process the string data held by the text object. HLA provides a couple of operators that deal with these two problems:

- @string:identifier
- @toString:identifier

The @string:identifier operator consists of @string, immediately followed by a colon and a text identifier (with no interleaving spaces or other characters). HLA returns a string constant corresponding to the text data associated with the text object. In other words, this operator lets you treat a text object as though it were a string constant within an expression.

Unfortunately, the @string operator converts a text object to a string constant, not a string identifier. Therefore, you cannot say something like

```
?@string:t := "Hello"
```

This doesn't work because @string:t replaces itself with the string constant associated with the text object *t*. Given the former assignment to *t*, this statement expands to

```
?"stdout.put" := "Hello";
```

This statement is still illegal.

The @toString:identifier operator comes to the rescue in this case. The @toString operator requires a text object as the associated identifier. It converts this text object to a string object (still maintaining the same string data) and then returns the identifier. Since the identifier is now a string object, you can assign a value to it (and change its type to something else, e.g., *text*, if that's what you need). To achieve the original goal, therefore, you'd use code like the following:

```

val
  t:text := "stdout.put";
  .
  .
  .
  ?@tostring:t : text := "fileio.put";

```

7.7 Conditional Compilation (Compile-Time Decisions)

HLA's compile-time language provides an IF statement, #IF, that lets you make various decisions at compile-time. The #IF statement has two main purposes: the traditional use of #IF is to support *conditional*

compilation (or *conditional assembly*) allowing you to include or exclude code during a compilation depending on the status of various symbols or constant values in your program. The second use of this statement is to support the standard IF statement decision making process in the HLA compile-time language. This section will discuss these two uses for the HLA #IF statement.

The simplest form of the HLA compile-time #IF statement uses the following syntax:

```
#if( constant_boolean_expression )

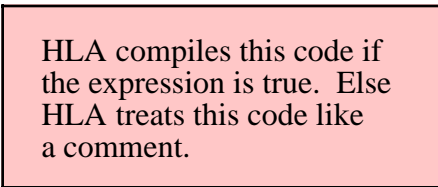
    << text >>

#endif
```

Note that you do not place semicolons after the #ENDIF clause. If you place a semicolon after the #ENDIF, it becomes part of the source code and this would be identical to inserting that semicolon immediately before the next text item in the program.

At compile-time, HLA evaluates the expression in the parentheses after the #IF. This must be a constant expression and its type must be boolean. If the expression evaluates true, then HLA continues processing the text in the source file as though the #IF statement were not present. However, if the expression evaluates false, then HLA treats all the text between the #IF and the corresponding #ENDIF clause as though it were a comment (i.e., it ignores this text).

```
#if( constant_boolean_expression )
```



HLA compiles this code if the expression is true. Else HLA treats this code like a comment.

```
#endif
```

Figure 7.2 Operation of HLA Compile-Time #IF Statement

Keep in mind that HLA's constant expressions support a full expression syntax like you'd find in a high level language like C or Pascal. The #IF expression syntax is not limited as are expressions in the HLA IF statement. Therefore, it is perfectly reasonable to write fancy expressions like the following:

```
#if( @length( someStrConst ) < 10 & ( MaxItems*2 < 100 | MinItems-5 < 10 ) )

    << text >>

#endif
```

Keep in mind that the items in a compile-time expression must all be CONST or VAL identifiers or an HLA compile-time function call (with appropriate parameters). In particular, remember that HLA evaluates these expressions at compile-time so they cannot contain run-time variables⁵. Also note that HLA's compile time language uses complete boolean evaluation, so any side effects that occur in the expression may produce undesired results.

The HLA #IF statement supports optional #ELSEIF and #ELSE clauses that behave in the intuitive fashion. The complete syntax for the #IF statement looks like the following:

```
#if( constant_boolean_expression1 )
```

5. Except, of course, as parameters to certain HLA compile-time functions like @size or @typeName.


```

    << text1 >>

#elseif( constant_boolean_expression2 )

    << text2 >>

#else

    << text3 >>

#endif

```

If the first boolean expression evaluates true then HLA processes the text up to the #ELSEIF clause. It then skips all text (i.e., treats it like a comment) until it encounters the #ENDIF clause. HLA continues processing the text after the #ENDIF clause in the normal fashion.

If the first boolean expression above evaluates false, then HLA skips all the text until it encounters a #ELSEIF, #ELSE, or #ENDIF clause. If it encounters a #ELSEIF clause (as above), then HLA evaluates the boolean expression associated with that clause. If it evaluates true, then HLA processes the text between the #ELSEIF and the #ELSE clauses (or to the #ENDIF clause if the #ELSE clause is not present). If, during the processing of this text, HLA encounters another #ELSEIF or, as above, a #ELSE clause, then HLA ignores all further text until it finds the corresponding #ENDIF.

If both the first and second boolean expressions in the example above evaluate false, then HLA skips their associated text and begins processing the text in the #ELSE clause. As you can see, the #IF statement behaves in a relatively intuitive fashion once you understand how HLA "executes" the body of these statements (that is, it processes the text or treats it as a comment depending on the state of the boolean expression). Of course, you can create a nearly infinite variety of different #IF statement sequences by including zero or more #ELSEIF clauses and optionally supplying the #ELSE clause. Since the construction is identical to the HLA IF..THEN..ELSEIF..ELSE..ENDIF statement, there is no need to elaborate further here.

A very traditional use of conditional compilation is to develop software that you can easily configure for several different environments. For example, the FCOMIP instruction makes floating point comparisons very easy but this instruction is available only on Pentium Pro and later processors. If you want to use this instruction on the processors that support it, and fall back to the standard floating point comparison on the older processors you would normally have to write two versions of the program - one with the FCOMIP instruction and one with the traditional floating point comparison sequence. Unfortunately, maintaining two different source files (one for newer processors and one for older processors) is very difficult. Most engineers prefer to use conditional compilation to embed the separate sequences in the same source file. The following example demonstrates how to do this.

```

const
    PentProOrLater: boolean := false; // Set true to use FCOMIxx instrs.
    .
    .
    .
    #if( PentProOrLater )

        fcomip(); // Compare st1 to st0 and set flags.

    #else

        fcomp(); // Compare st1 to st0.
        fstsw( ax ); // Move the FPU condition code bits
        sahf(); // into the FLAGS register.

    #endif

```

As currently written, this code fragment will compile the three instruction sequence in the #ELSE clause and ignore the code between the #IF and #ELSE clauses (because the constant *PentProOrLater* is

false). By changing the value of *PentProOrLater* to true, you can tell HLA to compile the single FCOMIP instruction rather than the three-instruction sequence. Of course, you can use the *PentProOrLater* constant in other #IF statements throughout your program to control how HLA compiles your code.

Note that conditional compilation does not let you create a single *executable* that runs efficiently on all processors. When using this technique you will still have to create two executable programs (one for Pentium Pro and later processors, one for the earlier processors) by compiling your source file twice; during the first compilation you must set the *PentProOrLater* constant to false, during the second compilation you must set this constant to true. Although you must create two separate executables, you need only maintain a single source file.

If you are familiar with conditional compilation in other languages, such as the C/C++ language, you may be wondering if HLA supports a statement like C's "#ifdef" statement. The answer is no, it does not. However, you can use the HLA compile-time function @DEFINED to easily test to see if a symbol has been defined earlier in the source file. Consider the following modification to the above code that uses this technique:

```
const
    // Note: uncomment the following line if you are compiling this
    // code for a Pentium Pro or later CPU.

    // PentProOrLater :=0; // Value and type are irrelevant
    .
    .
    .
    #if( @defined( PentProOrLater ) )

        fcomip();          // Compare st1 to st0 and set flags.

    #else

        fcomp();          // Compare st1 to st0.
        fstsw( ax );      // Move the FPU condition code bits
        sahf();           // into the FLAGS register.

    #endif
```

Another common use of conditional compilation is to introduce debugging and testing code into your programs. A typical debugging technique that many HLA programmers use is to insert "print" statements at strategic points throughout their code in order to trace through their code and display important values at various checkpoints. A big problem with this technique is that they must remove the debugging code prior to completing the project. The software's customer (or a student's instructor) probably doesn't want to see debugging output in the middle of a report the program produces. Therefore, programmers who use this technique tend to insert code temporarily and then remove the code once they run the program and determine what is wrong. There are at least two problems with this technique:

- Programmers often forget to remove some debugging statements and this creates defects in the final program, and
- After removing a debugging statement, these programmers often discover that they need that same statement to debug some different problem at a later time. Hence they are constantly inserting, removing, and inserting the same statements over and over again.

Conditional compilation can provide a solution to this problem. By defining a symbol (say, *debug*) to control debug output in your program, you can easily activate or deactivate *all* debugging output by simply modifying a single line of source code. The following code fragment demonstrates this:

```
const
    debug: boolean := false; // Set to true to activate debug output.
    .
    .
    .
```

```

#if( debug )

    stdout.put( "At line ", @lineNumber, " i=", i, nl );

#endif

```

As long as you surround all debugging output statements with a #IF statement like the one above, you don't have to worry about debug output accidentally appearing in your final application. By setting the *debug* symbol to false you can automatically disable all such output. Likewise, you don't have to remove all your debugging statements from your programs once they've served their immediate purpose. By using conditional compilation, you can leave these statements in your code because they are so easy to deactivate. Later, if you decide you need to view this same debugging information during a program run, you won't have to reenter the debugging statement - you simply reactivate it by setting the *debug* symbol to true.

We will return to this issue of inserting debugging code into your programs in the chapter on macros.

Although program configuration and debugging control are two of the more common, traditional, uses for conditional compilation, don't forget that the #IF statement provides the basic conditional statement in the HLA compile-time language. You will use the #IF statement in your compile-time programs the same way you would use an IF statement in HLA or some other language. Later sections in this text will present lots of examples of using the #IF statement in this capacity.

7.8 Repetitive Compilation (Compile-Time Loops)

HLA's #WHILE..#ENDWHILE statement provides a compile-time loop construct. The #WHILE statement tells HLA to repetitively process the same sequence of statements during compilation. This is very handy for constructing data tables (see "Constructing Data Tables at Compile Time" on page 996) as well as providing a traditional looping structure for compile-time programs. Although you will not employ the #WHILE statement anywhere near as often as the #IF statement, this compile-time control structure is very important when writing advanced HLA programs.

The #WHILE statement uses the following syntax:

```

#while( constant_boolean_expression )

    << text >>

#endif

```

When HLA encounters the #WHILE statement during compilation, it will evaluate the constant boolean expression. If the expression evaluates false, then HLA will skip over the text between the #WHILE and the #ENDWHILE clause (the behavior is similar to the #IF statement if the expression evaluates false). If the expression evaluates true, then HLA will process the statements between the #WHILE and #ENDWHILE clauses and then "jump back" to the start of the #WHILE statement in the source file and repeat this process.

```
#while( constant_boolean_expression )
```

HLA repetitively compiles this code as long as the expression is true. It effectively inserts multiple copies of this statement sequence into your source file (the exact number of copies depends on the value of the loop control expression).

```
#endwhile
```

Figure 7.3 HLA Compile-Time #WHILE Statement Operation

To understand how this process works, consider the following program:

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

static
  ary: uns32[5] := [ 2, 3, 5, 8, 13 ];

begin ctWhile;

  ?i := 0;
  #while( i < 5 )

    stdout.put( "array[ ", i, " ] = ", ary[i*4], nl );
    ?i := i + 1;

  #endwhile

end ctWhile;
```

Program 7.2 #WHILE..#ENDWHILE Demonstration

As you can probably surmise, the output from this program is the following:

```
array[ 0 ] = 2
array[ 1 ] = 3
array[ 2 ] = 4
array[ 3 ] = 5
array[ 4 ] = 13
```

What is not quite obvious is how this program generates this output. Remember, the #WHILE..#ENDWHILE construct is a compile-time language feature, not a run-time control construct. Therefore, the #WHILE loop above repeats five times during *compilation*. On each repetition of the loop, the HLA compiler processes the statements between the #WHILE and #ENDWHILE clauses. Therefore, the program above is really equivalent to the following:

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

static
    ary: uns32[5] := [ 2, 3, 5, 8, 13 ];

begin ctWhile;

    stdout.put( "array[ ", 0, " ] = ", ary[0*4], nl );
    stdout.put( "array[ ", 1, " ] = ", ary[1*4], nl );
    stdout.put( "array[ ", 2, " ] = ", ary[2*4], nl );
    stdout.put( "array[ ", 3, " ] = ", ary[3*4], nl );
    stdout.put( "array[ ", 4, " ] = ", ary[4*4], nl );

end ctWhile;
```

Program 7.3 Program Equivalent to the Code in Program 7.2

As you can see, the #WHILE statement is very convenient for constructing repetitive code sequences. This is especially invaluable for unrolling loops. Additional uses of the #WHILE loop appear in later sections of this text.

7.9 Putting It All Together

The HLA compile-time language provides considerable power. With the compile-time language you can automate the generation of tables, selectively compile code for different environments, easily unroll loops to improve performance, and check the validity of code you're writing. Combined with macros and other features that HLA provides, the compile-time language is probably the premier feature of the HLA language – no other assembler provides comparable features. For more information about the HLA compile time language, be sure to read the next chapter on macros.