Chapter 2: Advanced HLA Programming

2.1: Using Advanced HLA Features

This book uses several advanced features in the HLA language. Chances are pretty good that unless you ve spent a lot of time studying HLA prior to reading this book, you re not going to be familiar with many of these features. Now before you complain about the choice of using these features, and how this book should have avoided them in order to make the material more accessible to less than expert HLA programmers, just keep in mind that many of these advanced features were added to the HLA language specifically to support Windows programming. Therefore, avoiding these features would prove to be counter-productive.

Much of the advanced HLA programming we re going to wind up doing involves HLA compile-time language statements (including the macro processor), data declarations, high-level control structures, and high-level procedure call syntax. While *The Art of Assembly Language* covers each of these topics, this book isn t going to make the assumption that you ve carefully read those sections of *The Art of Assembly Language* (or even read AoA at all).Whether or not you ve studied this material in that book, you ll definitely want to carefully read this chapter. It contains important information that the rest of this book is going to use, with an emphasis on those HLA features that are important to Windows programmers.

2.2: HLA's High-Level Data Structure Facilities

One of the primary differences between high level languages and traditional assemblers is the support for *abstract data types* in high level languages versus only supporting low-level machine data types (e.g., bytes, words, and double words) in assembly language. One of the hallmarks of a high level assembler (like HLA, MASM, or TASM) is support for abstract data types and user-defined data types. The HLA assembler provides about the richest set of built-in data types you ll find in any assembler plus the ability to easily create new data types as the need arises. Because Windows uses sophisticated data structures in calls to Win32 API functions, you ll want to familiarize yourself with some of HLA s structured data types in order to write code that interfaces better with Windows.

2.2.1: Basic Data Types

Whereas a traditional assembler provides a few machine-level data types based on the sizes of those types, and then leaves it up to the assembly language programmer to manually interpret those types in a program, HLA provides a rich set of basic data types that denote the purpose of the object in addition to its size. For example, with a traditional assembler you might declare a variable to be a byte object; within your assembly language source code it is up to you to decide whether you want to treat that byte as a character variable, an unsigned integer, a boolean object, a signed integer, an enumerated data type, or some other value that you can represent with eight bits. With HLA, you can actually declare your variables to be `char, boolean, enum, int8, uns8`, etc., and make the intent of that variable s usage much clearer; HLA will even do some type checking when you use these data types to help find logical errors in your programs.

The first thing to note about HLAs abstract data types is that HLA *is an assembly language*. At the machine instruction level, the 80x86 CPU works with bytes, words, double-words, and other such primitive machine data types. Once you ve loaded an `int8` object into AL, for example, it is up to you to choose the correct machine instructions to treat that object as a signed integer. There is nothing stopping you from treating this as an unsigned integer object, as a character value, as a boolean value, etc. Do not, however, get the impression that HLAs abstract data typing facilities serve little purpose beyond providing documentative services that make your programs a little easier to read. HLA provides high-level control structures and procedure calling syntax, as well as a built-in compile-time language, that can take advantage of HLAs type-checking facilities. To achieve maximum benefit from HLA, therefore, you should choose the proper data types for your variables, constants, and other objects in HLA.

Although there are some very good reasons for using abstract (high-level) data types in an assembly language program, HLA doesn t lose sight of the fact that it is an assembler. Sometimes the use of high-level data abstractions interferes with the assembly language programming paradigm. In such situations, you ll want to use low-level machine data types to make your code more efficient. To support such situations, HLA supports the full range of low-level machine data types you ll find on the 80x86 CPU, including bytes, words, double words, quad words, ten-byte words, and long words (128-bit objects/16-byte objects). HLA uses the following type names for these primitive data types:

byte	Eight-byte (one byte) objects.
word	16-bit (two byte) objects
dword	32-bit (four byte) double word objects
qword	64-bit (eight byte) quad word objects
tbyte	80-bit (ten byte) ten byte objects
lword	128-bit (16 byte) long word objects

Note that some individuals might consider HLAs `real32`, `real64`, and `real80` data types to be primitive because the 80x86 CPU directly supports these types. However, HLA treats these types as abstract data types and performs type checking on them because of the special nature of their data.

The low-level data types provide a type-escape mechanism for HLA. The only type checking that HLA does on these machine level types is to ensure that the size of the object is appropriate for its current use (e.g., HLA will report an error if you try to load a byte object into a word-sized register). Other than the size check, which you can override if you desire, HLA will allow the use of any similarly-sized scalar object in place of one of these data types and you can use an object that has one of these data types in place of a similarly sized scalar object.

Note the use of the term scalar in the previous paragraph. What this means is that you can substitute any like-size single variable for a byte, word, dword, qword, tbyte, or lword object in an HLA statement. You may not, however, directly substitute an array, record, union, class, or other composite type in place of one of these data types, even if the size of the composite object is the same as the primitive data type. For example, you cannot directly substitute a two-byte array in place of a word object (though, using coercion, even this is possible in HLA; you ll find out more about coercion in just a little bit).

2.2.1.1: Eight Bit Data Types

HLA supports the following eight-bit data types, all of which are directly compatible with the `byte` type:

¥ boolean

¥ enum (HLA, by default, uses a byte to hold enum objects)

¥ uns8

¥ byte

¥ int8

¥ char

Directly compatible means that you may supply a byte object anywhere one of these other types is required and you may also supply any one of these other types wherever a byte type is required. Note that these data types are not compatible amongst themselves; that is, you cannot always supply, say, a `char` data type where an `uns8` object is required. The `byte` type is the only data type that is universally compatible with all of these types.

Note that HLA assigns the `byte` type to the 80x86 eight-bit registers, AL, AH, BL, BH, CL, CH, DL, and DH. Therefore, these registers are automatically compatible with any byte-sized object using one of HLAs single byte types.

HLA uses the `boolean` type to represent true/false values. Technically, a `boolean` variable requires only a single bit to represent the two possible values; however, the 80x86 CPU can only efficiently access objects that are multiples of eight bits. Therefore, HLA sets aside a whole byte for a `boolean` object. HLA represents `false` with the value zero and `true` with the value one. Traditionally, programmers have used zero for `false` and any non-zero value to represent `true`; therefore, HLAs definition is upwards compatible to the traditional (i.e., C) definition. In general, when using boolean values, you should always store a zero or one into a boolean variable and test for zero/not zero when checking `boolean` values. This will provide the maximum compatibility with all uses of this data type. Note that arithmetic expressions involving `boolean` operands may produce strange results when combining operands using the AND and OR operators. The next chapter discusses this issue in detail. Just be aware of this for the time being.

Enumerated data types are data types that associate sequential numeric values with symbolic constants. Internally, HLA treats `enum` objects as unsigned integers (e.g., for purposes of comparison), though `enum` objects are not directly compatible with `uns8` objects. An enumerated data type in HLA consists of a list of identifiers to which HLA assigns sequential unsigned integer values, starting with zero. For example, consider the following HLA type declaration:

```
type
   color_t :enum{  red, green, blue }:
```

HLA internally assigns the value zero with `red`, the value one with `green`, and the value two with `blue`. It is important to note, however, that `red, green,` and `blue` are not `uns8` objects. They are of type `color_t` and this will make a difference in certain circumstances. For more details on enumerated data types in HLA, please consult the *HLA Reference Manual*.

HLA uses the `uns8` data type to represent unsigned integer values in the range 0..255. This data type is fundamentally equivalent to the `byte` type except that `uns8` objects are not automatically compatible with the other eight-bit data types. In boolean expressions, HLA always defaults to using unsigned comparisons and conditional jump instructions when the operands are `uns8` operands.

HLA uses the int8 type to represent signed integers in the range -128..+127. In boolean expressions, HLA always uses signed comparisons and conditional jumps with at least one of the operands is signed (note that if you mix an unsigned and signed operand across a relational operator, HLA defaults to use a signed comparison).

Internally, HLA treats the `char` data type as an eight-bit unsigned integer value. Though not directly compatible with the `uns8` type, HLA uses unsigned comparisons and conditional jumps whenever it encounters a char operand in a run-time boolean expression.

As noted earlier, the `byte` type is fundamentally compatible with all other scalar eight-bit types. When two byte types appear in a run-time boolean expression around the same operand, HLA compares the two values using an unsigned comparison. If a byte operand appears with another eight-bit type around some relational operator in a run-time boolean expression, then HLA uses whatever type of comparison is appropriate for that other operand (i.e., unsigned for `enum, char, boolean,` and `uns8` operands or signed for `int8` operands).

There are three major areas where HLA differentiates the types of eight-bit objects: in run-time boolean expressions, in procedure call parameter lists, and in compile-time (constant) expressions. We ll address each of these areas separately in later sections of this chapter.

2.2.1.2: Sixteen-Bit Data Types

HLA supports the following 16-bit data types, all of which are directly compatible with the `word` type:

¥ int16

¥ uns16

¥ wchar

¥ word

HLA uses `int16` objects to represent signed values in the range -32768..+32767. Whenever HLA encounters an `int16` object in a run-time boolean expression, it uses signed comparisons and conditional jump instructions, even if the operand opposite the `int16` object in a relational expression is an unsigned operand.

HLA uses `uns16` objects to represent unsigned values in the range 0..65535. Whenever HLA encounters an uns16 object in a run-time boolean expression, and the corresponding object opposite the relational operator is not an `int16` value, HLA uses unsigned comparison and jump instructions.

HLA uses `wchar` objects to represent unicode characters. These are unsigned objects (though not directly compatible with `uns16` values) so HLA uses unsigned comparisons and conditional jump instructions whenever you compare `wchar` objects in a run-time boolean expression.

Objects of type `word` are directly compatible with any of the other 16-bit ordinal data types. If an object of type `word` appears in a run-time boolean expression, then HLA will emit code that uses the type of comparison required by the other operand in the relational expression. If the other operand is also of type `word`, HLA uses an unsigned comparison and jump to compare the two operands.

2.2.1.3: Thirty-Two-Bit Data Types

HLA supports the following 32-bit data types, all of which are directly compatible with the `dword` data type:

¥ dword

¥ int32

¥ uns32

¥ pointer types

¥ procedure pointer types

HLA also supports the following 32-bit data types, though these types are not directly compatible with the `dword` type:

¥ string (this is a pointer type)

¥ unicode (this is a pointer type)

¥ real32

The behavior of the `uns32` and `int32` types, with respect to the dword word, is similar to the relationship between `uns8/int8` and `byte`, and `uns16/int16` and `word`. Take a look at the previous sections for more details.

HLA generally treats pointer types as `dword` objects within most run-time expressions. In particular, HLA will perform an unsigned comparison when comparing two pointer objects.

Procedure pointer types are also treated just like `dwords` by HLA with one major exception: HLA provides implicit syntax for calling a procedure indirectly using HLA s high level procedure call syntax. So although you can treat a procedure pointer just like a dword object within a run-time boolean/relational expression, you cannot directly call a procedure via a pointer to a procedure in a `dword` variable using HLA s high-level procedure call syntax. Such activity is only available when using procedure pointers (or when specifying an explicit coercion operator). We ll take another look at HLA s procedure pointer objects in a few sections.

Although HLA string variables are pointers, HLA treats strings as a unique data type for the purposes of type checking. In particular, `dword` and `string` objects aren t always interchangeable nor are they even type compatible in all cases. To confuse the issue even farther, certain Win32 API function calls allow you to pass a small integer value (16 bits or less) in place of a string pointer as a function parameter. Therefore, in certain cases, HLA will allow you to substitute a small integer constant in place of an actual string pointer when calling a function (this use, however, is being depreciated, so don t count on it in future versions of HLA). Generally, though, you can use a string variable (not a constant) wherever a `dword` object is expected; you may not, however, always use a `dword` object where a `string` object is expected. This same discussion applies to the `unicode` data type (which is a string of unicode characters).

Because floating point values on the 80x86 are fundamentally different than integer values, HLA does not allow you to mix `real32` and `dword` objects in the same expression. Indeed, HLA doesn t even allow `real32` objects in a boolean/relational expression (though you may pass `real32` objects as procedure parameters). For that reason, we ll not consider `real32` objects here. See *The Art of Assembly Language* for more details concerning floating point arithmetic on the 80x86.

2.2.1.4: Sixty-Four-Bit Data Types

HLA supports a couple of 64-bit data types:

¥ int64

¥ real64

¥ uns64

¥ qword

As with the 32-bit data types, `qword` is directly compatible with `int64` and `uns64` types. The issue of `int64` and `uns64` compatibility isn t as much of an issue because the 80x86 doesn t directly support 64-bit comparisons with the integer instruction set[1]. Also like the 32-bit data types, the `real64` type is not directly compatible with the `qword` type, nor may you compare real64 objects within an HLA run-time boolean expression.

2.2.1.5: Eighty-Bit Data Types

HLA supports two 80-bit data types - the `tbyte` type and the `real80` type. Neither type is directly compatible with the other and HLA doesn t allow the comparison of 80-bit objects within a run-time boolean expression. `Tbyte` objects usually hold BCD values and `real80` objects hold floating point values. We ll not consider these two types any farther, here.

2.2.1.6: One Hundred Twenty-Eight Bit Data Types

HLA supports several 128-bit data types, they are:

¥ cset

¥ int128

¥ lword

¥ real128[2]

¥ uns128

Though HLA does not support the use of `int128, lword,` and `uns128` objects in a run-time boolean expression (because the 80x86 CPU doesn t support the direct comparison of 128-bit objects using the integer instruction set), you can still pass these objects as procedure parameters and use them in compile-time expressions, so it s worth mentioning that the relationship they share is similar to the `dword/uns32/int32` relationship sans the ability to compare them. The `cset` data type is, technically, a composite data type (character sets can be thought of as an array of 128 bits); therefore, there is no fundamental relationship between an `lword` and a `cset` other than their size. similarly, `real128` data types are also composite objects (it s an array of two real64 objects) so they aren t directly compatible with `lwords`, either. Because Windows doesn t use 128-bit data types very often, we ll not consider these data types any further.

2.2.2: Composite Data Types

HLA supports a fair number of *composite* data types as well as scalar data types. As the name suggests, a composite data type is one that is composed (or made up) of other data types. HLA provides direct support for the following composite data types: arrays, records (structures), unions, classes, pointers, procedure pointers, character sets, and thunks. Character sets, pointers, and procedure pointers are a special case - HLA provides direct support for these composite data types so that you can often treat these types as scalar (though not ordinal) types. Though it s sometimes convenient to treat these objects as scalars, there are times when it s more convenient to think of them as composite objects. Thus we ll consider them again in the following subsections.

1. We ll ignore MMX and SSE operands because they require special instructions that HLA s run-time boolean expressions do not support.
2. Real128 is actually a composite data type, specifically a record, that HLA supports for SSE/2 compatibility. In theory, it is not a scalar data type. However, we ll mention it here just for completeness.

2.2.2.1: HLA Array Types

HLA uses a high-level-like syntax for the declaration of arrays in your assembly language programs. A typical (run-time) array variable declaration might look like the following:

```
static
   runTimeArray :int32[ 16];
```

This declaration sets aside storage for a sequence of 16 32-bit integers in memory. The name `runTimeArray` refers to the address of the first element of this array. The next variable immediately following this array in memory will be at an address that is (at least) 64 bytes later (16 array elements times four bytes for each element is 64 bytes). You can even declare multi-dimensional arrays in HLA using a high level like syntax, e.g.,

```
type
   two_dim_array : char[  4, 16 ]; // 64 bytes of storage
   three_dim_array : dword[  4, 2, 2 ]; // Also 64 bytes of storage
   four_dim_array : real32[  2, 2, 2, 2 ]; // Also 64 bytes of storage
```

Note that multi-dimensional arrays consume as much storage as the product of their dimensions, also multiplied by the size of a single array element.

As is true in all assembly languages, indexes into an array start at zero and the last element (of a particular dimension or index) ends at the array bounds minus one. For example, the elements of the `runTimeArray` example given earlier range from index zero through 15.

One very fundamental point, which is the cause of much confusion to assembly language programmers and especially to HLA programmers, is that you do not index into an array using the same syntax and concepts as you do in a high level language. The fact that HLA uses a high-level language-like syntax for array declarations falsely propagates this idea. The fact that HLA s compile-time language works the way high level languages do, in direct contrast to the run-time (i.e., 80x86 assembly) language also aids in this confusion. *Don t get caught in this trap!*

Although both *The HLA Reference Manual* and *The Art of Assembly Language Programming* discuss accessing elements of arrays, emails and posts to assembly language related newsgroups indicate that this is one area that causes problems for assembly programmers (especially beginning assembly programmers). So it s worthwhile repeating some of that information here.

The most fundamental mistake people make when indexing into array in assembly language code is that they forget that they must multiply the index into the array by the size of an array element. More often than not, a beginning assembly language programmer will translate code like the following:

eax = runTimeArray[i]; // C example

into assembly code that looks like this:

```
   mov( i, ebx ); // Must load array index into a 32-bit register to use indexed addressing
   mov( runTimeArray[  ebx ], eax );
```

Nice try, but completely incorrect. The problem is that the 80x86 indexed addressing modes compute *byte offsets from some base address*, not element indexes into an array. That is, the runTimeArray[ebx] addressing mode computes the effective address obtained by adding the current value in EBX to the address associated with the `runTimeArray` label. If EBX contains five and the memory address associated with the *runTimeArray* label is $40_0000, then this addressing mode references the object at address $40_0005 in memory. Note that this is *not* the address of the sixth element of the `runTimeArray` array. Each element of the `runTimeArray` object consumes four bytes, so the elements appear in memory as follows:

runTimeArray[0]-	$40_0000
runTimeArray[1]-	$40_0004
runTimeArray[2]-	$40_0008
runTimeArray[3]-	$40_000C
runTimeArray[4]-	$40_0010
runTimeArray[5]-	$40_0014
runTimeArray[6]-	$40_0018
runTimeArray[7]-	$40_001C
runTimeArray[8]-	$40_0020
runTimeArray[9]-	$40_0024
runTimeArray[10]-	$40_0028
runTimeArray[11]-	$40_002C
runTimeArray[12]-	$40_0030
runTimeArray[13]-	$40_0034
runTimeArray[14]-	$40_0038
runTimeArray[15]-	$40_003C

The address $40_0005 (obtained as the effective address of the expression `runTimeArray[ebx]` when EBX contains five) is actually the address if the second byte of the `runTimeArray[1]` element. This probably isn t what the programmer had in mind.

As you can see in this list, each element of `runTimeArray` is located four bytes in memory apart. This makes perfect sense because each element of the array consumes four bytes of memory and we d like the array elements to occupy adjacent memory cells (that is, each element of the array should immediately follow the previous element of the array in memory). In order to achieve this, you must multiply the array index by the size of each array element (in bytes). Because each element of the array is a double word (four bytes), we have to multiply the index into the array by four and add the base address of the array (that is, the address of the first element) to obtain the actual address of the desired element. In the current example, if we want to access the element at index five (that is, the sixth element of the array), then we need to multiply the index (five) by four before adding in the base address. Five times four is 20 ($14), adding this to the base address produces $40_0014 in the current example. If you ll look at the list for `runTimeArray` element addresses, you ll find that `runTimeArray[5]` does, indeed, sit at address $40_0014 in memory.

Because programs commonly access elements of arrays whose element sizes are one, two, four, or eight bytes long, the 80x86 CPU provides a special set of addressing modes, the scaled indexed addressing modes, that let you easily compute the index into these arrays. These addressing modes use syntax like the following:

```
byteArray[ eax*1 ] -- note that this is functionally identical to byteArray[ ebx ]
wordArray[ ebx*2 ]
dwordArray[ ecx*4 ]
qwordArray[ edx*8 ]
```

The register you specify in the scaled indexed addressing mode must be a general-purpose 32-bit integer register; see *The Art of Assembly Language* for more details.

If you need to access an element of an array whose element sizes are not one, two, four, or eight bytes, then you ve got to manually multiply the index by the size of an array element (typically using a `shl` or `intmul` instruction). For example, if you ve got an array of `real80` objects you ll need to multiply the index into the array by 10 (10 bytes = 80 bits) before accessing the particular element. Here s some sample code that demonstrates this:

```
// Push element "i" of real80Array onto the FPU stack:

   intmul( 10, i, ecx ); // ecx = 10 * i
   fld( real80Array[ ecx ] );
```

Note that you do not use a scaled indexed addressing mode when you manually multiply the index by the element size. You ve already done the multiplication to obtain a byte offset into the array.

Accessing an element of a multidimensional array requires the use of a more complex calculation. In the Windows API world, all two-dimensional arrays are stored in a form known as *row-major order*. Rather than get into the complexities of row-major (versus column-major) addressing here, we ll just use the HLA Standard Library `array.index` function that does multi-dimensional array computations for you automatically. Those who are interested in the low-level implementation of multi-dimensional array access should take a look at the chapter covering arrays in *The Art of Assembly Language Programming*.

The HLA Standard Library `array.index` macro automatically computes the index into an array (with any number of dimensions). Because this is a macro, it directly generates code that is almost as good as hand-written assembly language code (in a few special cases you might be able to generate slightly better code by hand, but most of the time this macro generates exactly the same code you d manually write). To use the `array.index` macro, you must either include *stdlib.hhf* at the beginning of your source file or, at least, include *arrays.hhf*. The `array.index` macro uses the following invocation syntax:

array.index(<32-bit register>, <array name>, <<list of array indices>>);

The first argument must be a 32-bit general purpose integer register (i.e., EAX, EBX, ECX, EDX, ESI, EDI, EBP, or ESP). Generally, you would not use EBP or ESP here (because they have special purposes and they generally aren t available for use in accessing array elements).

The second argument to the `array.index` macro is the name of the array whose elements you want to access. HLA actually supports dynamically allocated multi-dimensional arrays in the arrays.hhf module, we ll not worry about those here (see the HLA Standard Library documentation for more details); instead, this book will simply assume that you ve supplied the name of an array you ve declared in a `static, readonly, storage,` or `var` declaration section, or as a procedure pass by value parameter. Any of the array examples appearing in this section would work fine, for example.

The third (through nth) parameter(s) specify the constants, variables, or registers that you want to use as the indexes into this array. The index objects must all be 32-bit integer values (presumably unsigned, though the macro accepts signed values too).

The `array.index` macro computes the address of the specified array element and leaves that address sitting in the 32-bit register you specify as the first argument in the `array.index` argument list. To access the specified array element, all you need to do is reference the memory location pointed at by that 32-bit register after the `array.index` invocation. Note that you do not have to multiply this value by the size of the array element (that s already done for you) nor do you have to add in the base address of the array (that s also done for you). Here are some examples of accessing array elements using the `array.index` macro:

```
// j = runTimeArray[ i ];

   array.index( ebx, runTimeArray, i );
   mov( [ebx], eax );
   mov( eax, j );

// c = two_dim_array[ i, j ];

   array.index( edx, two_dim_array, i, j );
   mov( [edx], al );
   mov( al, c );

// esi = three_dim_array[ i, j, k ]

   array.index( ecx, three_dim_array, i, j, k );
   mov( [ecx], edx );

// push four_dim_array[ eax, ebx, ecx, edx ] onto the FPU stack:

   array.index( edi, four_dim_array, eax, ebx, ecx, edx );
   fld( (type real32 [edi]) );
```

The HLA compile-time language provides some built-in compile-time functions that return values of interest to an HLA programmer using arrays in an HLA program. During compilation, HLA replaces each of these compile-time functions with a constant that is the result of computing the compile-time function s result. You may use these compile-time functions anywhere a constant is legal in an HLA program. See Table 2-1 for a list of the applicable functions.

Table 2-1: HLA Compile-Time Functions That Provide Useful Array Information

Function	Sample Call	Description
@size	@size(array_name)	Returns the size, in bytes, of the entire array (the run-time size of the array).
@elements	@elements(array_name)	Returns the total number of elements declared for an array. If the array is a multi-dimensional array, this returns the product of all the array bounds for that array (i.e., the total number of elements).
@elementsize	@elementsize(array_name)	Returns the run-time size, in bytes, of a single element of the array.
@arity	@arity(array_name)	Returns the number of dimensions of the array.
@dim	@dim(array_name)	Returns a compile-time constant array where each element of the array specifies the bounds for one of the dimensions of the array.
	@dim(array_name)[index]	Returns the number of elements of a particular dimension of an array (note that index is zero-based).

You can use the compile-time functions in Table 2-1 to automate much of your array calculations. This can make your programs much easier to maintain by avoiding hard-coded constants in your program. For example, suppose you have an array of some type (`myType_t`) whose definition is subject to change as you modify the program. Computing an index into this array is a bit of a problem; for example, if you know that the size of an element in the array is six bytes, you could (manually) compute an index into that array using code like the following:

```
intmul( 6, index, ebx );
lea( eax, myArray[ ebx] );  // Load eax with the address of myArray[ index] .
```

The problem with this approach is that it fails if you decide to change the definition of `myType_t` (`myArray's` type) without manually adjusting this code to deal with the fact that `myArray's` size is no longer six bytes. Now consider the following implementation of this code sequence:

```
intmul( @elementsize( myArray ), index, ebx );
lea( eax, myArray[  ebx ] );
```

This code sequence automatically recomputes the appropriate index into the array regardless of `myArray's` size (because HLA automatically substitutes the correct size for `@elementsize`). About the only drawback to this scheme is that it hides the fact that certain indexes (1, 2, 4, and 8) might be better computed using an 80x86 scaled indexed addressing mode, though it is possible to fix this problem by using a macro (the HLA Standard Library `array.index` function, for example, automatically checks for these special element sizes and adjusts the code it outputs in an appropriate fashion for these special cases).

HLA is interesting insofar as it allows you to declare and use compile-time constant arrays as well as run-time memory arrays. Few other languages, much less assemblers, provide the concept of a compile-time array. HLAs compile-time language makes extensive use of these compile-time arrays and you can use compile-time arrays (also known as array constants) to initialize run-time arrays at program load time (i.e., arrays you declare in the `static` and `readonly` sections). There is, however, one important issue of which you need to be aware - HLA compile-time arrays/array constants use semantics that are appropriate for the HLA compile-time language (which is a high level language). These semantics are different than the semantics for a run-time (assembly language) array. In particular, indexes into array constants are actual array indexes, not byte offsets. Therefore, you do not multiply the index of a constant array by the element size. Furthermore, HLA automatically computes the row-major index into a constant array for you, you do not have to use array.index or manually compute the index yourself. We ll return to this subject a little later in this chapter when we discuss the HLA compile-time language. Just be aware of the fact that compile-time array constants are fundamentally different that run-time assembly language arrays.

2.2.2.2: HLA Union Types

A discriminant union is a data type that combines several different objects so that they use the same storage in memory. In theory, access to each of the variables in a discriminant union are mutually exclusive; because the variables share the same storage in memory, storing a value into one of these variables disturbs the values of the other variables sharing that same memory address. Some high level language programmers use discriminant union types as a sneaky way to retype data in memory. Obviously, this feature is of little value in an assembly language program where is trivially easy to recast data in memory as some other type. However, there are two primary purposes for using a discriminate union, both of which apply in assembly language as well as in high level languages: memory conservation and the creation of variant types. Because many assembler authors perceive discriminant unions as a type casting trick, very few assemblers support the union data type. Unfortunately (for those assemblers), many Windows data types use unions and this makes using such assemblers inconvenient when working with these Windows data structures. Fortunately, HLA provides full support for discriminant union data types, so this isn t a problem at all in HLA.

HLA implements the discriminant union type using the `union..endunion` reserved words. The following HLA type declaration demonstrates a `union` declaration:

```
type
    allInts:
        union
            i8  :int8;
            i16 :int16;
            i32 :int32;
        endunion;
```

Each entry variable declaration appearing between the `union` and endunion keywords is a *field* of that `union`. All fields in a `union` have the same starting address in memory. The size of a `union` object is the size of the largest field in the union. The fields of a `union` may have any type that is legal in a variable (HLA `var`) declaration section.

Given a `union` object, say `i` of type `allInts`, you access the fields of the `union` by specifying the `union` variable name, a period (dot), and the field name. The following 80x86 `mov` instructions demonstrate how to access each of the fields of an `i` variable of type `allInts`:

```
   mov( i.i8, al );
   mov( i.i16, ax );
   mov( i.i32, eax );
```

A good example of a `union` type in action is the Windows `w.CHARTYPE` data type:

```
type
    CHARTYPE:
        union
            UnicodeChar: word;
            AsciiChar: byte;
        endunion;
```

The Windows `w.CHARTYPE` is a perfect example of a union data structure that encapsulates a couple of mutually exclusive values (that is, you wouldn t use the `UnicodeChar` and `AsciiChar` fields of this union simultaneously). Most Windows applications use either ASCII (ANSI) characters or Unicode characters; few applications would attempt to use both character sets in the same program (other than to convert one form to the canonical form that the application uses). Windows, as a general rule, provides two sets of API functions that deal with characters - one for ASCII and one for Unicode. By playing games with C preprocessor macros and accompanying data types (like `w.CHARTYPE`), Windows allows C programmers to call either set of functions while passing the same set of parameters. Of course, the programmer has to know which character type to supply (i.e., whether to store their data in the `Unicode` or `AsciiChar` fields), but the interface is the same either way.

Most of the unions you ll find in Windows data structures represent mutually exclusive data fields, like the fields belonging to `w.CHARTYPE`. Depending on your circumstances (e.g., whether your program works with ASCII or Unicode characters) you use one or the other of these two fields in your program and you will probably not use the other field at all. Another common use of discriminant union types is to create *variant* objects. A variant object is a variable that has a *dynamic type*, that is, the program determines the type of the object at run-time and, in fact, the variable s type can change under program control. Dynamically typed objects are great when you cannot anticipate exactly what type of data the user will enter while the program is running. The typical implementation of a variant data type consists of two components: a *tag value* that describes the current (dynamic) type of the object and a discriminant union value that can hold any of the possible types. During program execution, the software uses a `switch/case` statement (or similar logic) to execute appropriate code based on the current type of the dynamically typed object. Although the use of variant types is interesting and useful, Windows doesn t make much use of this advanced feature, so we won t discuss it any farther here.

2.2.2.3: HLA Record (Structure) Types

Records (structures) are probably the most important high level data structure an assembler can provide for Win32 assembly language programmers. In fact, if an assembler does not provide decent, built-in, capabilities for declaring and using records, you simply don t want to use that assembler for developing Win32 programs. Windows APIs make extensive use of records (C structs) and attempting to write assembly code with an assembler that doesn t properly support records is going to be a burden. Fortunately, HLA is not one of these assemblers - HLA provides excellent support for records. In this section we ll explore HLA s support for this important user-definable data type.

The record data type provides a mechanism for collecting together different variables into a physical unit so the program can treat this disparate collection of objects as a single entity. At the most basic level an array is also such a data type. However, the difference between an array and a record is the fact that all the elements of an array must have the same type, this restriction does not exist for the elements (*fields*) of a record. Another difference between an array and a record is that you reference the elements of an array using a numeric index whereas you access the fields of a record by the fields names.

HLA s records allow programmers to create data types whose fields can be different types. The following HLA type declaration defines a simple record with four fields:

```
type
    Planet:
        record
            x       :int32;
            y       :int32;
            z       :int32;
            density :real64;
        endrecord;
```

Objects of type `Planet` will consume 20 bytes of storage at run-time.

The fields of a `record` may be of any legal HLA data type including other composite data types. Like unions, anything that is legal in a `var` section is a legal field of a `record`. Also like unions, you use the dot-notation to access fields of a record object. For example, you could use the following 80x86 instructions to access an object `plnt` of type `Planet`:

```
mov( plnt.x, eax );
mov( edx, plnt.y );
add( plnt.z, ebx );
fld( plnt.density );
```

There are several advantages to using records versus simply creating separate variables for each of the fields of the record. First of all, a record type provides a template that makes it very easy to create new instances (copies) of that record type. For example, if you need nine planets, you can easily create them in an HLA `static` section as follows:

```
static
    Mercury  :Planet;
    Venus    :Planet;
    Earth    :Planet;
    Mars     :Planet;
    Jupiter  :Planet;
    Saturn   :Planet;
    Uranus   :Planet;
    Neptune  :Planet;
    Pluto    :Planet;
```

Imagine the mess you would have if you had to create four separate variables for each of these nine planets (e.g., `Mercury_x, Mercury_y, Mercury_z, Mercury_density`, etc.).

Another solution, of course, is to create an array of records, with one array element for each planet. You can easily do this as follows:

```
static
   Planets :Planet[ 9] ;
```

Each element of the array will have all the fields associated with the `Planet` data type. Because HLA treats an array name as though it were the first element of the array, you can access the fields of the `Planets` array using the same dot-notation syntax you use to access fields of a single `Planet` object, e.g.,

```
   mov( Planets.x, eax ); // Moves field "x" of Planets[ 0]  into eax.
```

If you want to access a field of some element other than the first element of an array of records, you tack on the indexed addressing mode and supply an appropriate index into the array, e.g.,

```
   intmul( @elementsize( Planets ), index, ebx );
   mov( Planets.x[ ebx] , eax );
```

Note that the indexed addressing mode specification follows the entire variable s name, unlike most high level languages, like C, where you would inject the array index into the name immediately after `Planets`. Keep in mind that you can also use HLAs array.index macro to index into an array of records, though you will need to coerce the resulting pointer to the `Planet` type in order to access the fields of that record, e.g.,

```
   array.index( ebx, Planets, index );
   mov( (type Planet [ ebx] ).x, eax );
```

In addition to saving you the effort of creating separate variables for each field, records also let you encapsulate related data, making it easier to copy record variables as single entities, manipulate record objects via pointers, and pass records as parameters to procedures. For example, it s much easier to pass a single parameter of type `Planet` to some procedure rather than passing the four separate fields that the `Planet` type encapsulates (if you don t think passing four parameters is a big deal, just keep in mind that many Win32 structures, that often appear as procedure parameters, have *dozens* of fields).

Beyond the issues of convenience and efficiency, there is one other important reason for using records in your assembly language programs: they re easier to read and maintain. Assembly language code is hard enough to read as it is; every opportunity you get to produce more readable assembly code is an opportunity you should take.

Record types may *inherit fields* from other record types. Consider the following two HLA type declarations:

```
type
    Pt2D:
        record
            x: int32;
            y: int32;
        endrecord;

    Pt3D:
        record inherits( Pt2D )
            z: int32;
        endrecord;
```

In this example, `Pt3D` (point 3-D) inherits all the fields from the `Pt2D` (point 2-D) type. The `inherits` keyword tells HLA to copy all the fields from the specified record (`Pt2D` in this example) to the beginning of the current record declaration (`Pt3D` in this example). Therefore, the declaration of `Pt3D` above is equivalent to:

```
type
    Pt3D:
        record

            x: int32;
            y: int32;
            z: int32;

        endrecord;
```

In some special situations you may want to override a field from a previous field declaration. For example, consider the following record declarations:

```
type
    BaseRecord:
        record
            a: uns32;
            b: uns32;
        endrecord;

    DerivedRecord:
        record inherits( BaseRecord )
            b: boolean;  // New definition for b!
            c: char;
        endrecord;
```

Normally, HLA will report a duplicate symbol error when attempting to compile the declaration for `DerivedRecord` since the `b` field is already defined via the `inherits( BaseRecord )` option. However, in certain cases it s quite possible that the programmer wishes to make the original field inaccessible in the derived class by using a different name. That is, perhaps the programmer intends to actually create the following record:

```
type
    DerivedRecord:
        record
            a: uns32;    // Derived from BaseRecord
            b: uns32;    // Derived from BaseRecord, but inaccessible here.
            b: boolean;  // New definition for b!
            c: char;
        endrecord;
```

HLA allows a programmer explicitly override the definition of a particular field by using the `overrides` keyword before the field they wish to override. So while the previous declarations for `DerivedRecord` produce errors, the following is acceptable to HLA:

```
type
    BaseRecord:
        record
            a: uns32;
            b: uns32;
        endrecord;

    DerivedRecord:
        record inherits( BaseRecord )
            overrides b: boolean;  // New definition for b!
            c: char;
        endrecord;
```

Normally, HLA aligns each field on the next available byte offset in a record. If you wish to align fields within a record on some other boundary, you may use the `align` directive to achieve this. Consider the following record declaration as an example:

```
type
    AlignedRecord:
        record
            b:boolean;              // Offset 0
            c:char;                 // Offset 1
            align(4);
            d:dword;                // Offset 4
            e:byte;                 // Offset 8
            w:word;                 // Offset 9
            f:byte;                 // Offset 11
        endrecord;
```

Note that field `d` is aligned at a four-byte offset while `w` is not aligned. We can correct this problem by sticking another `align` directive in this record:

```
type
    AlignedRecord2:
        record
            b:boolean;              // Offset 0
            c:char;                 // Offset 1
            align(4);
            d:dword;                // Offset 4
            e:byte;                 // Offset 8
            align(2);
            w:word;                 // Offset 10
            f:byte;                 // Offset 12
        endrecord;
```

Be aware of the fact that the align directive in a `record` only aligns fields in memory if the record object itself is aligned on an appropriate boundary. For example, if an object of type `AlignedRecord2` appears in memory at an odd address, then the `d` and `w` fields will also be misaligned (that is, they will appear at odd addresses in memory). Therefore, you must ensure appropriate alignment of any record variable whose fields you re assuming are aligned.

Note that the `AlignedRecord2` type consumes 13 bytes. This means that if you create an array of `AlignedRecord2` objects, every other element will be aligned on an odd address and three out of four elements will not be double-word aligned (so the `d` field will not be aligned on a four-byte boundary in memory). If you are expecting fields in a record to be aligned on a certain byte boundary, then the size of the record must be an even multiple of that alignment factor if you have arrays of the record. This means that you must pad the record with extra bytes at the end to ensure proper alignment. For the `AlignedRecord2` example, we need to pad the record with three bytes so that the size is an even multiple of four bytes. This is easily achieved by using an `align` directive as the last declaration in the record:

```
type
    AlignedRecord2:
        record
            b:boolean;              // Offset 0
            c:char;                 // Offset 1
            align(4);
            d:dword;                // Offset 4
            e:byte;                 // Offset 8
            align(2);
            w:word;                 // Offset 10
            f:byte;                 // Offset 12
            align(4)                // Ensures we’re padded to a multiple of four bytes.
        endrecord;
```

Note that you should only use values that are integral powers of two in the `align` directive.

If you want to ensure that all fields are appropriately aligned on some boundary within the record, but you don t want to have to manually insert `align` directives throughout the record, HLA provides a second alignment option to solve your problem. Consider the following syntax:

```
type
    alignedRecord3 : record[ 4]
        << Set of fields >>
    endrecord;
```

The [4] immediately following the `record` reserved word tells HLA to start all fields in the record at offsets that are multiples of four, regardless of the object s size (and the size of the objects preceding the field). HLA allows any integer expression that produces a value in the range 1..4096 inside these parenthesis. If you specify the value one (which is the default), then all fields are packed (aligned on a byte boundary). For values greater than one, HLA will align each field of the record on the specified boundary. For arrays, HLA will align the field on a boundary that is a multiple of the array element s size. The maximum boundary HLA will round any field to is a multiple of 4096 bytes.

Note that if you set the record alignment using this syntactical form, any `align` directive you supply in the record may not produce the desired results. When HLA sees an `align` directive in a record that is using field alignment, HLA will first align the current offset to the value specified by `align` and then align the next field s offset to the global record align value.

Nested record declarations may specify a different alignment value than the enclosing record, e.g.,

```
type
    alignedRecord4 : record[ 4]
        a:byte;
        b:byte;
        c:record[ 8]
            d:byte;
            e:byte;
        endrecord;
        f:byte;
        g:byte;
    endrecord;
```

In this example, HLA aligns fields `a, b, f,` and `g` on double word boundaries, it aligns `d` and `e` (within `c`) on eight-byte boundaries. Note that the alignment of the fields in the nested record is true only within that nested record. That is, if `c` turns out to be aligned on some boundary other than an eight-byte boundary, then `d` and `e` will not actually be on eight-byte boundaries; they will, however be on eight-byte boundaries relative to the start of `c`.

In addition to letting you specify a fixed alignment value, HLA also lets you specify a minimum and maximum alignment value for a record. The syntax for this is the following:

```
type
    recordname : record[ maximum : minimum]
        << fields >>
    endrecord;
```

Whenever you specify a maximum and minimum value as above, HLA will align all fields on a boundary that is at least the minimum alignment value. However, if the object s size is greater than the minimum value but less than or equal to the maximum value, then HLA will align that particular field on a boundary that is a multiple of the object s size. If the object s size is greater than the maximum size, then HLA will align the object on a boundary that is a multiple of the maximum size. As an example, consider the following record:

```
type
    r: record[  4:1 ];
        a:byte;             // offset 0
        b:word;             // offset 2
        c:byte;             // offset 4
        d:dword;[ 2]        // offset 8
        e:byte;             // offset 16
        f:byte;             // offset 17
        g:qword;            // offset 20
    endrecord;
```

Note that HLA aligns `g` on a double word boundary (not quad word, which would be offset 24) since the maximum alignment size is four. Note that since the minimum size is one, HLA allows the `f` field to be aligned on an odd boundary (since it s a byte).

If an array, record, or union field appears within a record, then HLA uses the size of an array element or the largest field of the record or union to determine the alignment size. That is, HLA will align the field without the outermost record on a boundary that is compatible with the size of the largest element of the nested array, union, or record.

HLAs sophisticated record alignment facilities let you specify record field alignments that match that used by most major high level language compilers. This lets you easily access data types used in those HLLs without resorting to inserting lots of `ALIGN` directives inside the record. We ll take a look at this feature in the next chapter when we discuss how to translate C structs into HLA records.

By default, the first field of a record is assigned offset zero within that record. If you would like to specify a different starting offset, you can use the following syntax for a record declaration:

```
type
    Pt3D:
        record := 4;
            x: int32;
            y: int32;
            z: int32;
        endrecord;
```

The constant expression specified after the assignment operator (:=) specifies the starting offset of the first field in the record. In this example `x, y,` and `z` will have the offsets 4, 8, and 12, respectively.

Warning: setting the starting offset in this manner does not add padding bytes to the record. This record is still a 12-byte object. If you declare variables using a record declared in this fashion, you may run into problems because the field offsets do not match the actual offsets in memory. This option is intended primarily for mapping records to pre-existing data structures in memory. Only really advanced assembly language programmers should use this option.

Note: Windows carefully defines the data fields in most of the record data structures so that you don t have to worry about field alignment. As long as you ensure that a record variable is sitting at a double-word aligned address in memory, you can be confident that the record s fields will be aligned at an appropriate address within that record.

2.2.3: Nested and Anonymous Unions and Records

The fields of a record or union may take on any legal data type. Such fields can be primitive types (as has been the case in the examples up to this point), arrays, or even other record and union types. Support for nestable (*recursive*) data types in an assembly language is very important to Win32 assembly programmers because many Win32 data structures employ this scheme.

Consider, for example, the following record type:

```
type
    recWrecAndUnion:
        record
            r1Field: byte;
            r2Field: word;
            r3Field: dword;
            rRecField:
                record
                    rr4Field: byte[ 2] ;
                    rr5Field: word[ 3] ;
                endrecord;
            r6Field: byte;
            rUnionField:
                union
                    r7Field:dword;
                    r8Field:real64;
                endunion;
        endrecord;
```

To access the nested `record` and `union` fields found within the `recWrecAndUnion` type, you use an extended form of the dot-notation . For example, if you have a variable, `r`, of type `recWrecAndUnion`, you can access the various fields of this variable using the following identifiers:

```
   r.r1Field
   r.r2Field
   r.r3Field
   r.rRecField.rr4Field
   r.rRecField.rr5Field
   r.r6Field
   r.rUnionField.r7Field
   r.rUnionField.r8Field
```

In a similar fashion, unions may contain records and other unions as fields, e.g.,

```
type
    unionWrecord:
        union
            u1Field: byte;
            u2Field: word;
            u3Field: dword;
            urField:
                record
                    u4Field: byte[ 2] ;
                    u5Field: word[ 3] ;
                endrecord;
            u6Field: byte;
        endunion;
```

Again, you would use the extended dot-notation syntax to access the fields of this `union`. Assuming that you have a variable `u` of type `unionWrecord`, you would access `u's` fields as follows:

```
   u.u1Field
   u.u2Field
   u.u3Field
   u.urField.u4Field
   u.urField.u5Field
   u.u6Field
```

Whenever a `record` appears within a `union`, the total size of that `record` is what HLA uses to determine the largest data object in the `union` for the purposes of determining the `union's` size. Also note that the fields of a `record` within a `union` all begin at separate offsets; it is the `record` object in the `union` that begins at the same offset as the other fields in the `union`. The fields of the `record`, however, begin at unique offsets within that `record`, just as for any `record`.

Unions also support a special field type known as an *anonymous record* . The following example demonstrates the syntax for an anonymous `record` in a `union`:

```
type
    unionWrecord:
        union
```

```
            u1Field: byte;
            u2Field: word;
            u3Field: dword;
            record
                u4Field: byte[ 2] ;
                u5Field: word[ 3] ;
            endrecord;
            u6Field: byte;
        endunion;
```

Fields appearing within the anonymous record do not necessarily start at offset zero in the data structure. In the example above, `u4Field` starts at offset zero while `u5Field` immediately follows it two bytes later. The fields in the `union` outside the anonymous `record` all start at offset zero. If the size of the anonymous record is larger than any other field in the `union`, then the record s size determines the size of the *union*. This is true for the example above, so the `union's` size is 16 bytes since the anonymous record consumes 16 bytes.

To access a field of an anonymous record in a `union`, you use the standard dot-notation syntax. Because the anonymous record does not have an explicit field name associated with the `record`, you access the fields of the `record` using a single-level dot notation, just like the other fields in the `union`. For example, if you have a variable named `uwr` of type `unionWrecord`, you d access the fields of `uwr` as follows:

```
   uwr.u1Field
   uwr.u2Field
   uwr.u3Field
   uwr.u4Field -- a field of the anonymous record
   uwr.u5Field -- a field of the anonymous record
   uwr.u6Field
```

You may also declare anonymous unions within a `record`. An anonymous union is a `union` declaration without a field name associated with the `union`, e.g.,

```
type
    DemoAU:
        record
            x: real32;
            union
                u1:int32;
                r1:real32;
            endunion;
            y:real32;
        endrecord;
```

In this example, `x, u1, r1,` and `y` are all fields of `DemoAU`. To access the fields of a variable `D` of type `DemoAU`, you would use the following names: `D.x, D.u1, D.r1,` and `D.y`. Note that `D.u1` and `D.r1` share the same memory locations at run-time, while `D.x` and `D.y` have unique addresses associated with them.

2.2.4: Pointer Types

HLA allows you to declare a pointer to some other type using syntax like the following:

```
pointer to base_type
```

The following example demonstrates how to create a pointer to a 32-bit integer within the type declaration section:

```
type pi32: pointer to int32;
```

HLA pointers are always 32-bit pointers.

HLA also allows you to define pointers to existing procedures using syntax like the following:

```
procedure someProc( parameter_list );
<< procedure options, followed by @external, @forward, or procedure body>>
      .
      .
      .
type
   p : pointer to procedure someProc;
```

The p procedure pointer inherits all the parameters and other procedure options associated with the original procedure. This is really just shorthand for the following:

```
procedure someProc( parameter_list );
<< procedure options, followed by @external, @forward, or procedure body>>
      .
      .
      .
type
   p : procedure ( Same_Parameters_as_someProc ); <<same options as someProc>>
```

The former version, however, is easier to maintain since you don t have to keep the parameter lists and procedure options in sync.

Note that HLA provides the reserved word null (or NULL, reserved words are case insensitive) to represent the nil pointer. HLA replaces NULL with the value zero. The NULL pointer is compatible with any pointer type (including strings, which are pointers).

2.2.5: Thunk Types

A thunk is an eight-byte variable that contains a pointer to a piece of code to execute and an execution environment pointer (i.e., a pointer to an activation record). The code associated with a thunk is, essentially, a small procedure that (generally) uses the activation record of the surround code rather than creating its own activation record. HLA uses thunks to implement the iterator `yield` statement as well as pass by name and pass by lazy evaluation parameters. In addition to these two uses of thunks, HLA allows you to declare your own thunk objects and use them for any purpose you desire. Windows also uses the term thunk but the Windows meaning is different than the HLA meaning. Fortunately, most Windows thunks involve calling 16-bit code in older versions of Windows. In modern Windows systems, you use thunks much less often than in older versions of Windows.

For more information about HLA thunks, please consult the *HLA Reference Manual*. Thunks are mentioned here only because of the possible confusion that might exist with Windows thunks. This book will not use HLA thunks, so there is little need to discuss this subject further.

2.2.6: Type Coercion

One HLA feature that many established assembly language programmers tend to have a problem with is the fact that HLA enforces a fair amount of type checking on its operands. Traditional assemblers generally treat variable names in a program as a memory address and nothing more; it is up to the programmer to choose an instruction that operates on the data type at that address. More modern assemblers, like MASM and TASM, enforce a small amount of type checking by storing the size of a memory object along with its address in the assembler s *symbol table* (a database where the assembler keeps track of information related to symbols during assembly). HLA takes this concept even farther. Although HLA is nowhere near as strongly typed as some high level languages (e.g., Ada), HLA does enforce type checking more strongly than most assemblers. While there are some well-established software engineering benefits associated with strong type checking, assembly programmers often need to access a object in a way that may be incompatible with its type. To achieve this in HLA, you use HLA s *type coercion operator*.

HLA s type coercion operator replaces the type of a memory or register operand. This syntax for this operator is

(type *newtype register_or_memory_operand*)

where *newtype* is a valid HLA type specification and *register_or_memory_operand* is a register name or a memory address. For register operands, the size of the new type must be compatible with the register s size; that is, if the register is an eight-bit register, then the type must be a one-byte type. This restriction does not apply to memory operands[3]. Here are some examples of HLA s type coercion operator:

```
(type byte uns8Var)
(type int32 eax) // Treat eax as a signed integer rather than unsigned (the default)
(type string esi) // esi is string object
(type recType [ ebx] ) // ebx is a pointer to a record of type recType
(type recType [ ebx] ).someField // Access field “someField” pointed at by EBX
(type procWith2Parms edx)(parm1, parm2); //Call procedure pointed at by edx
```

Note that there is a big difference between (type string esi) and (type string [esi]) . The string type is a pointer type, so (type string esi) tells HLA that ESI contains a string variable s value (that is, a pointer to character string data). The construct (type string [esi]) tells HLA that ESI points at a string variable (which is, itself, a pointer to character string data).

The important thing to note about HLA s type coercion operator is that you can treat everything inside the parentheses as though it were are single memory object (or register object) of that particular type. Therefore, you can use the dot-operator to reference fields of records, unions, and class, you can attach a parameter list to procedure pointer types, etc. Note, by the way, that there is a difference between the following two HLA statements:

```
(type procWithTwoParms edx)( parm1, parm2 );
(type procWithTwoParms [ edx] )( parm1, parm2 );
```

The first statement above calls the procedure whose address EDX contains. The second statement calls the procedure whose address is contained in the double word at which EDX points.

3. The restriction on registers exists because it doesn t make sense to cast a register as a size that is different than the register s actual size. Memory operands, on the other hand, may consume fewer or more bytes that the variable s original declaration because the 80x86 CPU can access addresses after the variable without any problems.

HLAs machine types (`byte, word, dword, qword, tbyte`, and `lword`) are generally type-free . The only type checking HLA does on operands of these types is to ensure that the value you store into one of these objects is the same size as the destination operand. The 80x86 s integer registers, for example, have the types byte, word, and dword associated with them, so they are compatible with most (ordinal/scalar) objects that are one, two, or four bytes long. HLAs `real32` type is a special case - HLA requires that you cast `real32` objects to `dword` if you want manipulate them as a 32-bit integer/ordinal object (e.g., in an integer register).

For those who complain about all the extra type involved with using HLAs type coercion operator, note that you can use an HLA TEXT constant or macro to replace some common type coercion with a single identifier, e.g.,

```
const
   recEBX :text := "(type recType [ ebx] )";
      .
      .
      .
   mov( recEBX.field, eax ); // copies (type recType [ ebx] ).field into EAX
```

As a general rule, you want to be careful about using HLAs type coercion operator. The overuse of the coercion operator is a sure sign of bad program design, particularly if you re constantly recasting memory variables (as opposed to constructs like [ebx] that truly require type casting). If you really need to access an object using one of several different types, you might consider using a `union` type rather than the type coercion operator. Using a `union` type is a little bit more structured (and less typing if you pick an appropriate union name) and helps document your intent a little better.

2.3: HLA High-Level Control Structures

An assumption that this book makes is that you are reasonably familiar with HLA and already a competent assembly/HLA programmer. Because HLAs high level control structures were originally intended as a device to help beginners learn assembly language programming via HLA, you might question why a chapter on advanced HLA usage would consider these statements. Indeed, because it s probably safe to assume that Win32 assembly programming is for advanced assembly language programmers, you might argue that a discussion of HLAs high-level control structures doesn t even belong in this book - advanced assembly language programmers already know the low-level way to do control structures.

A typical assembly programmer who has learned assembly language programming via HLA typically goes through three phases in their assembly education. In phase one (beginner), the programmer uses HLAs high-level control structures as a crutch to leverage their existing high level language programming knowledge in order to quickly learn assembly language. In phase two, the beginning assembly language programmer discovers that these statements aren t true assembly language and they learn the proper way to write control flow in assembly language using comparisons, tests, and conditional branches. Once an HLA programmer masters the low-level control flow and gains some experience using assembly language, that programmer enters a third phase where they gain a complete understanding of how HLA converts those high-level control structures into machine instructions and the programmer realizes that sometimes it s beneficial to go ahead and use those high level control statements in order to make their programs more readable.

The vast majority of Win32 assembly code written today is written with a high level assembler and the common convention is to prefer high level control structures over the low-level compare and branch sequences that low-level assemblers provide. That is to say, most Win32 assembly programmers have entered that third phase of their assembly programming careers where they begin to make intelligent choices concerning the use of high level control structures in order to make their programs more readable.

Windows assembly language programming is sufficiently complex that any tool we can employ to reduce the effort needed to write applications in assembly, that doesn t destroy the efficiency of the resulting application, is well worth using. Tools like HLAs high level control structures (and high level procedure definitions that we ll look at in the next section) provide a tremendous boost to assembly language productivity. So their use is worth serious consideration by Windows assembly programmers.

Long-time die-hard assembly programmers may find the use of anything high-level troublesome, even disgusting, in code that claims to be assembly language. There are two perceived problems with using high-level-like control structures in assembly code: inefficient compilation and inefficient paradigm.

The inefficient compilation issue concerns the fact that few assemblers (if any) provide statement-level optimization facilities such as those found in high-level language compilers (e.g., C). This means that high level compilers are theoretically capable of producing better code than a high level assembler because the high level language compiler will attempt to optimize the code it produces whereas no high level assembler does that (today, anyway). This isn t quite as bad as it seems, though, because 80x86 high level assemblers (including HLA) generally place sufficient restrictions on their high level control statements that enable them to generate fairly decent code. Still, do keep in mind that high level assemblers (and HLA in particular) don t optimize the code generated for high level control structures.

Of the two problems, the inefficient paradigm problem is probably the bigger problem of the two. Programmers who write assembly code using high level control structures often think in high level terms rather than in assembly terms (i.e., they write C code using mov statements). While this is a very real problem, this book will make the tacit assumption that a good assembly language programmer always considers the code a high-level control structure will generate and will avoid the use of such code if it leads to something that is overly inefficient.

The purpose of this section is to describe how HLA generates machine code for various high level control structures so that an assembly language programmer can make an educated decision concerning their use. While it s easy enough to get lazy and use a high level control structure where it s inappropriate, this book assumes that someone who cares about efficiency will not succumb to the temptation to overuse these control structures.

One other fact is worth pointing out concerning program efficiency and the need to optimize an assembly language sequence to squeeze out every last cycle - most Win32 programs spend the majority of their time waiting for program events. It is perfectly possible for a Win32 assembly programmer to make the application s code run ten times faster yet the user of that software won t perceive any difference in performance. This can happen because the program spends 95% of its time in the Windows kernel (or some other spot outside of the application) and all you achieve by speeding up the application code by a factor of ten is to shift 4.5% of the time that the application used to consume into Windows. The user will not notice a practical difference in the execution time of such a program improvement[4]. This is not to suggest that improved program efficiency isn t something to strive for; it simply means that a good software engineer carefully considers the need for optimization and weighs whether the benefits of such optimization are worth the cost (e.g., the production of less readable code by using low-level control structures in assembly language).

2.3.1: Boolean Control Expressions

A big reason why HLA generates reasonably good machine code for its high level control structures is the fact that HLA places several restrictions on the boolean expression you can use with these statements. In this section we ll review HLAs boolean expressions and then discuss how to write the most efficient forms of these expressions.

4. Obviously, this does not apply in every case, some applications are compute bound and consume an inordinate amount of CPU time within the application. Most Win32 applications, however, are not compute bound.

The primary limitation of HLAs `if` and other HLL statements has to do with the conditional expressions allowed in these statements. These expressions must take one of the following forms:

```
operand1 relop operand2

register in constant .. constant
register not in constant .. constant

memory in constant .. constant
memory not in constant .. constant

reg8 in CSet_Constant
reg8 in CSet_Variable

reg8 not in CSet_Constant
reg8 not in CSet_Variable

register
!register

memory
!memory

Flag

( boolean_expression )
!( boolean_expression )

boolean_expression && boolean_expression

boolean_expression || boolean_expression
```

For the first form, operand1 *relop* operand2 , *relop* is one of:

```
=  or ==        (either one, both are equivalent)
<> or !=        (either one)
<
<=
>
>=
```

`Operand1` and `operand2` must be operands that would be legal for a `cmp(operand1, operand2);` instruction. In fact, HLA simply translates these expressions into a `cmp` instruction a a conditional jump instruction (the exact conditional jump instruction is chosen by the operator and the type of operands). For the `if` statement, HLA emits a `cmp` instruction with the two operands specified and an appropriate conditional jump instruction that skips over the statements following the `then` reserved word if the condition is false. For example, consider the following code:

```
if( al = 'a' ) then

   stdout.put( "Option 'a' was selected", nl );

endif;
```

HLA translates this to:

```
cmp( al, 'a' );
jne skip_if;

   stdout.put( "Option 'a' was selected", nl );

skip_if:
```

HLA efficiently translates expressions of this form into machine code. As any decent assembly programmer knows, you ll get more efficient code if one (or both) of the operands are an 80x86 register. As the cmp instruction does not allow two memory operands, you may not compare one memory operand to another using the relational expression.

Unlike the conditional branch instructions, the six relational operators cannot differentiate between signed and unsigned comparisons (for example, HLA uses < for both signed and unsigned less than comparisons). Since HLA must emit different instructions for signed and unsigned comparisons, and the relational operators do not differentiate between the two, HLA must rely upon the types of the operands to determine which conditional jump instruction to emit.

By default, HLA emits unsigned conditional jump instructions (i.e., `ja, jae, jb, jbe`, etc.). If either (or both) operands are signed values, HLA will emit signed conditional jump instructions (i.e., `jg, jge, jl, jle`, etc.) instead. For example, suppose `sint32` is a signed integer variable and `uint32` is an unsigned 32-bit integer, consider the code that HLA will generate for the following expressions:

```
// sint32 < 5

   cmp( sint32, 5 );
   jnl skip;
      .
      .
      .
// uint32 < 5

   cmp( uint32, 5 );
   jnb skipu;
```

HLA considers the 80x86 registers to be *unsigned*. This can create some problems when using the HLA `if` statement. Consider the following code:

```
if( eax < 0 ) then

   << do something if eax is negative >>

endif;
```

Since neither operand is a signed value, HLA will emit the following code:

```
      cmp( eax, 0 );
      jnb SkipThenPart;
      << do something if eax is negative >>
SkipThenPart:
```

Unfortunately, it is never the case that the value in EAX is below zero (since zero is the minimum unsigned value), so the body of this `if` statement never executes. Clearly, the programmer intended to use a signed comparison here. The solution is to ensure that at least one operand is signed. However, as this example demonstrates, what happens when both operands are intrinsically unsigned?

The solution is to use coercion to tell HLA that one of the operands is a signed value. In general, it is always possible to coerce a register so that HLA treats it as a signed, rather than unsigned, value. The `if` statement above could be rewritten (correctly) as

```
if( (type int32 eax) < 0 ) then

   << do something if eax is negative >>

endif;
```

HLA will emit the `jnl` instruction (rather than `jnb`) in this example. Note that if either operand is signed, HLA will emit a signed condition jump instruction. Therefore, it is not necessary to coerce both unsigned operands in this example.

The second form of a conditional expression that the `if` statement accepts is a register or memory operand followed by in and then two constants separated by the .. operator, e.g.,

```
   if( al in 1..10 ) then ...
```

This code checks to see if the first operand is in the range specified by the two constants. The constant value to the left of the .. must be less than the constant to the right for this expression to make any sense. The result is true if the operand is within the specified range. For this instruction, HLA emits a pair of compare and conditional jump instructions to test the operand to see if it is in the specified range, e.g.,

```
   cmp( al, 1 );
   jb notInRange;
   cmp( al, 10 );
   ja notInRange;
```

Once again, HLA uses a signed or unsigned comparison based on the type of the memory or register operand present.

HLA also allows a exclusive range test specified by an expression of the form:

```
   if( al not in 1..10 ) then ...
```

In this case, the expression is true if the value in AL is outside the range 1..10. HLA generates code like the following for this expression:

```
   cmp( al, 1 );
   jb IsInRange;
   cmp( al, 10 );
   jb notInRange;
IsInRange:
```

In addition to integer ranges, HLA also lets you use the `in` operator with character set constants and variables. The generic form is one of the following:

```
reg_8 in CSetConst
reg_8 not in CSetConst
reg_8 in CSetVariable
reg_8 not in CSetVariable
```

For example, a statement of the form `if( al in {'a'..'z'}) then ...` checks to see if the character in the AL register is a lower case alphabetic character. Similarly,

```
    if( al not in {'a'..'z', 'A'..'Z'}) then...
```

checks to see if AL is not an alphabetic character. This form generates some particularly ugly code, so you want to be careful about using it. The code it generates looks something like the following:

```
push( eax );
movzx( al, eax );
bt( eax, compilerGeneratedCset );
pop( eax );
jnc notInSet;
```

On top of this code, HLA also generates a 16-bit character set constant object containing the members specified by the character set constant in the boolean expression.

The fifth form of a conditional expression that the `if` statement accepts is a single register name (eight, sixteen, or thirty-two bits). The `if` statement will test the specified register to see if it is zero (false) or non-zero (true) and branches accordingly. If you specify the not operator (!) before the register, HLA reverses the sense of this test.

The sixth form of a conditional expression that the `if` statement accepts is a single memory location. The type of the memory location must be boolean, byte, word, or dword. HLA will emit code that compares the specified memory location against zero (false) and generate an appropriate branch depending upon the value in the memory location. If you put the not operator (!) before the variable, HLA reverses the sense of the test.

The seventh form of a conditional expression that the `if` statement accepts is a Flags register bit or other condition code combination handled by the 80x86 conditional jump instructions. The following reserved words are acceptable as `if` statement expressions:

```
@c, @nc, @o, @no, @z, @nz, @s, @ns, @a, @na, @ae, @nae, @b, @nb, @be,
@nbe, @l, @nl, @g, @ne, @le, @nle, @ge, @nge, @e, @ne
```

These items emit an appropriate jump (of the opposite sense) around the `then` portion of the `if` statement if the condition is false.

If you supply any legal boolean expression in parentheses, HLA simply uses the value of the internal expression for the value of the whole expression. This allows you to override default precedence for the &&, ||, and ! operators.

The !(boolean_expression) evaluates the expression and does just the opposite. That is, if the interior expression is false, then !(boolean_expression) is true and vice versa. This is mainly useful with conjunction and disjunction since all of the other interesting terms already provide a logically negated form. Note that in general, the ! operator must precede some parentheses. You cannot say ! AX < BX , for example.

Originally, HLA did not include support for the conjunction (&&) and disjunction (||) operators. This was explicitly left out of the design so that beginning students would be forced to rethink their logical operations in assembly language. Unfortunately, it was so inconvenient not to have these operators that they were eventually added.

The conjunction and disjunction operators are the operators && and ||. They expect two valid HLA boolean expressions around the operator, e.g.,

```
   eax < 5 && ebx <> ecx
```

Since the above forms a valid boolean expression, it, too, may appear on either side of the && or || operator, e.g.,

```
   eax < 5 && ebx <> ecx || !dl
```

HLA gives && higher precedence than ||. Both operators are left-associative so if multiple operators appear within the same expression, they are evaluated from left to right if the operators have the same precedence. Note that you can use parentheses to override HLAs default precedence.

When generating code for these expressions, HLA employs *short-circuit evaluation*. Short-circuit evaluation means that HLA can generate code that uses control flow to maintain the current state (true or false) of the boolean expression rather than having to save the result somewhere (e.g., in a register). Because HLA promises not to disturb register values in its HLL-like code, short-circuit evaluation is the best way to proceed (it won t require pushing and popping registers as was necessary in the character set test earlier). For the current example, HLA would probably generate code like the following:

```
// eax < 5 && ebx <> ecx || !dl

   cmp( eax, 5 );
   jnb tryDL;
   cmp( ebx, ecx );
   jne isTrue;
tryDL:
   test( dl, dl );
   jnz isFalse;
isTrue:
```

One wrinkle with the addition of && and || is that you need to be careful when using the flags in a boolean expression. For example, eax < ecx && @nz hides the fact that HLA emits a compare instruction that affects the Z flag. Hence, the @nz adds nothing to this expression since EAX must not equal ECX if eax<ecx. So take care when using && and ||.

If you would prefer to use a less abstract scheme to evaluate boolean expressions, one that lets you see the low-level machine instructions, HLA provides a solution that allows you to write code to evaluate complex boolean expressions within the HLL statements using low-level instructions. Consider the following syntax:

```
if
(#{
   <<arbitrary HLA statements >>
}#) then

   << "True" section >>
```

```
else //or elseif...

   << "False" section >>

endif;
```

The #{ and }# brackets tell HLA that an arbitrary set of HLA statements will appear between the braces. HLA will *not* emit any code for the if expression. Instead, it is the programmer s responsibility to provide the appropriate test code within the #{---}# section. Within the sequence, HLA allows the use of the boolean constants true and false as targets of conditional jump instructions. Jumping to the true label transfers control to the true section (i.e., the code after the then reserved word). Jumping to the false label transfers control to the false section. Consider the following code that checks to see if the character in AL is in the range a .. z :

```
   if
   (#{
      cmp( al, 'a' );
      jb false;
      cmp( al, 'z' );
      ja false;
   }#) then

      << code to execute if AL in {'a'..'z'} goes here >>

   endif;
```

With the inclusion of the #{---}# operand, the if statement becomes much more powerful, allowing you to test any condition possible in assembly language. Of course, the #{---}# expression is legal in the elseif expression as well as the if expression.

2.3.2: The HLA IF..ENDIF Statement

HLA provides a limited if..then..elseif..else..endif statement that can help make your programs easier to read. For the most part, HLAs if statement provides a convenient substitute for a cmp and a conditional branch instruction pair (or chain of such instructions when employing elseif's).

The generic syntax for the HLA if statement is the following:

```
if( conditional_expression ) then

   << Statements to execute if expression is true >>

endif;


if( conditional_expression ) then

   << Statements to execute if expression is true >>

else

   << Statements to execute if expression is false >>
```

```
endif;

if( expr1 ) then

   << Statements to execute if expr1 is true >>

elseif( expr2 ) then

   << Statements to execute if expr1 is false
        and expr2 is true >>

endif;


if( expr1 ) then

   << Statements to execute if expr1 is true >>

elseif( expr2 ) then

   << Statements to execute if expr1 is false
        and expr2 is true >>

else

   << Statements to execute if both expr1 and
        expr2 are false >>

endif;
```

Note: HLAs if statement allows multiple `elseif` clauses. All `elseif` clauses must appear between `if` clause and the `else` clause (if present) or the `endif` (if an `else` clause is not present).

For simple boolean expressions (see the previous section) HLA generally emits code that is comparable to what an expert assembly language programmer would generate for these statements. There are, however, two issues to keep in mind when using HLAs high-level `if..elseif..else..endif` control structures: the cost of branching and inefficiencies associated with nested `if` statements.

High level control structures abstract away the low-level machine code implementation. Most of the time, this is a good thing. The whole purpose of the high level control structures is to hide what s going on underneath and make the code easier to read and understand. Sometimes, however, this abstraction can hide inefficiencies. Consider the following HLA `if` statement and its low-level implementation:

```
   if( eax < ebx ) then

      << do this if eax < ebx >>

   else

      << do this if eax >= ebx >>

   endif;

// Low-level equivalent:

   cmp( eax, ebx );
   jnb eaxGEebx;

      << do this if eax < ebx >>

      jmp ifIsDone;

eaxGEebx:
      << do this if eax >= ebx >>

ifIsDone:
```

This looks relatively straight-forward and simple. Probably the code that most people would have written if HLAs high-level `if` statement was not available. By carefully studying this code, you will note that there are two execution paths in this code, one path taken if the expression evaluates true and one path taken if the expression evaluates false. There is a transfer of control instruction that is taken along either path (the `jmp` instruction along the true path and the `jnb` instruction along the false path). Because jump instructions can execute slowly on modern CPU architectures, this particular code sequence may run slower than necessary because a jump is always taken.

An expert assembly programmer might consider the frequency with which the boolean expression in this `if` statement evaluates true or false. If the result of this boolean expression is random (that is, there is a 50/50 chance it will be true or false) then there is little you can do to improve the performance. However, in most if statements like this one, it s usually the case that one value occurs more frequently than the other (often, a boolean expression tends to evaluate true more often than false because most programmers tend to put the most common case in the true section of an `if..else..endif` statement). If that is the case, and the `if` statement is sitting in some section of time-critical code, an expert assembly programmer will often convert this `if` statement to something like the following:

```
   cmp( eax, ebx );
   jnb eaxGEebx;
      << do this if eax < ebx >>
returnHere:
      .
      .
      .
// Somewhere convenient:

eaxGEebx:
   << do this if eax >=ebx >>
   jmp returnHere;
```

By moving the `else` section to some other point in the code (presumably, after a control transfer instruction) this code provides a straight-through path (i.e., no `jmp`) for the true section of the code. The `else` section, on the other hand, now has to execute two jump instructions (the condition jump and the `jmp` at the end of the else sequence). However, if the `else` section executes less frequently than the `true` section of the `if` statement, this code sequence will run a little faster on many CPUs. One problem with the high-level `if..else..endif` statement is that it obscures the presence of that `jmp` instruction that transfers control over the `else` section. So you need to remember this fact if you re writing some highly time-critical code. Most of the time, this type of change (spaghetti programming) will not make a noticeable difference in the execution time of your program. Under the right circumstances, however, it will make a difference.

In general, any single instance of an inefficiency like these two is going to have a negligible impact on the performance or size of your programs. However, keep in mind that if you write assembly code using a high level language mindset, inefficiencies like this will begin to creep in all over the place and may begin to have an impact on the execution speed of your applications. Therefore, you should carefully consider each high level control statement you use in HLA to verify that HLA will be able to generate reasonable code for the sequence you re supplying.

2.3.3: The HLA WHILE..ENDWHILE Statement

The `while..endwhile` statement allows the following syntax:

```
while( boolean_expression ) do

   << while loop body>>

endwhile;


while( boolean_expression ) do

   << while loop body>>

welse

   << Code to execute when boolean_expression is false >>

endwhile;


while(#{ HLA_statements }#) do

   << while loop body>>

endwhile;

while(#{ HLA_statements }#) do

   << while loop body>>

welse

   << Code to execute when expression is false >>

endwhile;
```

The `while` statement allows the same boolean expressions as the HLA `if` statement. Like the HLA `if` statement, HLA allows you to use the boolean constants `true` and `false` as labels in the #{...}# form of the `while` statement above. Jumping to the `true` label executes the body of the `while` loop, jumping to the `false` label exits the while loop.

For the `while( expr ) do` forms, HLA moves the test for loop termination to the bottom of the loop and emits a jump at the top of the loop to transfer control to the termination test. For example, consider the following `while` statement:

```
while( eax < ebx ) do
```

```
      << statements to execute while eax < ebx >>

   endwhile;
```

Typical hand conversion of this `while` loop:

```
whlLabel:
      cmp( eax, ebx );
      jnb endWhileLabel;

         << statements to execute while eax < ebx >>

      jmp whlLabel;
endWhileLabel:
```

Here s the code that HLA actually emits for this `while` loop:

```
      jmp doWhile;
whlLabel:
         << statements to execute while eax < ebx >>

doWhile:
      cmp( eax, ebx );
      jb whlLabel;
```

The advantage of rotating the test to the bottom of the loop is that it eliminates a jump out of the body s path. That is, each iteration of this `while` loop executes one less `jmp` instruction than the typical conversion. As long as the loop s body executes one or more times (on the average), this conversion scheme is slightly more efficient. Note, however, that if the loop body doesn t execute the majority of the time, then the former conversion runs slightly faster.

For the `while(#{stmts}#)` form, HLA compiles the termination test at the top of the emitted code for the loop (i.e., the standard conversion). Therefore, the standard `while` loop may be slightly more efficient (in the typical case) than the hybrid form; just something to keep in mind.

The HLA `while` loop supports an optional `welse` (while-else) section. The `while` loop will execute the code in this section only when then the expression evaluates false. Note that if you exit the loop via a `break` or `breakif` statement the `welse` section does not execute. This provides logic that is sometimes useful when you want to do something different depending upon whether you exit the loop via the expression going false or by a `break` statement. Here s a quick example that demonstrates how HLA translates a `while..welse..endwhile` statement into low-level code:

```
   while( eax < ebx ) do

      << statements to execute while eax < ebx >>
      breakif( ecx = 0 );
      << more statements to execute >>

   welse

      << statements to execute when the boolean expression evaluates false >>

   endwhile;
```

```
// Typical HLA conversion to low-level code:

      jmp whlExpr;
whlLoop:
         << statements to execute while eax < ebx >>

         // breakif conversion:

         test( ecx, ecx );
         jz whileDone;

         << more statements to execute >>

whlExpr:
      cmp( eax, ebx );
      jb whlLoop;

// code for the welse section:

      << statements to execute when the boolean expression evaluates false >>

whileDone:
```

2.3.4: The HLA REPEAT..UNTIL Statement

HLA s `repeat..until` statement uses the following syntax:

```
repeat

   << statements to execute repeatedly >>

until( boolean_expression );


repeat

   << statements to execute repeatedly >>

until(#{  HLA_statements } #);
```

The body of the loop always executes at least once and the test for loop termination occurs at the bottom of the loop. The `repeat..until` loop (unlike C/C++ s `do..while` statement) terminates loop execution when the expression is true (that is, `repeat..until` repeats while the expression is false).

As you can see, the syntax for this is very similar to the `while` loop. About the only major difference is the fact that jump to the `true` label in the #{---}# sequence exits the loop while jumping to the `false` label in the #{---}# sequence transfers control back to the top of the loop. Also note that there is no equivalent to the `welse` clause in a `repeat..until` loop.

If you take away the jmp that HLA emits at the top of a (standard) `while` loop, you ll have a pretty good idea of the type of code that HLA generates for a `repeat..until` loop.

As a general rule, `repeat..until` loops are slightly more efficient than `while` loops because they don t execute that extra `jmp` instruction. Otherwise, the performance characteristics are virtually identical to that for the `while` loop.

2.3.5: The HLA FOR Loops

The HLA `for..endfor` statement is very similar to the C/C++ `for` loop. The `for` clause consists of three components:

```
for( initialize_stmt; boolean_expression; increment_statement ) do
```

The *initialize_statement* component is a single machine instruction. This instruction typically initializes a loop control variable. HLA emits this statement before the loop body so that it executes only once, before the test for loop termination.

The *boolean_expression* component is a simple boolean expression. This expression determines whether the loop body executes. Note that the `for` statement tests for loop termination before executing the body of the loop, just like the `while` statement (though the compiler will move this test to the bottom of the loop for efficiency reasons, the semantics are the same as though the test were physically at the beginning of the loop).

The *increment_statement* component is a single machine instruction that HLA emits at the bottom of the loop, just before jumping back to the top of the loop. This instruction is typically used to modify the loop control variable.

The syntax for the HLA `for` statement is the following:

```
for( initStmt; BoolExpr; incStmt ) do

   << loop body >>

endfor;

-or-

for( initStmt; BoolExpr; incStmt ) do

   << loop body >>

felse

   << statements to execute when BoolExpr evaluates false >>

endfor;
```

Semantically, this statement is identical to the following `while` loop:

```
initStmt;
while( BoolExpr ) do
   << loop body >>
   incStmt;
endwhile;

-or-
```

```
initStmt;
while( BoolExpr ) do
   << loop body >>
   incStmt;

welse

   << statements to execute when BoolExpr evaluates false >>
endwhile;
```

Note that HLA does not include a form of the `for` loop that lets you bury a sequence of statements inside the boolean expression. Use the `while` loop if you want to do that. If this is inconvenient, you can always create your own version of the `for` loop using HLAs macro facilities.

The `felse` section in the `for..felse..endfor` loop executes when the boolean expression evaluates false. Note that the `felse` section does not execute if you break out of the `for` loop with a `break` or `breakif` statement. You can use this fact to do different logic depending on whether the code exits the loop via the boolean expression going false or via some sort of `break`.

The code that HLA generates for the `for` loop is identical to the code that HLA generates for the converted `while` example. Therefore, efficiency concerns about the `for` loop are identical to those for the `while` loop.

HLAs `forever..endfor` loop creates an infinite loop[5]. The `endfor` efficiently compiles into a single `jmp` instruction that transfers control to the top of the loop (i.e., to the `forever` clause). Typically, you d use a `break` or `breakif` statement to exit a `forever..endfor` at some point in the middle of the loop. This makes `forever..endfor` loops slightly less efficient that a standard `while..endwhile` or `repeat..until` loop because you ll always execute two jump instructions in such a loop - one that breaks out of the loop and one at the bottom of the loop that transfers control back to the top of the loop. For example, consider the following:

```
   forever
      << Code to execute before the test >>
      breakif( eax = ecx );
      << Code to execute after the test >>
   endfor;

// conversion to “pure” assembly language

   forLabel:
      << Code to execute before the test >>
      cmp( eax, ecx );
      jne exitFor;
      << Code to execute after the test >>
   jmp forLabel;
   exitFor:
```

An experienced assembly language programmer would probably rewrite this as a loop that tests for loop termination at the bottom of the loop using code like the following:

```
// conversion to slightly more efficient assembly language

   jmp IntoLoop
```

5. Though you may use HLAs `break, breakif, exit,` or `exitif` statements to escape from such a loop.

```
   forLabel:
      << Code to execute after the test >>
   IntoLoop:
      << Code to execute before the test >>
      cmp( eax, ecx );
      je forLabel;
```

HLA provides a third `for` loop, the `foreach` loop, that lets you create user-defined loops via HLA s `iterator` facility. We re not going to discuss the code generation for the `foreach` loop here, but be aware that HLA generates some fairly sophisticated code for the `foreach` loop and the corresponding `iterator`. Though the code is not particularly inefficient, you should note that using a `foreach` loop where a simple `for` or `while` loop will suffice is always going to be less efficient. Please see the HLA documentation for more details concerning the use of iterators and the `foreach` statement.

2.3.6: HLA’s BREAK, CONTINUE, and EXIT Statements

The `break` and `breakif` statements allow you to exit a loop at some point other than the normal test for loop termination. These two statements allow the following syntax:

```
break;
breakif( boolean_expression );
breakif(#{  stmts } #);
```

The `continue` and `continueif` statements allow you to restart a loop. These two statements allow the following syntax:

```
continue;
continueif( boolean_expression );
continueif(#{  stmts } #);
```

Restarting a loop means jumping to the loop termination test for `while, repeat..until`, and `for` loops, it means jumping to the (physical) top of the loop for `forever` and `foreach` loops.

Note that the `break`, `breakif, continue,` and `continueif` statements are legal only inside `while, for, forever, foreach`, and `repeat` loops. HLA does not recognize loops you ve coded yourself using discrete assembly language instructions (of course, you can probably write a macro to provide a `break` function for your own loops). Note that the `foreach` loop pushes data on the stack that the `break` statement is unaware of. Therefore, if you break out of a `foreach` loop, garbage will be left on the stack. The HLA `break` statement will issue a warning if this occurs. It is your responsibility to clean up the stack upon exiting a `foreach` loop if you break out of it. Using a continue within a `foreach` loop is perfectly fine.

The `begin..end` statement block provides a structured goto statement for HLA. The `begin` and `end` clauses surround a group of statements; the `exit` and `exitif` statements allow you to exit such a block of statements in much the same way that the `break` and `breakif` statements allow you to exit a loop. Unlike `break` and `breakif`, which can only exit the loop that immediately contains the `break` or `breakif`, the exit statements allow you to specify a `begin` label so you can exit several nested contexts at once. The syntax for the `begin..end, exit`, and `exitif` statements is as follows:

```
begin contextLabel ;

   << statements within the specified context >>
```

```
end contextLabel;

exit contextLabel;
exitif( boolean_expression ) contextLabel;
exitif(#{ stmts }#) contextLabel;
```

The `begin..end` clauses do not generate any machine code (although `end` does emit a label to the assembly output file). The `exit` statement simply emits a `jmp` to the first instruction following the `end` clause. The `exitif` statement emits a compare and a conditional jump to the statement following the specified end.

If you break out of a `foreach` loop using the `exit` or `exitif` statements, there will be garbage left on the stack. It is your responsibility to be aware of this situation (i.e., HLA doesn t warn you about it) and clean up the stack, if necessary.

You can nest `begin..end` blocks and `exit` out of any enclosing `begin..end` block at any time. The `begin` label provides this capability. Consider the following example:

```
program ContextDemo;

#include( "stdio.hhf" );

static
   i:int32;

begin ContextDemo;

   stdout.put( "Enter an integer:" );
   stdin.get( i );

   begin c1;

      begin c2;

         stdout.put( "Inside c2" nl );
         exitif( i < 0 ) c1;

      end c2;
      stdout.put( "Inside c1" nl );
      exitif( i = 0 ) c1;
      stdout.put( "Still inside c1" nl );

   end c1;
   stdout.put( "Outside of c1" nl );

end ContextDemo;
```

The `exit` and `exitif` statements let you exit any `begin..end` block; including those associated with a program unit such as a procedure, iterator, method, or even the main program. Consider the following (unusable) program:

```
      program mainPgm;

         procedure LexLevel1;
```

```
        procedure LexLevel2;
        begin LexLevel2;

            exit LexLevel2;   // Returns from this procedure.
            exit LexLevel1;   // Returns from this procedure and
                              //  and the LexLevel1 procedure
                              //  (including cleaning up the stack).
            exit mainPgm;     // Terminates the main program.

        end LexLevel2;

    begin LexLevel1;
        .
        .
        .
    end LexLevel1;

begin mainPgm;
    .
    .
    .
end mainPgm;
```

Note: You may only exit from procedures that have a display and all nested procedures from the procedure you wish to exit from through to the `exit` statement itself must have displays. In the example above, both `LexLevel1` and `LexLevel2` must have displays if you wish to exit from the `LexLevel1` procedure from inside `LexLevel2`. By default, HLA emits code to build the display unless you use the “`@nodisplay`” procedure option.

Note that to exit from the current procedure, you must not have specified the “`@noframe`” procedure option. This applies only to the current procedure. You may exit from nesting (lower lex level) procedures as long as the display has been built. For more information on displays and stack frames, see the upcoming section on procedures and procedure invocations.

2.3.7: Exception Handling Statements in HLA

HLAs exception handling system is an interesting example for those seeking to write efficient code. While using HLAs try..endtry and raise statements is quite a bit more efficient than manually checking for problems in your code, it s also the case that these statements generate a fair number of instructions and their overuse can lead to the creation of some inefficient code. Therefore, it s important to understand the code that HLA generates for these statements so you can write code efficiently by using these statements properly.

HLA uses the `try..exception..endtry` and `raise` statements to implement exception handling. The syntax for these statements is as follows:

```
try
   << HLA Statements to execute >>

<<  unprotected // Optional unprotected section.
   << HLA Statements to execute >>
>>

exception( const1 )

   << Statements to execute if exception const1 is raised >>
<< optional exception statements for other exceptions >>

<<  anyexception //Optional anyexception section.
   << HLA Statements to execute >>
>>

endtry;


raise( const2 );
```

Const1 and *const2* must be unsigned integer constants. Usually, these are values defined in the *excepts.hhf* header file. Some examples of predefined values include the following:

```
ex.StringOverflow
ex.StringIndexError

ex.ValueOutOfRange
ex.IllegalChar
ex.ConversionError

ex.BadFileHandle
ex.FileOpenFailure
ex.FileCloseError
ex.FileWriteError
ex.FileReadError
ex.DiskFullError
ex.EndOfFile

ex.MemoryAllocationFailure

ex.AttemptToDerefNULL

ex.WidthTooBig
ex.TooManyCmdLnParms

ex.ArrayShapeViolation
ex.ArrayBounds

ex.InvalidDate
ex.InvalidDateFormat
ex.TimeOverflow
ex.AssertionFailed
ex.ExecutedAbstract
```

Windows Structured Exception Handler exception values:

```
ex.AccessViolation
ex.Breakpoint
ex.SingleStep

ex.PrivInstr
ex.IllegalInstr
ex.BoundInstr
ex.IntoInstr

ex.DivideError

ex.fDenormal
ex.fDivByZero
ex.fInexactResult
ex.fInvalidOperation
ex.fOverflow
ex.fStackCheck
ex.fUnderflow

ex.InvalidHandle
ex.StackOverflow

ex.ControlC
```

This list is constantly changing as the HLA Standard Library grows, so it is impossible to provide a compete list of standard exceptions here. Please see the *excepts.hhf* header file for a complete list of standard exceptions.

The HLA Standard Library currently reserves exception numbers zero through 1023 for its own internal use. User-defined exceptions should use an integer value greater than or equal to 1024 and less than or equal to 65535 ($FFFF). Exception value $10000 and above are reserved for use by Windows Structured Exception Handler and Linux signals.

The `try..endtry` statement contains two or more blocks of statements. The statements to protect immediately follow the `try` reserved word. During the execution of the protected statements, if the program encounters the first exception block, control immediately transfers to the first statement following the endtry reserved word. The program will skip all the statements in the exception blocks.

If an exception occurs during the execution of the protected block, control is immediate transferred to an exception handling block that begins with the exception reserved word and the constant that specifies the type of exception.

Example:

```
repeat

   mov( false, GoodInput );
   try
      stdout.put( "Enter an integer value:" );
      stdin.get( i );
      mov( true, GoodInput );

   exception( ex.ValueOutOfRange )

      stdout.put( "Numeric overflow, please reenter ", nl );

   exception( ex.ConversionError )

      stdout.put( "Conversion error, please reenter", nl );

   endtry;

until( GoodInput = true );
```

In this code, the program will repeatedly request the input of an integer value as long as the user enters a value that is out of range (+/- 2 billion) or as long as the user enters a value containing illegal characters.

Multiple try..endtry statements can be *nested*. If an exception occurs within a nested try protected block, the exception blocks in the innermost try block containing the offending statement get first shot at the exceptions. If none of the exception blocks in the enclosing try..endtry statement handle the specified exception, then the next innermost try..endtry block gets a crack at the exception. This process continues until some exception block handles the exception or there are no more try..endtry statements.

If an exception goes unhandled, the HLA run-time system will handle it by printing an appropriate error message and aborting the program. Generally, this consists of printing Unhandled Exception (or a similar message) and stopping the program. If you include the excepts.hhf header file in your main program, then HLA will automatically link in a somewhat better default exception handler that will print the number (and name, if known) of the exception before stopping the program.

Note that try..endtry blocks are dynamically nested, not statically nested. That is, a program must actually execute the try in order to activate the exception handler. You should never jump into the middle of a protected block, skipping over the try. Doing so may produce unpredictable results.

You should not use the try..endtry statement as a general control structure. For example, it will probably occur to someone that one could easily create a switch/case selection statement using try..endtry as follows:

```
try
   raise( SomeValue );

   exception( case1_const)
      <code for case 1>

   exception( case2_const)
      <code for case 2>

   etc.
endtry
```

While this might work in some situations, there are two problems with this code.

First, if an exception occurs while using the try..endtry statement as a switch statement, the results may be unpredictable. Second, HLAs run-time system assumes that exceptions are rare events. Therefore, the code generated for the exception handlers doesn t have to be efficient. You will get much better results implementing a switch/case statement using a table lookup and indirect jump (see the Art of Assembly) rather than a try..endtry block.

Warning: As you ll see in a moment, the try statement pushes data onto the stack upon initial entry and pops data off the stack upon leaving the try..endtry block. Therefore, jumping into or out of a try..endtry block is an absolute no-no. As explained so far, then, there are only two reasonable ways to exit a try statement, by falling off the end of the protected block or by an exception (handled by the try statement or a surrounding try statement).

The unprotected clause in the try..endtry statement provides a safe way to exit a try..endtry block without raising an exception or executing all the statements in the protected portion of the try..endtry statement. An unprotected section is a sequence of statements, between the protected block and the first exception handler, that begins with the keyword unprotected. E.g.,

```
try
```

```
    << Protected HLA Statements >>

  unprotected

    << Unprotected HLA Statements >>

  exception( SomeExceptionID )

    << etc. >>

endtry;
```

control flows from the protected block directly into the unprotected block as though the `unprotected` keyword were not present. However, between the two blocks HLA compiler-generated code removes the data pushed on the stack. Therefore, it is safe to transfer control to some spot outside the `try..endtry` statement from within the unprotected section.

If an exception occurs in an unprotected section, the `try..endtry` statement containing that section does not handle the exception. Instead, control transfers to the (dynamically) nesting `try..endtry` statement (or to the HLA run-time system if there is no enclosing `try..endtry`).

If you re wondering why the `unprotected` section is necessary (after all, why not simply put the statements in the `unprotected` section after the `endtry`?), just keep in mind that both the protected sequence and the handled exceptions continue execution after the `endtry`. There may be some operations you want to perform after exceptions are released, but only if the protected block finished successfully. The `unprotected` section provides this capability. Perhaps the most common use of the `unprotected` section is to break out of a loop that repeats a `try..endtry` block until it executes without an exception occurring. The following code demonstrates this use:

```
forever

   try

         stdout.put( "Enter an integer: " );
         stdin.geti8();   // May raise an exception.

      unprotected

         break;

      exception( ex.ValueOutOfRange )

         stdout.put( "Value was out of range, reenter" nl );

      exception( ex.ConversionError )

         stdout.put( "Value contained illegal chars" nl );

   endtry;

endfor;
```

This simple example repeatedly asks the user to input an int8 integer until the value is legal and within the range of valid integers.

Another clause in the try..except statement is the anyexception clause. If this clause is present, it must be the last clause in the try..except statement, e.g.,

```
try
   << protected statements >>

<<
   unprotected

      Optional unprotected statements
>>

<< exception( constant ) // Note: may be zero or more of
                            of these.

      Optional exception handler statements
>>

   anyexception
      << Exception handler if none of the others execute >>

endtry;
```

Without the anyexception clause present, if the program raises an exception that is not specifically handled by one of the exception clauses, control transfers to the enclosing try..endtry statement. The anyexception clause gives a try..endtry statement the opportunity to handle any exception, even those that are not explicitly listed. Upon entry into the anyexception block, the EAX register contains the actual exception number.

The HLA raise statement generates an exception. The single parameter is an 8, 16, or 32-bit ordinal constant. Control is (ultimately) transferred to the first (most deeply nested) try..endtry statement that has a corresponding exception handler (including anyexception).

If the program executes the raise statement within the protected block of a try..endtry statement, then the enclosing try..endtry gets first shot at handling the exception. If the raise statement occurs in an unprotected block, or in an exception handler (including anyexception), then the next higher level (nesting) try..endtry statement will handle the exception. This allows *cascading* exceptions; that is, exceptions that the system handles in two or more exception handlers. Consider the following example:

```
try
   << Protected statements >>

  exception( someException )
   << Code to process this exception >>

   // The following re-raises this exception, allowing
   // an enclosing try..endtry statement to handle
   // this exception as well as this handler.

   raise( someException );

  << Additional, optional, exception handlers >>

endtry;
```

To understand the cost of using HLAs exception handling statements, it s wise to take a look at the code the compiler generates for these statements. Consider the following HLA code and it s conversion to pure assembly language:

```
   try
      << Protected Code >>

   exception( ex.ValueOutOfRange )
      << Code to execute if system raises “out of range” exception >>

   exception( ex.StringIndexError )
      << Code to execute if there is a string index exception >>

   endtry;

// “Pure” assembly code HLA produces for the above statements:

   push( &exceptionLabel );
   push( ebp );
   mov( ExceptionPtr__hla_, ebp ); // This is a global symbol HLA defines
   push( &HWexcept__hla_ );        //  "   "   " "  "   "  "   "   "    "
   mov( esp, ExceptionPtr__hla_ );

      << Protected Code >>

   mov( ExceptionPtr__hla_, esp ); // Code that cleans up the stack after
   pop( ExceptionPtr__hla_ );      //  executing the protected code.
   add( 8, esp );
   pop( ebp );
   add( 4, esp );
   jmp endTryLabel;

exceptionLabel:
   cmp( eax, 3 ); // ex.ValueOutOfRange = 3
   jne tryStrIndexEx
      << Code to execute if system raises “out of range” exception >>
   jmp endTryLabel;

tryStrIndexEx:
   cmp( eax, 2 ); // ex.StringIndexError = 2
   jne Raise__hla_; // Re-raises exception in surrounding try..endtry block.

endTryLabel:
```

As you can see by reading through this code, HLA emits five instructions upon encountering the `try` clause and emits six instructions after the protected block (i.e., before the first exception statement). Assuming exceptions don t occur very frequently (that is, they are the exceptions rather than the rule), most of the other statements that HLA emits will not execute frequently. These 11 statements before and after the protected code, however, execute everything the program encounters and leaves a `try..endtry` block. While 11 statements isn t a horrendous number, if you ve only got a few statements within the `try..endtry` statement and you execute this sequence within a loop, this 11 statements can have an impact on your program s performance.

If the performance of the `try..endtry` statement is an issue, and you would still like to protect a sequence of statements with `try..endtry`, you can reduce the overhead by placing more statements inside the `try..endtry` protected region. For example, if you ve got a `try..endtry` block inside a loop you ll have to execute that `try..endtry` sequence on each iteration of the loop. However, if you place these statements around the loop, you only have to execute the 11 statements that the `try..endtry` block generates once for each execution of the loop. The drawback to widening the scope of the `try..endtry` statement is that it becomes more difficult to pinpoint the cause of the exception to just a few statements. For example, if you re reading a set of values into an array within a loop and you place the try..endtry block around the loop, you may not be able to determine which iteration of the loop triggered the exception (note that you cannot count on the registers containing reasonable values whenever someone raises an exception, so if your loop index is in a register, it s lost).

The big problem with exceptions is that they shouldn t occur under normal circumstances; therefore, the execution of those 11 statements is pure overhead, most of the time. Therefore, you want to be judicious with your use of the `try..endtry` block, especially in nested loops and other code sequences that execute frequently. It goes without saying that you should not use exception handling as a normal flow control sequence (that is, you shouldn t attempt to use the `try..endtry` block as a fancy form of a `switch/case` statement). The 11 statement overhead applies to code sequences that execute normally, without raising any exceptions. Processing an exception can require several dozen (or more) instructions. Normally, the overhead associated with processing an actual exception isn t an issue in most applications because exceptions rarely occur. However, if you attempt to use exceptions as a normal execution path, you ll probably be disappointed with the performance (note that the code that executes when an exception occurs is part of the HLA run-time system, it doesn t appear in the code that HLA emits for the `try..endtry` block shown earlier).

2.4: HLA Procedure Declarations and Invocations

HLA provides a very sophisticated system for declaring and calling procedures using a high level syntax. Indeed, HLA s procedure syntax, in many ways, is higher-level than most high level programming languages. We aren t going to get into all the gory details here because, quite frankly, all these features aren t necessary for writing Win32 applications (because C doesn t support these features). Nevertheless, HLA does have many useful features you won t normally find in an assembly language that help make writing Win32 assembly code much easier. We ll discuss those features in this section.

The general syntax for an HLA procedure declaration is:

```
procedure identifier ( optional_parameter_list );  procedure_options
   declarations
begin identifier;
   statements
end identifier;
```

The optional parameter list consists of a list of var-type declarations taking the form:

```
   optional_access_keyword  identifier1 : identifier2 optional_in_reg
```

optional_access_keyword, if present, must be `val, var, valres, result, name,` or `lazy` and defines the parameter passing mechanism (pass by value, pass by reference, pass by value/result [or value/returned], pass by result, pass by name, or pass by lazy evaluation, respectively). The default is pass by value (`val`) if an access keyword is not present. For pass by value parameters, HLA allocates the specified number of bytes according to the size of that object in the activation record. For pass by reference, pass by value/result, and pass by result, HLA allocates four bytes to hold a pointer to the object. For pass by name and pass by lazy evaluation, HLA allocates eight bytes to hold a pointer to the associated thunk and a pointer to the thunk s execution environment. Because Win32 applications typically only use pass by value and pass by reference, those are the two parameter passing mechanisms we ll discuss in this book. For details on the other HLA parameter passing mechanisms, please consult the *HLA Reference Manual*.

The *optional_in_reg* clause, if present, corresponds to the phrase in *reg* where *reg* is one of the 80x86 s general purpose 8-, 16-, or 32-bit registers. This clause tells HLA to pass the parameter in the specified register rather than on the stack.

HLA also allows a special parameter of the form:

```
var identifier : var
```

This creates an *untyped* reference parameter. You may specify any memory variable as the corresponding actual parameter and HLA will compute the address of that object and pass it on to the procedure without further type checking. Within the procedure, the parameter is given the `dword` type.

The *procedure_options* component above is a list of keywords that specify how HLA emits code for the procedure. There are several different procedure options available: `@noalignstack`, `@alignstack`, `@pascal`, `@stdcall`, `@cdecl`, `@align(` *`int_const`*`)`, `@use reg32`, `@leave`, `@noleave`, `@enter`, `@noenter`, and `@returns(` *`"text"`* `)`. See Table 2-2 for a description of each of these procedure options.

Table 2-2: Procedure Options in an HLA Program

`@noframe`, `@frame`	By default, HLA emits code at the beginning of the procedure to construct a stack frame. The `@noframe` option disables this action. The `@frame` option tells HLA to emit code for a particular procedure if stack frame generation is off by default. HLA also uses these two special identifiers as a compile-time variable to set the default display generation for all procedures. Setting `@frame` to true (or `@noframe` to false) turns on frame generation by default; setting `@frame` to false (or `@noframe` to true) turns off frame generation.
`@nodisplay`, `@display`	By default, HLA emits code at the beginning of the procedure to construct a display within the frame. The `@nodisplay` option disables this action. The `@display` option tells HLA to emit code to generate a display for a particular procedure if display generation is off by default. Note that HLA does not emit code to construct the display if `'@noframe'` is in effect, though it will assume that the programmer will construct this display themselves. HLA also uses these two special identifiers as a compile-time variable to set the default display generation for all procedures. Setting `@display` to true (or `@nodisplay` to false) turns on display generation by default; setting `@display` to false (or `@nodisplay` to true) turns off display generation. Because most Win32 applications do not typically use nested procedures (and, in particular, none of the examples in this book use them), you'll usually see the following statement near the top of most example programs appearing in this book: `?nodisplay := true;`

@noalignstack, @alignstack	By default (assuming frame generation is active), HLA will an instruction to align ESP on a four-byte boundary after allocating local variables. Win32, Linux, and other 32-bit OSes require the stack to be dword-aligned (hence this option). If you know the stack will be dword-aligned, you can eliminate this extra instruction by specifying the `@noalignstack` option. Conversely, you can force the generation of this instruction by specifying the `@alignstack` procedure option. HLA also uses these two special identifiers as a compile-time variable to set the default display generation for all procedures. Setting `@alignstack` to true (or `@noalignstack` to false) turns on stack alignment generation by default; setting `@alignstack` to false (or `@noalignstack` to true) turns off stack alignment code generation. Because Windows guarantees that the stack will be aligned when it transfers control to a procedure in your application, and because programs in this book never force a misalignment of the stack, you'll commonly see the following statement near the beginning of the HLA example programs in this book: `?@noalignstack := true;`
@pascal, @cdecl, @stdcall	These options give you the ability to specify the parameter passing mechanism for the procedure. By default, HLA uses the `@pascal` calling sequence. This calling sequence pushes the parameters on the stack in a left-to-right order (i.e., in the order they appear in the parameter list). The `@cdecl` procedure option tells HLA to pass the parameters from right-to-left so that the first parameter appears at the lowest address in memory and that it is the user's responsibility to remove the parameters from the stack. The `@stdcall` procedure option is a hybrid of the `@pascal` and `@cdecl` calling conventions. It pushes the parameters in the right-to-left order (just like `@cdecl`) but `@stdcall` procedures automatically remove their parameter data from the stack (just like `@pascal`). Win32 API calls use the `@stdcall` calling convention. In the Win32 examples appearing in this book, you'll see the following conventions adhered to: ¥ All Win32 API functions use the `@stdcall` calling convention (there are some Windows API functions that require the `@cdecl` calling convention, but this book does not call those routines). ¥ All functions in the application that Windows calls (the window procedure and any callback functions) use the `@stdcall` calling convention. ¥ Local procedures in the program (those that only HLA code calls) use the `@pascal` calling convention.
@align(*int_constant*)	The `@align(` *`int_const`* `)` procedure option aligns the procedure on a 1, 2, 4, 8, or 16 byte boundary. Specify the boundary you desire as the parameter to this option. The default is `@align(1)`, which is unaligned; HLA also uses this special identifier as a compile-time variable to set the default procedure alignment . Setting `@align := 1` turns off procedure alignment while supplying some other value (which must be a power of two) sets the default procedure alignment to the specified number of bytes.

`@use reg32`	When passing parameters, HLA can sometimes generate better code if it has a 32-bit general purpose register for use as a scratchpad register. By default, HLA never modifies the value of a register behind your back; so it will often generate less than optimal code when passing certain parameters on the stack. By using the `@use` procedure option, you can specify one of the following 32-bit registers for use by HLA: EAX, EBX, ECX, EDX, ESI, or EDI. By providing one of these registers, HLA may be able to generate significantly better code when passing certain parameters. Note that it is not legal for you to specify the EBP or ESP registers after an `@use` option. Procedures only require a single 32-bit register, so if you specify multiple `@use` options, HLA only uses the last register you specify. Although HLA allows the use of ESI and EDI, HLA generates better code in certain circumstances if you specify one of EAX, EBX, ECX or EDX.
`@returns(` *"text"* `)`	This option specifies the compile-time return value whenever a function name appears as an instruction operand. For example, suppose you are writing a function that returns its result in EAX. You should probably specify a "returns" value of "EAX" so you can compose that procedure just like any other HLA machine instruction. For more details on this option, check out the discussion of "instruction composition" in the *HLA Reference Manual*.

`@leave, @noleave`	These two options control the code generation for the standard exit sequence. If you specify the `@leave` option then HLA emits the x86 `leave` instruction to clean up the activation record before the procedure returns. If you specify the `@noleave` option, then HLA emits the primitive instructions to achieve this, e.g.,
	mov(ebp, esp);
	pop(ebp);
	
	The manual sequence is faster on some architectures, the `leave` instruction is always shorter.
	
	Note that `@noleave` occurs by default if you've specified `@noframe`. By default, HLA assumes `@noleave` but you may change the default using these special identifiers as a compile-time variable to set the default `leave` generation for all procedures. Setting `@leave` to true (or `@noleave` to false) turns on `leave` generation by default; setting `@leave` to false (or `@noleave` to true) turns off the use of the `leave` instruction.

`@enter, @noenter`	These two options control the code generation for a procedure's standard entry sequence. If you specify the `@enter` option then HLA emits the x86 `enter` instruction to create the activation record. If you specify the `@noenter` option, then HLA emits the primitive instructions to achieve this. The manual sequence is always faster, using the `enter` instruction is usually shorter. Note that `@noenter` occurs by default if you've specified `@noframe`. By default, HLA assumes `@noenter` but you may change the default using these special identifiers as a compile-time variable to set the default `enter` generation for all procedures. Setting `@enter` to true (or `@noenter` to false) turns on `enter` generation by default; setting `@enter` to false (or `@noenter` to true) turns off the use of the `enter` instruction. The enter instruction is primarily of interest to programmers using nested procedures and displays (see the @nodisplay option). Because most Win32 applications don't use nested procedures and a display, you'll rarely see this option used in a Win32 program.

The `@returns` option and instruction composition in HLA deserve a special mention. Instruction composition is a feature in HLA that allows you to substitute an HLA instruction (or other statement) in place of certain instruction operands. For example, consider the following perfectly legal HLA statement:

```
   mov( mov( 0, eax ), ebx );
```

To process a statement like this, HLA associates a compile-time return value with every instruction. The return value is a string (text) value and is usually the destination operand of the instruction. For example, the destination operand of the nested instruction in this example is eax so HLA substitutes EAX for the nested instruction after emitting the code for that instruction. In other words, this particular sequence is equivalent to the following:

```
   mov( 0, eax );
   mov( eax, ebx );  // First operand was the result of instruction composition.
```

You may also supply procedure invocations as instruction operands. When HLA encounters a procedure invocation in an instruction operand, it emits the code for that procedure call and then substitutes the returns string for that procedure invocation (the returns string is the string operand you supply in the `@returns` procedure option). The following example demonstrates how the `@returns` option works:

```
program returnsDemo;
#include( "stdio.hhf" );

   procedure eax0; @returns( "eax" );
   begin eax0;

      mov( 0, eax );

   end eax0;

begin returnsDemo;

   mov( eax0(), ebx );
   stdout.put( "ebx=", ebx, nl );

end returnsDemo;
```

If you do not explicitly supply an `@returns` procedure option, then HLA will use the empty string as the `@returns` value for that procedure. Obviously, if you compose such a procedure invocation inside some other instruction you ll probably create a syntax error.

You ll rarely see code that uses procedure invocations in an instruction as in these examples. However, instruction composition can take place in other areas too. For example, the operands of a boolean expression in an HLA `if` statement also support instruction composition, allowing you to specify statements like the following:

```
   if( eax0() <> 0 ) then
      .
      .
      .
   endif;
```

HLA will emit the code for the call to the procedure (eax0 in this example) and then substitute the procedure s @returns value for the procedure invocation in the boolean expression (i.e., eax in this example).

To help those who insist on constructing the activation record themselves, HLA declares two local constants within each procedure: _vars_ and _parms_. The _vars_ symbol is an integer constant that specifies the number of local variables declared in the procedure. This constant is useful when allocating storage for your local variables. The _parms_ constants specifies the number of bytes of parameters. You would normally supply this constant as the parameter to a ret() instruction to automatically clean up the procedure s parameters when it returns. Here s an example of these two constants in use:

```
procedure hasParmsAndLocals( p1:dword; p2:int32; p3:uns32 ); @noframe; @nodisplay;
var
   localA :dword;
   localB :dword;
   localC :dword;
   localD :dword;
begin hasParmsAndLocals;

   push( ebp ); // we have to build the activation record ourselves!
   mov( esp, ebp );
   sub( _vars_, esp ); // make room for local variables (16 bytes, as it turns out).
      .
      .
      .
   mov( ebp, esp );  // Clean up the activation record on return
   pop( ebp );       // Note that we could also use "leave" here.
   ret( _parms_ );   // Return and remove parameters from the stack (12 bytes).
```

If you do not specify @nodisplay, then HLA defines a run-time variable named _display_ that is an array of pointers to activation records. As Win32 applications generally do not use nested procedures and displays, we will not discuss the use of the @nodisplay option (or displays) in this book. For more details on displays and the @nodisplay (or @display) procedure option, please consult the *HLA Reference Manual*.

You can also declare @external procedures (procedures defined in other HLA units or written in languages other than HLA) using the following syntaxes:

```
procedure externProc1 (optional parameters) ; @returns( "text" ); @external;

procedure externProc2 (optional parameters) ;
   @returns( "text" ); @external( "different_name" );
```

As with normal procedure declarations, the parameter list and @returns clause are optional.

The first form is generally used for HLA-written functions. HLA will use the procedure s name (externProc1 in this case) as external name.

The second form lets you refer to the procedure by one name (externProc2 in this case) within your HLA program and by a different name (*different_name* in this example) in the MASM generated code. This second form has two main uses: (1) if you choose an external procedure name that just happens to be a MASM reserved word, the program may compile correctly but fail to assemble. Changing the external name to something else solves this problem. (2) When calling procedures written in external languages you may need to specify characters that are not legal in HLA identifiers. For example, Win32 API calls often use names like WriteFile@24 containing illegal (in HLA) identifier symbols. The string operand to the external option lets you specify any name you choose. Of course, it is your responsibility to see to it that you use identifiers that are compatible with the linker and MASM, HLA doesn t check these names.

By default, HLA does the following:

- ¥ Creates a display for every procedure.
- ¥ Emits code to construct the stack frame for each procedure.
- ¥ Emits code to align ESP on a four-byte boundary upon procedure entry.
- ¥ HLA assumes that it cannot modify any register values when passing (non-register) parameters.
- ¥ The first instruction of the procedure is unaligned.

These options are the most general and safest for beginning assembly language programmers. However, the code HLA generates for this general case may not be as compact or as fast as is possible in a specific case. For example, few procedures will actually need a display data structure built upon procedure activation. Therefore, the code that HLA emits to build the display can reduce the efficiency of the program. Most Win32 application programmers, of course, will want to use procedure options like `@nodisplay` to tell HLA to skip the generation of this code. However, if a program contains many procedures and none of them need a display, continually adding the `@nodisplay` option can get really old. Therefore, HLA allows you to treat these directives as pseudo-compile-time-variables to control the default code generation. E.g.,

```
? @display := true; // Turns on default display generation.
? @display := false; // Turns off default display generation.
? @nodisplay := true; // Turns off default display generation.
? @nodisplay := false; // Turns on default display generation.

? @frame := true; // Turns on default frame generation.
? @frame := false; // Turns off default frame generation.
? @noframe := true; // Turns off default frame generation.
? @noframe := false; // Turns on default frame generation.

? @alignstack := true; // Turns on default stk alignment code generation.
? @alignstack := false; // Turns off default stk alignment code generation.
? @noalignstack := true; // Turns off default stk alignment code generation.
? @noalignstack := false; // Turns on default stk alignment code generation.

? @enter := true; // Turns on default ENTER code generation.
? @enter := false; // Turns off default ENTER code generation.
? @noenter := true; // Turns off default ENTER code generation.
? @noenter := false; // Turns on default ENTER code generation.

? @leave := true; // Turns on default LEAVE code generation.
? @leave := false; // Turns off default LEAVE code generation.
? @noleave := true; // Turns off default LEAVE code generation.
? @noleave := false; // Turns on default LEAVE code generation.

?@align := 1; // Turns off procedure alignment (align on byte boundary).
?@align := int_expr; // Sets alignment, must be a power of two.
```

These directives may appear anywhere in the source file. They set the internal HLA default values and all procedure declarations following one of these assignments (up to the next, corresponding assignment) use the specified code generation option(s). Note that you can override these defaults by using the corresponding procedure options mentioned earlier.

2.4.1: Disabling HLA's Automatic Code Generation for Procedures

Before jumping in and describing how to use the high level HLA features for procedures, the best place to start is with a discussion of how to disable these features and write plain old fashioned assembly language code. This discussion is important because procedures are the one place where HLA automatically generates a lot of code for you and many assembly language programmers prefer to control their own destinies; they don t want the compiler to generate any excess code for them. Though most Win32 assembly language programmers use HLAs high level procedure features, many may want to disable some or all of HLAs code generation features for procedures. So disabling HLAs automatic code generation capabilities is a good place to start this discussion.

By default, HLA automatically emits code at the beginning of each procedure to do five things: (1) Preserve the pointer to the previous activation record (EBP); (2) build a display in the current activation record; (3) allocate storage for local variables; (4) load EBP with the base address of the current activation record; (5) adjust the stack pointer (downwards) so that it points at a dword-aligned address.

When you return from a procedure, by default HLA will deallocate the local storage and return, removing any parameters from the stack.

To understand the code that HLA emits, consider the following simple procedure:

```
procedure p( j:int32 );
var
    i:int32;
begin p;
end p;
```

Here is a dump of the symbol table that HLA creates for procedure p[6]:

```
p   <0,proc>:Procedure type (ID=?1_p)
    --------------------------------
    _vars_          <1,cons>:uns32, (4 bytes)  =4
    i               <1,var >:int32, (4 bytes, ofs:-12)
    _parms_         <1,cons>:uns32, (4 bytes)  =4
    _display_       <1,var >:dword, (8 bytes, ofs:-4)
    j               <1,valp>:int32, (4 bytes, ofs:8)
    p               <1,proc>:
------------------------------------
```

The important thing to note here is that local variable `i` is at offset -12 and HLA automatically created an eight-bit local variable named `_display_` which is at offset -4.

HLA emits code similar to the following for the procedure above[7]:

```
procedure p; @nodisplay; @noframe; @nostackalign;
begin p;
      push( ebp );        //Dynamic link (pointer to previous activation record)
      pushd( [ebp-4] );   //Display for lex level 0
      lea( ebp, [esp+04] );  //Get frame ptr (point EBP at current activation record)
```

6. Note that the symbol table output routines in HLA change from time to time, so this output may not exactly match what HLA produces in the particular version that you re using.
7. Again, code generation is subject to change in HLA, so this may not exactly match what the particular version of HLA that you re using produces.

```
      push( ebp );           //Ptr to this proc's A.R. (part of display construction)
      sub( 4, esp );         //Local storage.
      and( $fffffffc, esp ); //dword-align stack

// Exit point for the procedure:

      mov( ebp, esp );  //Deallocate local variables.
      pop( ebp );       //Restore pointer to previous activation record.
      ret( 4 );         //Return, popping parameters from the stack.
end p;
```

Building the display data structure is not very common in standard assembly language programs. This is only necessary if you are using nested procedures and those nested procedures need to access non-local variables. Because this is a rare situation, many programmers will immediately want to tell HLA to stop emitting the code to generate the display. This is easily accomplished by adding the `@nodisplay` procedure option to the procedure declaration. Adding this option to procedure `p` produces the following:

```
procedure p( j:int32 ); @nodisplay;
var
    i:int32;
begin p;
end p;
```

Compiling this procedures the following symbol table dump:

```
p                <0,proc>:Procedure type (ID=?1_p)
    --------------------------------
    _vars_          <1,cons>:uns32, (4 bytes)  =4
    i               <1,var >:int32, (4 bytes, ofs:-4)
    _parms_         <1,cons>:uns32, (4 bytes)  =4
    j               <1,valp>:int32, (4 bytes, ofs:8)
    p               <1,proc>:
------------------------------------
```

Note that the `_display_` variable is gone and the local variable `i` is now at offset -4. Here is the code that HLA emits for this new version of the procedure:

```
procedure p; @nodisplay; @noframe; @nostackalign;
begin p;
      push( ebp );           //Dynamic link (pointer to previous activation record)
      mov( esp, ebp );       //Point EBP at current activation record.
      sub( 4, esp );         //Local storage.
      and( $fffffffc, esp ); //dword-align stack

// Exit point for the procedure:

      mov( ebp, esp );  //Deallocate local variables.
      pop( ebp );       //Restore pointer to previous activation record.
      ret( 4 );         //Return, popping parameters from the stack.
end p;
```

As you can see, this code is smaller and a bit less complex. Unlike the code that built the display, it is fairly common for an assembly language programmer to construct an activation record in a manner similar to this.

Indeed, about the only instruction out of the ordinary in this example is the `and` instruction that dword-aligns the stack (Win32 calls require the stack to be double word aligned, and the system performance is much better if the stack is double word aligned).

This code is still relatively inefficient if you don t pass parameters on the stack and you don t use automatic (non-static, local) variables. Many assembly language programmers pass their few parameters in machine registers and also maintain local values in the registers. If this is the case, then the code above is pure overhead. You can inform HLA that you wish to take full responsibility for the entry and exit code by using the `"@noframe"` procedure option. Consider the following version of `p`:

```
procedure p( j:int32 ); @nodisplay; @noframe;
var
    i:int32;
begin p;
end p;
```

(this produces the same symbol table dump as the previous example).

HLA emits the following code for this version of `p`:

```
procedure p; @nodisplay; @noframe; @nostackalign;
begin p;
end p;
```

Whoa! There s nothing there! But this is exactly what the advanced assembly language programmer wants. With both the `@nodisplay` and `@noframe` options, HLA does not emit any extra code for you. You would have to write this code yourself.

By the way, you *can* specify the `@noframe` option without specifying the `@nodisplay` option. HLA still generates no extra code, but it will assume that you are allocating storage for the display in the code you write. That is, there will be an eight-byte `_display_` variable created and `i` will have an offset of -12 in the activation record. It will be your responsibility to deal with this. Although this situation is possible, it s doubtful this combination will be used much at all.

Note a major difference between the two versions of `p` when `@noframe` is not specified and `@noframe` is specified: if `@noframe` is not present, HLA automatically emits code to return from the procedure. This code executes if control falls through to the `"end p;"` statement at the end of the procedure. Therefore, if you specify the `@noframe` option, you must ensure that the last statement in the procedure is a `ret()` instruction or some other instruction that causes an unconditional transfer of control. If you do not do this, then control will fall through to the beginning of the next procedure in memory, probably with disastrous results.

The `ret()` instruction presents a special problem. It is dangerous to use this instruction to return from a procedure that does not have the `@noframe` option. Remember, HLA has emitted code that pushes a lot of data onto the stack. If you return from such a procedure without first removing this data from the stack, your program will probably crash. The correct way to return from a procedure without the `@noframe` option is to jump to the bottom of the procedure and run off the end of it. Rather than require you to explicitly put a label into your program and jump to this label, HLA provides the `exit` *`procname`*`;` instruction. HLA compiles the `exit` instruction into a `jmp` that transfers control to the clean-up code HLA emits at the bottom of the procedure. Consider the following modification of `p` and the resulting assembly code produced:

```
procedure p( j:int32 ); @nodisplay;
var
    i:int32;
begin p;
    exit p;
    nop();
end p;


procedure p( j:int32 ); @noframe; @nodisplay;
var
    i:int32;
begin p;
    push( ebp )
    mov( esp, ebp )l
    sub( 4, esp );
    and( $fffffffc, esp );
    jmp     _x_2_p;          // This is what the exit statement compiles to
    nop;
_x_2_p:
    mov( ebp, esp );
    pop( ebp );
    ret( 4 );
end p;
```

As you can see, HLA automatically emits a label to the assembly output file (“`_x_2_p`” in this instance) at the bottom of the procedure where the clean-up code starts. HLA translates the “`exit p;`” instruction into a `jmp` to this label.

If you look back at the code emitted for the version of `p` with the `@noframe` option, you ll note that HLA did not emit a label at the bottom of the procedure. Therefore, HLA cannot generate a jump to this nonexistent label, so you cannot use the `exit` statement in a procedure with the `@noframe` option (HLA will generate an error if you attempt this).

Of course, HLA will *not* stop you from putting a `ret()` instruction into a procedure without the `@noframe` option (some people who know exactly what they are doing might actually want to do this). Keep in mind, if you decide to do this, that you must deallocate the local variables (that s what the “`mov esp, ebp`” instruction is doing), you need to restore EBP (via the “`pop ebp`” instruction above), and you need to deallocate any parameters pushed on the stack (the “`ret 4`” handles this in the example above). The following code demonstrates this:

```
procedure p( j:int32 ); @nodisplay;
var
    i:int32;
begin p;

    if( j = 0 ) then

        // Deallocate locals.

        mov( ebp, esp );

        // Restore old EBP

        pop( ebp );
        // Return and pop parameters

        ret( 4 );

    endif;
    nop();
end p;


procedure p; @nodisplay; @noframe;
begin p;
   push( ebp );
   mov( esp, ebp );
   sub( 4, esp );
   and( $fffffffc, esp );

   cmp( (type dword [ebp+8]), 0 );
   jne _2_false;
   mov( ebp, esp );
   pop( ebp );
   ret( 4 );

_2_false:
   nop;
   mov( ebp, esp );
   pop( ebp );
   ret( 4 );
end p;
```

If real assembly language programmers would generally specify both the `@noframe` and `@nodisplay` options, why not make them the default case (and use "`@frame`" and "`@display`" options to specify the generation of the activation record and display)? Well, keep in mind that HLA was originally designed as a tool to teach assembly language programming to beginning students and this default behavior is safer for beginning programmers. Also, most Win32 applications use these defaults (that is, most Win32 application programmers do not specify @noframe, regardless of how advanced they are) so these defaults make a lot of sense for Win32 applications.

If you are absolutely certain that your stack pointer is aligned on a four-byte boundary upon entry into a procedure, you can tell HLA to skip emitting the `and( $ffff_fffc, esp );` instruction by specifying the `@noalignstack` procedure option. Because Win32 apps always keep the stack dword aligned, most Win32 applications set the stack alignment to off via a statement like this one near the beginning of the source file:

```
   ?@nostackalign := true;
```

Note that specifying `@noframe` also specifies `@noalignstack`.

Note that HLA does provide a few additional procedure options. Please consult the *HLA Reference Manual* for more details on these options. They are not of immediate interest to us here, so this book will ignore them.

2.4.2 Calling HLA Procedures

There are two standard ways to call an HLA procedure: use the call instruction or simply specify the name of the procedure as an HLA statement. Both mechanisms have their pluses and minuses.

To call an HLA procedure using the `call` instruction is exceedingly easy. Simply use either of the following syntaxes:

```
call( procName );
call procName;
```

Either form compiles into an 80x86 `call` instruction that calls the specified procedure. The difference between the two is that the first form (with the parentheses) returns the procedure s returns value, so this form can appear as an operand to another instruction. The second form above always returns the empty string, so it is not suitable as an operand of another instruction. Also, note that the second form requires a statement or procedure label, you may not use memory addressing modes in this form; on the other hand, the second form is the only form that lets you call a statement label (as opposed to a procedure label); this form is useful on occasion.

If you use the `call` statement to call a procedure, then you are responsible for passing any parameters to that procedure. In particular, if the parameters are passed on the stack, you are responsible for pushing those parameters (in the correct order) onto the stack before the call. This is a lot more work than letting HLA push the parameters for you, but in certain cases you can write more efficient code by pushing the parameters yourself.

The second way to call an HLA procedure is to simply specify the procedure name and a list of actual parameters (if needed) for the call. This method has the advantage of being easy and convenient at the expense of a possible slight loss in efficiency and flexibility. This calling method should also prove familiar to most HLL programmers. As an example, consider the following HLA program:

```
program parameterDemo;

#include( "stdio.hhf" );

procedure PrtAplusB( a:int32; b:int32 ); @nodisplay;
begin PrtAplusB;

    mov( a, eax );
    add( b, eax );
    stdout.put( "a+b=", (type int32 eax ), nl );

end PrtAplusB;

static
    v1:int32 := 25;
    v2:int32 := 5;

begin parameterDemo;

    PrtAplusB( 1, 2 );
    PrtAplusB( -7, 12 );
    PrtAplusB( v1, v2 );

    mov( -77, eax );
    mov( 55, ebx );
    PrtAplusB( eax, ebx );

end parameterDemo;
```

This program produces the following output:

```
a+b=3
a+b=5
a+b=30
a+b=-22
```

As you can see, the calls to `PrtAplusB` in HLA are very similar to calling procedures (and passing parameters) in a high level language like C/C++ or Pascal. There are, however, some key differences between and HLA call and a HLL procedure call. The next section will cover those differences in greater detail. The important thing to note here is that if you choose to call a procedure using the HLL syntax (that is, the second method described here), you can pass the parameters in the parameter list and let HLA push the parameters for you. If you want to take complete control over the parameter passing code, one way to do this is to use the `call` instruction.

2.4.3: Parameter Passing in HLA, Value Parameters

The previous section probably gave you the impression that passing parameters to a procedure in HLA is nearly identical to passing those same parameters to a procedure in a high level language. The truth is, the examples in the previous section were rigged. There are actually many restrictions on how you can pass parameters to an HLA procedure. This section discusses the parameter passing mechanism in detail.

The most important restriction on actual parameters in a call to an HLA procedure is that HLA only allows memory variables, registers, constants, and certain other special items as parameters. In particular, you cannot specify an arithmetic expression that requires computation at run-time (although a constant expression, computable at compile time is okay). The bottom line is this: if you need to pass the value of an expression to a procedure, you must compute that value prior to calling the procedure and pass the result of the computation; HLA will not automatically generate the code to compute that expression for you.

The second point to mention here is that HLA is a strongly typed language when it comes to passing parameters. This means that with only a few exceptions, the type of the actual parameter must exactly match the type of the formal parameter. If the actual parameter is an int8 object, the formal parameter had better not be an int32 object or HLA will generate an error. The only exceptions to this rule are the `byte, word,` and `dword` types. If a formal parameter is of type `byte`, the corresponding actual parameter may be any one-byte data object. If a formal parameter is a `word` object, the corresponding actual parameter can be any two-byte object. Likewise, if a formal parameter is a `dword` object, the actual parameter can be any four-byte data type. Conversely, if the actual parameter is a `byte, word,` or `dword` object, it can be passed without error to any one, two, or four-byte actual parameter (respectively). Programmers who are really lazy make all their parameters `bytes, words`, or `dwords` (at least, wherever possible). Programmers who care about the quality of their code use untyped parameters cautiously.

If you want to use the high level calling sequence, but you don t like the inefficient code HLA sometimes produces when generating code to pass your parameters, you can always use the #{...}# sequence parameter to override HLAs code generation and substitute your own code for one or two parameters. Of course, it doesn t make any sense to pass all the parameters is a procedure using this trick, it would be far easier just to use the call instruction. Here s an example of using HLAs manual parameter passing facility within a HLL style procedure call:

```
   PrtAplusB
   (
      #{
         mov( i, eax );   // First parameter is "i+5"
         add( 5, eax );
         push( eax );
      } #,
      5
   );
```

HLA will automatically copy an actual value parameter into local storage for the procedure, regardless of the size of the parameter. If your value parameter is a one million byte array, HLA will allocate storage for 1,000,000 bytes and then copy that array in on each call. C/C++ programmers may expect HLA to automatically pass arrays by reference (as C/C++ does) but this is not the case. If you want your parameters passed by reference, you must explicitly state this.

The code HLA generates to copy value parameters, while not particularly bad, certainly isn t optimal. If you need the fastest possible code when passing parameters by value on the stack, it would be better if you explicitly pushed the data yourself. Another alternative that sometimes helps is to use the use reg_{32} procedure option to provide HLA with a hint about a 32-bit scratchpad register that it can use when building parameters on the stack. For more details on this facility, please see the *HLA Reference Manual* or *The Art of Assembly Language*.

2.4.4: Parameter Passing in HLA: Reference Parameters

The one good thing about pass by reference parameters is that they are always four byte pointers, regardless of the size of the actual parameter. Therefore, HLA has an easier time generating code for these parameters than it does generating code for pass by value parameters.

Like high level languages, HLA places a whopper of a restriction on pass by reference parameters: they can only be memory locations. Constants and registers are not allowed since you cannot compute their address. Do keep in mind, however, that any valid memory addressing mode is a valid candidate to be passed by reference; you do not have to limit yourself to static and local variables. For example, “[eax]” is a perfectly valid memory location, so you can pass this by reference (assuming you type-cast it, of course, to match the type of the formal parameter). The following example demonstrate a simple procedure with a pass by reference parameter:

```
program refDemo;

#include( "stdio.hhf" );

    procedure refParm( var a:int32 );
    begin refParm;

        mov( a, eax );
        mov( 12345, (type int32 [ eax] ));

    end refParm;

    static
        i:int32:=5;

begin refDemo;

    stdout.put( "(1) i=", i, nl );
    mov( 25, i );
    stdout.put( "(2) i=", i, nl );
    refParm( i );
    stdout.put( "(3) i=", i, nl );


end refDemo;
```

The output produced by this code is

```
(1) i=5
(2) i=25
(3) i=12345
```

As you can see, the parameter a in `refParm` exhibits pass by reference semantics since the change to the value `a` in `refParm` changed the value of the actual parameter (`i`) in the main program.

Note that HLA passes the address of `i` to `refParm`, therefore, the `a` parameter contains the address of `i`. When accessing the value of the `i` parameter, the `refParm` procedure must deference the pointer passed in `a"`. The two instructions in the body of the `refParm` procedure accomplish this.

Take a look at the code that HLA generates for the call to `refParm`:

```
         pushd( &i );
         call refParm;
```

As you can see, this program simply computed the address of `i` and pushed it onto the stack. Now consider the following modification to the main program:

```
program refDemo;

#include( "stdio.hhf" );

    procedure refParm( var a:int32 );
    begin refParm;

        mov( a, eax );
        mov( 12345, (type int32 [eax]));

    end refParm;

    static
        i:int32:=5;

    var
        j:int32;

begin refDemo;

    mov( 0, j );
    refParm( j );
    refParm( i );
    lea( eax, j );
    refParm( [eax] );
```

```
end refDemo;
```

This version emits the following code:

```
   mov( 0, (type dword [ ebp-8]) );
   push( eax );
   lea( eax, (type dword [ ebp-8]));
   xchg( eax, [ esp] );
   call refParm;

   pushd( &i );
   call refParm;

   lea( eax, (type dword [ ebp-8]));
   push( eax );
   push( eax );
   lea( eax, (type dword [ eax+0]));
   mov( eax, [ esp+4] );
   pop( eax );
   call refParm;
```

As you can see, the code emitted for the last call is pretty ugly (we could easily get rid of three of the instructions in this code). This call would be a good candidate for using the `call` instruction directly. Also see Hybrid Parameters a little later in this book. Another option is to use the "`@use` *`reg32`*" option to tell HLA it can use one of the 32-bit registers as a scratchpad. Consider the following:

```
procedure refParm( var a:int32 ); use eax;
      .
      .
      .
    lea( ebx, j );
    refParm( [ ebx] );
```

This sequence generates the following code (which is a little better than the previous example):

```
   lea( ebx, j );
   lea( eax, (type dword [ ebx+0]));
   push( eax );
   call refParm;
```

As a general rule, the type of an actual reference parameter must exactly match the type of the formal parameter. There are a couple exceptions to this rule. First, if the formal parameter is `dword`, then HLA will allow you to pass any four-byte data type as an actual parameter by reference to this procedure. Second, you can pass an actual `dword` parameter by reference if the formal parameter is a four-byte data type. There are some other exceptions that we ll explore in a later section of this chapter.

2.4.5: Untyped Reference Parameters

HLA provides a special formal parameter syntax that tells HLA that you want to pass an object by reference and you don t care what its type is. Consider the following HLA procedure:

```
procedure zeroObject( var object:byte; size:uns32 );
begin zeroObject;
   << code to write "size" zeros to "object" >
end zeroObject;
```

The problem with this procedure is that you will have to coerce non-byte parameters to a byte before passing them to `zeroObject`. That is, unless you re passing a byte parameter, you ve always got to call `zeroObject` thusly:

```
   zeroObject( (type byte NotAByte), sizeToZero );
```

For some functions you call frequently with different types of data, this can get painful very quickly.

The HLA untyped reference parameter syntax solves this problem. Consider the following declaration of `zeroObject`:

```
procedure zeroObject( var object:var; size:uns32 );
begin zeroObject;
   << code to write "size" zeros to "object" >
end zeroObject;
```

Notice the use of the reserved word `var` instead of a data type for the object parameter. This syntax tells HLA that you re passing an arbitrary variable by reference. Now you can call `zeroObject` and pass any (memory) object as the first parameter and HLA won t complain about the type, e.g.,

```
   zeroObject( NotAByte, sizeToZero );
```

Note that you may only pass untyped objects by reference to a procedure.

Note, and this is very important, when you pass an actual parameter to a procedure that has an untyped reference parameter in that slot, HLA will always take the address of the object you pass it. This is true even if the actual parameter is a pointer to some data (which is in direct contrast to what HLA normally does with pointer objects that you pass as reference parameters). For more details, read the section on Passing Pointers and Values as Reference Parameters on page 92.

2.4.6: Hybrid Parameter Passing in HLA

HLA s automatic code generation for parameters specified using the high level language syntax isn t always optimal. In fact, sometimes it is downright inefficient. This is because HLA makes very few assumptions about your program. For example, suppose you are passing a word parameter to a procedure by value. Since all parameters in HLA consume an even multiple of four bytes on the stack, HLA will zero extend the word and push it onto the stack. It does this using code like the following:

```
   pushw( 0 );
   pushw( Parameter );
```

Clearly you can do better than this if you know something about the variable. For example, if you know that the two bytes following `Parameter` are in memory (as opposed to being in the next page of memory that isn t allocated, and access to such memory would cause a protection fault), you could get by with the single instruction:

```
        push( (type dword Parameter) );
```

Unfortunately, HLA cannot make these kinds of assumptions about the data because doing so could create malfunctioning code (admittedly, in very rare circumstances).

One solution, of course, is to forego the HLA high level language syntax for procedure calls and manually push all the parameters yourself and call the procedure via the `call` instruction. However, this is a major pain that involves lots of extra typing and produces code that is difficult to read and understand. Therefore, HLA provides a hybrid parameter passing mechanism that lets you continue to use a high level language calling syntax yet still specify the exact instructions needed to pass certain parameters. This hybrid scheme works out well because HLA actually does a good job with most parameters (e.g., if they are an even multiple of four bytes, HLA generates efficient code to pass the parameters; it s only those parameters that have a weird size that HLA generates less than optimal code for).

If a parameter consists of the `#{` and `}#` brackets with some corresponding code inside the brackets, HLA will emit the code inside the brackets in place of any code it would normally generate for that parameter. So if you wanted to pass a 16-bit parameter efficiently to a procedure named `Proc` and you re sure there is no problem accessing the two bytes beyond this parameter, you could use code like the following:

```
        Proc( #{ push( (type dword WordVar) ); }# );
```

Whenever you pass a non-static[8] variable as a parameter by reference, HLA generates the following code to pass the address of that variable to the procedure:

```
push( eax );
push( eax );
lea( eax, Variable );
mov( eax, [esp+4] );
pop( eax );
```

It generates this particular code to ensure that it doesn t change any register values (after all, you could be passing some other parameter in the EAX register). If you have a free register available, you can generate slightly better code using a calling sequence like the following (assuming EBX is free):

```
HasRefParm
(
   #{
         lea( ebx, Variable );
         push( ebx );
   }#
);
```

2.4.7: Parameter Passing in HLA, Register Parameters

HLA provides a special syntax that lets you specify that certain parameters are to be passed in registers rather than on the stack. The following are some examples of procedure declarations that use this feature:

8. Static variables are those you declare in the static, readonly, and storage sections. Non-static variables include parameters, VAR objects, and anonymous memory locations.

```
procedure a( u:uns32 in eax ); forward;
procedure b( w:word in bx ); forward;
procedure d( c:char in ch ); forward;
```

Whenever you call one of these procedures, the code that HLA automatically emits for the call will load the actual parameter value into the specified register rather than pushing this value onto the stack. You may specify any general purpose 8-bit, 16-bit, or 32-bit register after the `in` keyword following the parameter s type. Obviously, the parameter must fit in the specified register. You may only pass reference parameters in 32-bit registers; you cannot pass parameters that are not one, two, or four bytes long in a register. Also note that HLA does not support passing parameters in FPU, MMX, or XMM registers (at least, using the high level syntax).

You can get in to trouble if you re not careful when using register parameters, consider the following two procedure definitions:

```
procedure one( u:uns32 in eax; v:dword in ebx ); forward;
procedure two( a:uns32 in eax );
begin two;

   one( 25, a );

end two;
```

The call to `one` in procedure `two` looks like it passes the values 25 and whatever was passed in for `a` in procedure `two`. However, if you study the HLA output code, you will discover that the call to `one` passes 25 for both parameters. They reason for this is because HLA emits the code to load 25 into EAX in order to pass 25 in the `u` parameter. Unfortunately, this wipes out the value passed into `two` in the `a` variable, hence the problem. Be aware of this if you use register parameters often.

2.4.8: Passing Pointers and Values as Reference Parameters

When it comes to pass by reference parameters, HLA relaxes the type checking rules a little bit in order to make calling procedures with reference parameters a bit more convenient. For the most part, this relaxation (which you ll see in a moment) is intuitive and works well. However, there are a few special cases where the lack of strict type checking creates some surprising results; so we ll discuss this issue in detail here.

To understand the motivation for HLA s relaxation of strict type-checking, consider the following procedure declaration:

```
   procedure uns8RefParm( var b:uns8 );
   begin uns8RefParm;
      .
      .
      .
   end uns8RefParm;
```

Whenever you call this procedure, you have to pass it an actual parameter that is either an `uns8` typed object or a `byte` object (remember, you can pass a byte object as a parameter whenever a byte-sized reference parameter is required). HLA will emit the code to take the address of that one-byte object in memory and pass this address on to the procedure. Note that within the `uns8RefParm` procedure, like any reference parameter, HLA treats b as though it were a dword object (because pointers always require 32 bits, regardless of the size of the object at which they point). Here is a typical call to the `uns8RefParm` procedure:

```
static
   uns8Var :uns8;
      .
      .
      .
   uns8RefParm( uns8Var );
```

Here s the code HLA might generate for this procedure call:

```
   pushd( &uns8Var );
   call uns8RefParm;
```

Now consider the following code fragments:

```
type
   pUns8 : pointer to uns8;

static
   pUns8Var :pUns8;
   uns8Var  :uns8;
      .
      .
      .
   mov( &uns8Var, pUns8Var ); // Initialize pUns8Var with the address of uns8Var
      .
      .
      .
   uns8RefParm( pUns8Var );
```

Where HLA to strictly enforce type checking of reference parameters, this call to `uns8RefParm` would generate an error. After all, this code is trying to pass a four-byte pointer object by reference to a procedure that is expecting an `uns8` reference parameter. The two variables in memory are not at all compatible even though the pointer object, itself, points at an `uns8` variable in memory. In a typical high level language, it makes a lot of sense to report this as an error and reject the source file as uncompilable. In assembly language, however, it is very common to refer to objects (even static ones) using pointers to those objects that you generate on the fly (e.g., pointer values that you load into registers and possibly manipulate in those registers). As a result, you ve often got the address of a variable sitting around in a register or a pointer and you d like to pass that address on through to the procedure rather than taking the address of the object. After all, any assembly language programmer realizes that pass by reference parameters are just syntactical sugar for saying take the address of the actual parameter because the procedure is expecting us to pass it an address. If you ve already got the address somewhere, why bother computing it again?

If the address is sitting in a 32-bit register, you can always pass the address using a memory operand like [ebx] . HLA will, nicely, assume that an untyped memory operand like this actually points at a correct object. So you could call `uns8RefParm` as follows:

```
   uns8RefParm( [ebx] );
```

Theoretically, it would be better for you to write this as

```
   uns8RefParm( (type uns8 [ebx]) );
```

However, experience shows that requiring the type coercion here is just too inconvenient to consider. So HLA relaxes the requirement a bit and allows the use of anonymous memory references (those involving an indirect address in a 32-bit register without any other type or name information) as legal arguments for any pass by reference parameter.

Now consider the following call to `uns8RefParm`:

```
   uns8RefParm( pUns8Var );
```

The `pUns8Var` argument should, technically, be illegal. This is not an object of byte `uns8` (or `byte`) and, therefore, HLA should generate a type mismatch error in response to this procedure invocation. As it turns out, however, it is very common in assembly language programs (especially in Win32 assembly programs) to want to pass the pointer to a memory object in a reference parameter slot. One could easily do this by explicitly passing the value of the pointer as the parameter, e.g.,

```
   push( pUns8Var );
   call uns8RefParm;

// or

   uns8RefParm( #{ push( pUns8Var ); }# );
```

However, the need to pass such a pointer variable s value in place of taking the address of some variable occurs so frequently that HLA relaxes the type checking of a reference parameter to allow any pointer variable that points at the base type of the formal parameter s type. That is, it is legal to pass a pointer to an `uns8` variable as a pass by reference parameter whose type is `uns8`. Rather than take the address of the pointer variable, such a procedure invocation would copy the value of that pointer variable (which is the address of some other object) as the actual parameter data, e.g.,

```
   uns8RefParm( uns8Var );  // Legal, of course
   uns8RefParm( pUns8Var ); // Also legal

// The above two statements emit the following code:

   push( &uns8Var );
   call uns8RefParm;

   push( pUns8Var );  // Pushes the value of pUns8Var, not it’s address!
   call uns8RefParm;
```

Note that untyped reference parameters always take the address of the actual parameter to pass on to the procedure, even if the actual parameter is a pointer. This is an unfortunately gotcha that you re going to have to remember. Consider the following code as an example of this problem:

```
procedure untypedParm( var v:var );
      .
      .
      .
procedure typedParm( var v:char );
      .
      .
      .
static
   p :pointer to char;
      .
      .
      .
   typedParm( p ); // Legal, passes value of p to typedParm.
   untypedParm( p ); // Legal passes the *address* of p to untypedParm.

// code equivalent to the above two calls:

   pushd( p );   // Note that this pushes p’s value, not its address.
   call typedParm;

   pushd( &p );  // Notice that this pushes the address of p, not p’s value.
   call untypedParm;
```

This behavior isn t entirely intuitive. As a result, many programmers who ve gotten in the habit of passing a pointer variable as a reference parameter wind up getting burned when the formal parameter is an untyped reference parameter. This problem crops up enough that it s quite possible that HLA will drop support for automatically passing the value of a pointer variable through a reference parameter in some future version of the compiler. Therefore, you should avoid passing a pointer variable as a reference parameter, expecting HLA to automatically push the value (rather than the address) of the pointer variable onto the stack.

Although you should avoid having HLA automatically pass a pointer variable s value through a reference parameter, the need to pass such a value exists. Recent versions of HLA have made use of HLA s val reserved word as a prefix to an actual parameter to tell HLA to pass that parameter s value as the address the reference parameter requires. To do this, simply prefix the actual parameter with the `val` keyword, e.g.,

```
      untypedParm( ptrVar );      // Passes the address of ptrVal
      untypedParm( val ptrVar );  // Passes ptrVar’s value.
```

The `val` keyword actually works with any `dword` or pointer variable type, not just pointers to the underlying type, e.g.,

```
         typedParm( val dwordVar );
```

In HLA, strings are special cases of pointer objects. If you pass a string variable as a parameter that is an untyped reference parameter, HLA will actually pass the address of the string variable (that is, the address of the pointer) rather than the address of the string data (which is where the string is pointing). Several Win32 API functions, for example, expect you to pass them the address of a sequence of characters (i.e., a string) and you ll probably have that data sitting in an HLA string variable in your code. However, if you directly pass the string variable as the parameter to the Win32 API function, HLA will actually take the address of the string pointer variable rather than pass the address of the character data on through to the procedure. You can counteract this behavior by prefixing the string variable with the `val` keyword in the actual procedure call, e.g.,

```
   procWithUntypedParm( val strVariable );
```

Because strings are pointers, you may also use the `val` keyword to pass an arbitrary 32-bit numeric value for a string parameter. This is useful in certain Win32 API calls where you pass either a pointer to a zero-terminated sequence of characters (i.e., a string) or a small integer ATOM value, e.g.,

```
proceWithStrParm( val 12345 );  // Normally requires a string parameter.
```

Whenever you re in doubt as to what HLA is doing with your code, you should take a look at the assembly language output file that HLA produces. Although the syntax is different than HLA, you should be able to figure out what code the compiler is generating for your procedure calls. If this isn t the code you want, you can always pass the parameters manually or apply some operator like `val` to an actual parameter.

2.5: The HLA Compile-Time Language

One of HLAs more powerful features, and a feature we ll use quite a bit in this book, is the HLA *compile-time language*. The concept of a compile-time language may seem pretty weird when you first consider it, but the truth is this term is really just a new label for something that s been around a long time: a built-in macro processor. The main reason HLA calls its macro processor a compile-time language is because HLAs macro facilities are quite a bit more powerful than those available with most assemblers.

Actually, to equate the term compile-time language with macro processor is a bit of an injustice. A compile-time language includes several things that many people often think of as being separate from macro processing, including conditional assembly/compilation, string processing (e.g., of macro operands), repetitive code emission, and so on. Most languages that provide compile-time language features provide a minimal set of such constructs in the overall language. Not so with HLA. It s actually possible to write complete applications using nothing more than the HLA compile-time language; such usage isn t entirely appropriate or efficient, but it is possible to create simple applications within HLAs compile-time language facility.

The real purpose of the HLA compile-time language is to automate the creation of certain common code sequences. A simple macro is a good, well-understood, example of this. By invoking a macro, you can expand some sequence of machine instructions over and over again in an assembly language file, thus automating the creation of the code sequences those macros produce. Macros aren t the only constructs that can automate the generation of code at compile-time, however. HLA also provides loops, declarations, assignments, I/O, and other facilities we can use to automate code generation.

Before explaining how to use HLAs compile-time language, perhaps it s a good idea to explicitly define what a compile-time language and a run-time language are and how you would use them. The run-time language is the language that you use to create programs that run after compilation completes; in the case of HLA, the run-time language is 80x86 assembly language and a run-time program is the 80x86 machine code resulting from an HLA compilation. The run-time language includes all the 80x86 machine instructions, data declaration statements, procedure declarations, and so on). It is not physically possible for the run-time program to execute concurrently with its production by the HLA compiler (obviously, you can run a previous instance of that program while HLA is running, but we re talking about the machine code that HLA produces during compilation here). The compile-time language, on the other hand, consists of a handful of statements that HLA executes, via interpretation, while it is compiling the run-time program. The two languages are not entirely distinct. For example, constant and value declarations in the HLA declaration section are usable by both the run-time and compile-time programs. Despite the overlap, it s important to realize that the two languages are distinct and the presence of compile-time language statements in an HLA source file may not produce equivalent execution semantics in a run-time program.

The best place to start when discussing the compile-time language is with the HLA declaration section. The compile-time and run-time languages share the HLA declaration sections (though the two languages sometimes interpret the declarations differently). So let s look at each of the declaration sections that appear in an HLA source file and compare their effects on both the compile-time and run-time programs in that source file.

The HLA `const` declaration section defines constant values in both the compile-time and run-time languages. Any declaration appearing in an HLA `const` section is available in both languages in the same source file (subject, of course, to the scoping rules that apply to that identifier). The only issue is that HLA `text` constants are only usable at compile-time.

The HLA `val` declaration section also declares constants as far as the run-time language is concerned. However, this is where you declare compile-time *variables*. The difference between HLA `const` values and HLA `val` values is the fact that (during assembly) you can change the value of a `val` object. Hence, at compile-time, `val` objects are variables. Note that the HLA compile-time ? (assignment) statement also creates `val` objects that the run-time program can treat as a constant value during the application s execution. We ll return to this statement in a moment.

The HLA type declaration section is also applicable to both compile-time and run-time programs. That is, you can define types that are usable by both compile-time and run-time programs in this declaration section. Note, however, that there are a couple of types that don t make sense at compile-time. Specifically you cannot create compile-time `class` objects, `thunks`, and a few other types. As noted for `const` and `val` objects, if you define a new type based on HLA s built-in `text` type, that type is only usable in a compile-time program.

The `static, readonly, storage,` and `var` sections create run-time variables. HLA s compile-time language provides compile-time function that let you test the compile-time attributes of these variable declarations (e.g., determine their type, the number of array elements, stuff like that), but the compile-time program cannot access the run-time values of these variables (because run-time occurs long after the compile-time program terminates). For `static` and `readonly` objects the compile-time language can specify the initial value loaded into memory for these run-time variables, but the compile-time language really has no idea what value a run-time variable will have at a given point during program execution.

HLA procedures, methods, and iterators are purely run-time entities. Like run-time variables, the HLA compile-time language can test certain `procedure/iterator/method` name attributes, but the compile-time language cannot call this subroutines nor determine much else about them.

An HLA namespace is an encapsulation of a declaration section. Namespaces may encapsulate run-time as well as compile-time declarations, so namespaces are equally applicable to both types of programs. Accessing fields of the namespace is done the same way in both compile-time and run-time programs (e.g., using dot-notation).

HLA classes are mainly a run-time construct. However, classes, like namespaces, open up a new declaration section in which you can declare constants, values, and variables. So although you cannot create `class` constants at compile-time, a `class` may contain several compile-time language elements (e.g., `const, val,` and macro declarations). Though this book use classes and objects in a few sample programs, such usage is going to be rare enough that we can effectively ignore classes in our discussion here.

So what does a compile-time program look like, anyway? That s a good question and there isn t really a specific definition of a compile-time program. Some might consider an entire HLA source file containing compile-time statements to be a single compile-time program. Other programmers might consider a short sequence of compile-time statements in a source file to be a single compile-time program (so a single HLA source file could contain several compile-time program-ettes). HLA doesn t really define what constitutes a single compile-time program, so we ll just work on the assumption that any sequence of compile-time statements used to compute some compile-time value or create a single run-time sequence of statements constitutes a single compile-time program (i.e., there can be multiple compile-time programs in a single HLA source file). Well, enough of the philosophy, let s look at the statements that are available to HLA compile-time language programmers.

2.5.1: Compile-Time Assignment Statements

One of the more important compile-time statements is the compile-time assignment statement. This statement takes one of these forms:

```
? compile_time_variable := compile_time_expression ;
? compile_time_variable : type_id := compile_time_expression ;
? compile_time_variable : type_id;
? compile_time_array_variable[ dimension_list ] := compile_time_expression ;
? @built_in_variable := compile_time_expression ;
```

Here are some examples:

```
? i := 5;
? b :boolean := true;
? a :int32[ 4] ;  // Default initialization is all zeros.
? a[ 2]  := 5;
? @nodisplay := true;
```

Note that the compile-time assignment statement is almost semantically identical to a `val` section declaration. In fact, the internal HLA code is almost identical for handling the `val` section versus the ? statement. The only semantic difference between the two has to do with identifier scope. Consider the following code fragment:

```
val
   i :int32;
   j :uns32;

   procedure nestMe;
   val
      i :real := 3.0; // Creates a new “i” in this procedure’s scope.

      i := 2.2;       // Resets the local i’s value to 2.2, does not affect value of i
                      //  in the outer scope.
         .
         .
         .
      ?j := 5;        // Stores 5 into the outer scope’s “j”.
         .
         .
         .
   end nestMe;
```

As you can see in this example, a `val` declaration will always create a new instance of a compile-time variable if that variable does not already exist in the current scope, then HLA will create a new variable in the current scope (though if the compile-time variable does exist in the current scope, then HLA will use that variable). The compile-time assignment operator will always attempt to use a predefined version of the compile-time variable, even if that variable was declared in an outer scope. The compile-time assignment statement will only create a new compile-time variable if it is not visible at all at the point in the program where the assignment occurs.

HLA compile-time variables use a *dynamic typed system*. This means that you can (usually) change the type of a compile-time variable after you ve declared it as some other type. E.g.,

```
? i:int32;  // Declare i as an int32 compile-time object
    .
    .   // compile-time code that uses i as an int32 compile-time variable.
    .
? i :string := "i is now a string"; //From this point forward, i is a string object.
```

The one point we ve not really discussed here is exactly what is legal for an expression in an HLA compile-time assignment statement. The truth is, HLA provides a very sophisticated expression syntax that is very similar to what you would find in a high level language like C/C++ or Pascal. In fact, with all the built-in compile-time functions, one could easily argue that HLAs compile-time language is even more sophisticated that what you ll find in many high level languages. The bottom line is that HLA constant expressions are a bit too complex to do justice here; you ll have to read the HLA Reference Manual to get the full picture. Nevertheless, it s worth listing the arithmetic and relational operators because you ll frequently use them in HLA compile-time programs.

```
! (unary not), - (unary negation)
*, div, mod, /, <<, >>
+, -
=, = =, <>, !=, <=, >=, <, >
&, |, &, in
```

One of HLAs more powerful features is the ability to support structured constants in the compile-time language. Because we ll be using structured constants throughout this book, it s worthwhile to review character set, record, and union constants.

A character set literal constant consists of several comma delimited character set expressions within a pair of braces. The character set expressions can either be individual character values or a pair of character values separated by an ellipse (..). If an individual character expression appears within the character set, then that character is a member of the set; if a pair of character expressions, separated by an ellipse, appears within a character set literal, then all characters between the first such expression and the second expression are members of the set.

Examples:

```
{ 'a' ,' b' ,' c' }            // a, b, and c.
{ 'a' ..' c' }                 // a, b, and c.
{ 'A' ..' Z' ,' a' ..' z' }    //Alphabetic characters.
{ ' ',#$d,#$a,#$9}             //Whitespace (space, return, linefeed, tab).
```

HLA character sets are currently limited to holding characters from the 128-character ASCII character set.

HLA lets you specify an array literal constant by enclosing a set of values within a pair of square brackets. Since array elements must be homogenous, all elements in an array literal constant must be the same type or conformable to the same type. Examples:

```
[  1, 2, 3, 4, 9, 17 ]
[  ' a' , ' A' , ' b' , ' B'  ]
[  "hello", "world" ]
```

Note that each item in the list of values can actually be a constant expression, not a simple literal value.

HLA array constants are always one dimensional. This, however, is not a limitation because if you attempt to use array constants in a constant expression, the only thing that HLA checks is the total number of elements. Therefore, an array constant with eight integers can be assigned to any of the following arrays:

```
const
   a8:       int32[ 8]        := [ 1,2,3,4,5,6,7,8];
   a2x4:     int32[ 2,4]      := [ 1,2,3,4,5,6,7,8];
   a2x2x2:   int32[ 2,2,2]    := [ 1,2,3,4,5,6,7,8];
```

Although HLA doesn t support the notation of a multi-dimensional array constant, HLA does allow you to include an array constant as one of the elements in an array constant. If an array constant appears as a list item within some other array constant, then HLA expands the interior constant in place, lengthening the list of items in the enclosing list. E.g., the following three array constants are equivalent:

```
[ [ 1,2,3,4], [ 5,6,7,8] ]
[ [ [ 1,2], [ 3,4] ], [ [ 5,6], [ 7,8] ] ]
[ 1,2,3,4,5,6,7,8]
```

Although the three array constants are identical, as far as HLA is concerned, you might want to use these three different forms to suggest the shape of the array in an actual declaration, e.g.,

```
const
   a8:       int32[ 8]        := [ 1,2,3,4,5,6,7,8];
   a2x4:     int32[ 2,4]      := [  [ 1,2,3,4], [ 5,6,7,8] ];
   a2x2x2:   int32[ 2,2,2]    := [[[ 1,2], [ 3,4] ], [[ 5,6], [ 7,8]]];
```

Also note that symbol array constants, not just literal array constants, may appear in a literal array constant. For example, the following literal array constant creates a nine-element array holding the values one through nine at indexes zero through eight:

```
const Nine: int32[  9 ]      := [  a8, 9 ];
```

This assumes, of course, that a8 was previously declared as above. Since HLA flattens all array constants, you could have substituted a2x4 or ax2x2x for a8 in the example above and obtained identical results.

You may also create an array constant using the HLA dup operator. This operator uses the following syntax:

expression dup [*expression_to_replicate*]

Where *expression* is an integer expression and *expression_to_replicate* is a some expression, possibly an array constant. HLA generates an array constant by repeating the values in the *expression_to_replicate* the number of times specified by the expression. (Note: this does not create an array with *expression* elements unless the *expression_to_replicate* contains only a single value; it creates an array whose element count is expression times the number of items in the *expression_to_replicate*). Examples:

```
10 dup [ 1]    -- equivalent to [ 1,1,1,1,1,1,1,1,1,1]
5 dup [ 1,2]   -- equivalent to [ 1,2,1,2,1,2,1,2,1,2]
```

Please note that HLA does not allow class constants, so class objects may not appear in array constants.

HLA supports record constants using a syntax very similar to array constants. You enclose a comma-separated list of values for each field in a pair of square brackets. To further differentiate array and record constants, the name of the record type and a colon must precede the opening square bracket, e.g.,

```
type
   Planet:
      record
         x:          int32;
         y:          int32;
         z:          int32;
         density:    real64;
      endrecord;


const
   somePlanet : Planet := Planet:[  1, 12, 34, 1.96 ]
```

HLA associates the items in the list with the fields as they appear in the original record declaration. In this example, the values 1, 12, 34, and 1.96 are associated with fields `x, y, z,` and `density`, respectively. Of course, the types of the individual constants must match (or be conformable to) the types of the individual fields.

Note that you may not create a record constant for a particular record type if that record includes data types that cannot have compile-time constants associated with them. For example, if a field of a record is a class object, you cannot create a record constant for that type since you cannot create class constants.

Union constants allow you to initialize static union data structures in memory as well as initialize union fields of other data structures (including anonymous union fields in records). There are some important differences between HLA compile-time union constants and HLA run-time unions (as well as between the HLA run-time union constants and unions in other languages). Therefore, it s a good idea to begin the discussion of HLA s union constants with a description of these differences.

There are a couple of different reasons people use unions in a program. The original reason was to share a sequence of memory locations between various fields whose access is mutually exclusive. When using a union in this manner, one never reads the data from a field unless they ve previous written data to that field and there are no intervening writes to other fields between that previous write and the current read. The HLA compile-time language fully (and only) supports this use of union objects.

A second reason people use unions (especially in high level languages) is to provide aliases to a given memory location; particularly, aliases whose data types are different. In this mode, a programmer might write a value to one field and then read that data using a different field (in order to access that data s bit representation as a different type). **HLA does not support this type of access to union constants**. The reason is quite simple: internally, HLA uses a special variant data type to represent all possible constant types. Whenever you create a union constant, HLA lets you provide a value for a single data field. From that point forward, HLA effectively treats the union constant as a scalar object whose type is the same as the field you ve initialized; access to the other fields through the union constant is no longer possible. Therefore, you cannot use HLA compile-time constants to do type coercion; nor is there any need to since HLA provides a set of type coercion operators like `@byte, @word, @dword, @int8,` etc. As noted already, the main purpose for providing HLA union constants is to allow you to initialize static union variables; since you can only store one value into a memory location at a time, union constants only need to be able to represent a single field of the union at one time (of course, at run-time you may access any field of the static union object you ve created; but at compile-time you may only access the single field associated with a union constant).

An HLA literal union constant takes the following form:

```
    typename.fieldname:[  constant_expression ]
```

The *`typename`* component above must be the name of a previously declared HLA union data type (i.e., a `union` type you ve created in the `type` section). The *`fieldname`* component must be the name of a field within that union type. The *constant_expression* component must be a constant value (expression) whose type is the same as, or is automatically coercible to, the type of the *fieldname* field. Here is a complete example:

```
type
   u:union
      b:byte;
      w:word;
      d:dword;
      q:qword;
   endunion;

static
   uVar  :u    := u.w:[ $1234] ;
```

The declaration for `uVar` initializes the first two bytes of this object in memory with the value $1234. Note that `uVar` is actually eight bytes long; HLA automatically zeros any unused bytes when initializing a static memory object with a union constant.

Note that you may place a literal union constant in records, arrays, and other composite data structures. The following is a simple example of a record constant that has a union as one of its fields:

```
type
   r  :record
         b:byte;
         uf:u;
         d:dword;
   endrecord;

static
   sr :r := r:[ 0, u.d:[ $1234_5678] , 12345] ;
```

In this example, HLA initializes the `sr` variable with the byte value zero, followed by a `dword` containing $1234_5678 and a `dword` containing zero (to pad out the remainder of the union field), followed by a `dword` containing 12345.

You can also create records that have anonymous unions in them and then initialize a record object with a literal constant. Consider the following type declaration with an anonymous union:

```
type
   rau :record
         b:byte;
         union
            c:char;
            d:dword;
         endunion;
         w:word;
   endrecord;
```

Since anonymous unions within a record do not have a type associated with them, you cannot use the standard literal union constant syntax to initialize the anonymous union field (that syntax requires a type name). Instead, HLA offers you two choices when creating a literal record constant with an anonymous union field. The first alternative is to use the reserved word union in place of a typename when creating a literal union constant, e.g.,

```
static
```

```
   srau :rau := rau:[  1, union.d:[ $12345] , $5678 ] ;
```

The second alternative is a shortcut notation. HLA allows you to simply specify a value that is compatible with the first field of the anonymous union and HLA will assign that value to the first field and ignore any other fields in the union, e.g.,

```
static
   srau2 :rau := rau:[  1, 'c' , $5678 ] ;
```

This is slightly dangerous since HLA relaxes type checking a bit here, but when creating tables of record constants, this is very convenient if you generally provide values for only a single field of the anonymous union; just make sure that the commonly used field appears first and you re in business.

Although HLA allows anonymous records within a union, there was no syntactically acceptable way to differentiate anonymous record fields from other fields in the union; therefore, HLA does not allow you to create union constants if the union type contains an anonymous record. The easy workaround is to create a named record field and specify the name of the record field when creating a union constant, e.g.,

```
type
   r  :record
         c:char;
         d:dword;
   endrecord;

   u  :union
         b:byte;
         x:r;
         w:word;
   endunion;

static
   y :u := u.x:[  r:[  'a' , 5]] ;
```

You may declare a union constant and then assign data to the specific fields as you would a record constant. The following example provides some samples of this:

```
type
   u_t :union
         b:byte;
         x:r;
         w:word;
   endunion;

val
   u :u_t;
      .
      .
      .
   ?u.b := 0;
      .
      .
      .
   ?u.w := $1234;
```

The two assignments above are roughly equivalent to the following:

```
?u := u_t.b:[ 0];
```

and

```
?u := u_t.w:[ $1234];
```

However, to use the straight assignment (the former example) you must first declare the value u as a u_t union.

To access a value of a union constant, you use the familiar dot notation from records and other languages, e.g.,

```
?x := u.b;
    .
    .
    .
?y := u.w & $FF00;
```

Note, however, that you may only access the last field of the union into which you ve stored some value. If you store data into one field and attempt to read the data from some other field of the union, HLA will report an error. Remember, you don t use union constants as a sneaky way to coerce one value s type to another (use the coercion functions for that purpose).

HLA allows a very limited form of a pointer constant. If you place an ampersand (&) in front of a static object s name (i.e., the name of a static variable, readonly variable, uninitialized variable, segment variable, procedure, method, or iterator), HLA will compute the run-time offset of that variable. Pointer constants may not be used in arbitrary constant expressions. You may only use pointer constants in expressions used to initialize static or readonly variables or as constant expressions in 80x86 instructions. The following example demonstrates how pointer constants can be used:

```
program pointerConstDemo;

static
   t:int32;
   pt: pointer to int32 := &t;

begin pointerConstDemo;

   mov( &t, eax );

end pointerConstDemo;
```

Also note that HLA allows the use of the reserved word NULL anywhere a pointer constant is legal. HLA substitutes the value zero for NULL.

You may assign any of these structured literal constants to a compile-time variable or constant using an HLA compile-time assignment statement.

2.5.2: Compile-Time Functions

HLA provides the ability to create your own compile-time functions using macros, and we ll discuss how to do that in a little bit. In this section, however, we ll take a look at a small sample of the built-in compile-time functions that HLA automatically provides for you. HLA actually provides a large number of compile-time functions, too many to present here, so in this section we ll take a look at the ones that Win32 programmers commonly use. Please see the HLA Reference manual for more details on the HLA compile-time function facilities.

Though it is perfectly possible to write an assembly language program without ever calling an HLA compile-time function (or using the compile-time language at all, for that matter), the HLA compile-time functions can make it easier to develop certain macros and they can help make your programs easier to write and maintain. Therefore, it s a good idea to familiarize yourself with the HLA compile-time functions.

Note: don t confuse HLA s compile-time functions with the functions available in the HLA Standard Library. Although there are many similarities, Standard Library functions are run-time functions, compile-time functions execute during compilation.

The first set of compile-time functions to look at are the type-conversion functions. These compile-time function convert a constant expression you pass as an argument to the corresponding data type (or create a compile-time error if the conversion is not possible). These conversion functions are unusual insofar as they are the only set of HLA compile-time functions that don t begin with @ (this functions just use the built-in HLA type names).

```
boolean( const_expr )
```

The expression must be an ordinal or string expression. If `const_expr` is numeric, this function returns false for zero and true for everything else. If `const_expr` is a character, this function returns true for T and false for F . It generates an error for any other character value. If `const_expr` is a string, the string must contain true or false else HLA generates an error.

```
int8( const_expr ), int16( const_expr ), int32( const_expr )
int64( const_expr ), int128( const_expr )
uns8( const_expr ), uns16 const_expr ), uns32( const_expr )
uns64( const_expr ), uns128( const_expr )
byte( const_expr ), word( const_expr ), dword( const_expr )
qword( const_expr ), lword( const_expr )
```

These functions convert their parameter to the specified integer. For real operands, the result is truncated to form a numeric operand. For all other numeric operands, the result is ranged checked. For character operands, the ASCII code of the specified character is returned. For `boolean` objects, zero or one is returned. For `string` operands, the string must be a sequence of decimal characters which are converted to the specified type. Note that `byte`, `word`, and `dword` types are synonymous with `uns8`, `uns16`, and `uns32` for the purposes of range checking.

```
real32( const_expr ), real64( const_expr ), real80( const_expr )
```

Similar to the integer functions above, except these functions produce the obvious real results. Only numeric and string parameters are legal.

```
char( const_expr )
```

`Const_expr` must be a ordinal or string value. This function returns a character whose ASCII code is that ordinal value. For strings, this function returns the first character of the string.

```
string( const_expr )
```

This function produces a reasonable string representation of the parameter. Almost all data types are legal.

```
cset( const_expr )
```

The parameter must be a character, string, or cset. For character parameters, this function returns the singleton set containing only the specified character. For strings, each character in the string is unioned into the set and the function returns the result. If the parameter is a cset, this function makes a copy of that character set.

The type conversion functions just described will automatically convert their operands from the source type to the destination type. Sometimes you might want to change the type of some object without changing its value. For many conversions this is exactly what takes place. For example, when converting and `uns8` object to an `uns16` value using the `uns16(---)` function, HLA does not modify the bit pattern at all. For other conversions, however, HLA may completely change the underlying bit pattern when doing the conversion. For example, when converting the `real32` value 1.0 to a `dword` value, HLA completely changes the underlying bit pattern ($3F80_0000) so that the `dword` value is equal to one. On occasion, however, you might actually want to copy the bits straight across so that the resulting `dword` value is $3F80_0000. The HLA bit-transfer type conversion compile-time functions provide this facility.

The HLA bit-transfer type conversion functions are the following:

```
@int8( const_expr ), @int16( const_expr ), @int32( const_expr )
@int64( const_expr ), @int128( const_expr )
@uns8( const_expr ), @uns16 const_expr ), @uns32( const_expr )
@uns64( const_expr ), @uns128( const_expr )
@byte( const_expr ), @word( const_expr ), @dword( const_expr )
@qword( const_expr ), @lword( const_expr )
@real32( const_expr ), @real64( const_expr ), @real80( const_expr )
@char( const_expr )
@cset( const_expr )
```

The above functions extract eight, 16, 32, 64, or 128 bits from the constant expression for use as the value of the function. Note that supplying a string expression as an argument isn t particularly useful since the functions above will simply return the address of the string data in memory while HLA is compiling the program. The `@byte` function provides an additional syntax with two parameters, we ll describe that in a moment.

```
@string( const_expr )
```

HLA string objects are pointers (in both the language as well as within the compiler). So simply copying the bits to the internal string object would create problems since the bit pattern probably is not a valid pointer to string data during the compilation. With just a few exceptions, what the `@string` function does is takes the bit data of its argument and translates this to a string (up to 16 characters long). Note that the actual string may be between zero and 16 characters long since the HLA compiler (internally) uses zero-terminated strings to represent string constants. Note that the first zero byte found in the argument will end the string.

If you supply a `string` expression as an argument to `@string`, HLA simply returns the value of the string argument as the value for the `@string` function. If you supply a `text` object as an argument to the `@string` function, HLA returns the text data as a string without first expanding the text value (similar to the `@string:`*`identifier`* token). If you supply a pointer constant as an argument to the `@string` function, HLA returns the string that HLA will substitute for the static object when it emits the assembly file.

HLA provides a fair set of functions that let you determine attributes of various symbols and expressions during compilation. Here are some of these compile-time functions you ll commonly use:

```
]@name( identifier )
```

This function returns a string of characters that corresponds to the name of the identifier (note: after text/macro expansion). This is useful inside macros when attempting to determine the name of a macro parameter variable (e.g., for error messages, etc.). This function returns the empty string if the parameter is not an identifier.

`@typename(` *`identifier_or_expression`* `)`

This function returns the string name of the type of the identifier or constant expression. Examples include int32 , boolean , and real80 .

`@size(` *`identifier_or_expression`* `)`

This function returns the size, in bytes, of the specified object.

`@elementsize(` *`identifier_or_expression`* `)`

This function returns the size, in bytes, of an element of the specified array. If the parameter is not an array identifier, this function generates an assembly-time error.

`@offset(` *`identifier`* `)`

For VAR, PARM, METHOD, and class ITERATOR objects only, this function returns the integer offset into the activation record (or object record) of the specified symbol.

`@Is External(` *`identifier`* `)`

This function returns true if the specified identifier is an external symbol.

`@arity(` *`identifier_or_expression`* `)`

This function returns zero if the specified identifier is not an array. Otherwise it returns the number of dimension of that array.

`@dim(` *`array_identifier_or_expression`* `)`

This function returns a single array of integers with one element for each dimension of the array passed as a parameter. Each element of the array returned by this function gives the number of elements in the specified dimension. For example, given the following code:

```
val threeD: int32[  2, 4, 6];
   tdDims:= @dim( threeD );
```

The `tdDims` constant would be an array with the three elements [2, 4, 6];

`@elements(` *`array_identifier_or_expression`* `)`

This function returns the total number of elements in the specified array. For multi-dimensional array constants, this function returns the number of all elements, not just a particular row or column.

`@defined(` *`identifier`* `)`

This function returns true if the specified identifier is has been previously defined in the program and is currently in scope.

```
@isconst( expr ), @isreg( expr ), @isreg8( expr ), @isreg16( expr ), @isreg32( expr )
@isfreg( expr ),@ismem( expr ), @istype( identifier )
```

These functions return true if their argument is a constant, register, memory address, or type ID. These functions are useful in classifying macro parameters. This is not a complete list of the HLA compile-time classification functions, please consult the HLA Reference Manual to see the complete set.

`@linenumber`

This function returns the current line number in the source file.

`@filename`

This function returns the name of the current source file.

HLA provides several general purpose numeric functions that are great for initializing tables and doing other compile-time calculations. Here are some of these functions:

`@abs(` *`numeric_expr`* `)`

Returns the absolute equivalent of the numeric value passed as a parameter.

`@byte(` *`integer_expr, which`* `), @byte(` *`real32_expr, which`* `), @byte(` *`real64_expr, which`* `)`
`@byte(` *`real80_expr, which`* `)`

The @byte function extracts a single byte from a multi-byte data type. For integer/ordinal expressions, the *`which`* parameter is a value in the range 0..15. For `real32` operands, the *`which`* parameter is a value in the range 0..3. This function extracts the specified byte from the value of the *`real32_expression`* parameter. For `real64` operands, the *`which`* parameter is a value in the range 0..7. For real80 operands, the *`which`* parameter is a value in the range 0..9. These functions extract the specified byte from the value of their second operand.

`@ceil(` *`real_expr`* `), @floor(` *`real_expr`* `)`

The `@ceil` function returns the smallest integer value larger than or equal to the expression passed as a parameter. The `@floor` function returns the largest integer value less than or equal to the supplied expression. Note that although the result will be an integer, these functions return a `real80` value.

`@cos(` *`real_expr`* `), @sin(` *`real_expr`* `), @tan(` *`real_expr`* `)`

The real parameter is an angle in radians. These functions return the usual trigonometric values for the parameter you pass them.

`@date`

This function returns a string of the form YYYY/MM/DD containing the current date.

`@exp(` *`real_expr`* `)`

This function returns a `real80` value that is the result of the computation e^{real_expr}.

`@extract(` *`cset_expr`* `)`

This function returns a character from the specified character set constant. Note that this function doesn t actually remove the character from the set, if you want to do that, then you will need to explicitly remove the character yourself.

`@isalpha(` *`char_expr`* `), @isalphanum(` *`char_expr`* `), @isdigit(` *`char_expr`* `)`
`@islower(` *`char_expr`* `), @isspace(` *`char_expr`* `), @isupper(` *`char_expr`* `)`
`@isxdigit(` *`char_expr`* `)`

These predicate functions return true or false based on whether there arguments belong to a certain set of characters. Note that isxdigit returns true if its argument is a hexadecimal character.

`@log(` *`real_expr`* `), @log10(` *`real_expr`* `)`

@log returns the natural (base-e) logarithm of its argument, @log10 returns the base-10 logarithm of the supplied parameter.

`@max(` *`comma_separated_list_of_ordinal_or_real_values`* `),`
`@min(` *`comma_separated_list_of_ordinal_or_real_values`* `)`

These functions return the maximum or minimum of the values in the specified list.

`@odd(` *`int_expr`* `)`

This function returns true if the integer expression is an odd number.

`@random(` *`int_expr`* `)`

This function returns a random uns32 value.

`@randomize(` *`int_expr`* `)`

This function uses the integer expression passed as a parameter as the new seed value for the random number generator.

`@sqrt(` *`real_expr`* `)`

This function returns the square root of the parameter.

`@time`

This function returns a string of the form HH:MM:SS xM (x= A or P) denoting the time at the point this function was called (according to the system clock).

HLA provides a large number of compile-time string functions. These functions are especially useful for processing macro parameters and other string constants in an HLA compile-time program. Here are some of the compile-time string functions that HLA provides (see the *HLA Reference Manual* for the complete list):

`@delete(` *`str_expr, int_start, int_len`* `)`

This function returns a string consisting of the `str_expr` passed as a parameter with (possibly) some characters removed. This function removes `int_len` characters from the string starting at index `int_start` (note that strings have a starting index of zero).

`@index(` *`str_expr1, int_start, str_expr2`* `)`

This function searches for `str_expr2` within `str_expr1` starting at character position `int_start` within `str_expr1`. If the string is found, this function returns the index into `str1_expr1` of the first match (starting at `int_start`). This function returns -1 if there is no match.

`@insert(` *`str_expr1, int_start, str_expr2`* `), @index(` *`str_expr1, int_start, str_expr2`* `)`

The insert function inserts `str_expr2` into `str_expr1` just before the character at index `int_start`. The `rindex` function does the same, except `int_start` is from the end of the string rather than the start of the string. These functions return the converted string (note that these functions do not modify the original string).

`@length(` *`str_expr`* `)`

This function returns the length of the specified string.

`@lowercase( str_expr, int_start ), @uppercase( str_expr, int_start )`

These functions return a string of characters from `str_expr` with all uppercase alphabetic characters converted to lower case or vice versa. Only those characters from `int_start` on are copied into the result string.

`@substr( str_expr, int_start, int_len )`

This function returns the substring specified by the starting position and length in `str_expr`.

HLA also provides a large number of string/pattern matching functions. Because of the large number of these functions, we ll not regurgitate their description here. Please see the HLA Reference Manual for details on the compile-time pattern matching functions.

HLA provides a large number of additional compile-time functions and pseudo-variables that you may find useful when creating compile-time programs. Space limitations prevent the complete exposition of these functions here. This chapter just high-lights some of the more common functions in order to give you a taste of what is possible when using the HLA compile-time language. As a Win32 programmer, you might not use all of the functions this chapter describes and you might also use several of HLA s compile-time functions that this chapter does not describe. The set of functions you might use in an application are as varied as the number of applications that have been written.

2.5.3: Generating Code With a Compile-Time Statement

Up to this point, the HLA compile-time statements and functions we ve seen generate constant results. They re useful for embedded within machine instructions or in data declaration statements (e.g., to initialize tables); for example mov(@length(someString), eax); automatically substitutes the length of a compile-time string constant as the source operand of this instruction, the program reflects any changes made to the string by adjusting the value of the source operand each time you recompile the program. However, we haven t looked at how you can actually select which statements in your program to compile or how to iteratively process statements (that is, inject multiple copies of a statement into a program).

Why would you want to do this? Well, consider a data table with 2,000 elements where each entry of the table is initialized with it s index plus the sum of the previous two elements (assuming elements before zero contain the value zero). You could manually create this table in HLA as follows:

```
uns32 0, 1, 3, 7, 14, 26, ...
```

Of course, filling in a table like this one, with 2,000 elements, is laborious. Furthermore, chances are pretty good you d make a mistake somewhere along the line (quick, can you spot any mistakes in this example?). Fortunately, generating a table like this one is very easy to do by writing a short HLA compile-time program. Writing such HLA compile time programs can save you considerable effort, help you write correct code, and make it easier to verify the correctness of your programs. So let s take a look at some of the statements in the HLA compile-time language that allow you to do this.

2.5.4: Conditional Assembly (#if..#elseif..#else..#endif)

HLA s compile-time language provides an if statement that allows you to decide, at compile-time, whether certain statements will be present in the actual object code. Many languages (including various assemblers) provide this facility and they call it conditional assembly or conditional compilation . The conditional compilation statements in HLA use the following syntax:

```
#if( constant_boolean_expression )

   << Statements to compile if the >>
   << expression above is true.    >>

#elseif( constant_boolean_expression )

   << Statements to compile if the >>
   << expression immediately above >>
   << is true and the first expres->>
   << sion above is false.         >>

#else

   << Statements to compile if both    >>
   << the expressions above are false. >>

#endif
```

The `#elseif` and `#ELSE` clauses are optional. Like the HLA run-time `elseif` clause, there may be more than one `#elseif` clause in the same conditional `#if` sequence.

Unlike some other assemblers and high level languages, HLAs conditional compilation directives are legal anywhere whitespace is legal. You could even embed them in the middle of an instruction! While directly embedding these directives in an instruction isn t recommended (because it would make your code very hard to read), it s nice to know that you can place these directives in a macro and then replace an instruction operand with a macro invocation.

The constant expression in the `#if` and `#elseif` clauses must be of type boolean or HLA will emit an error. Any legal compile-time expression that produces a boolean result is legal here. Keep in mind that conditional compilation directives are executed at compile-time, not at run-time. You would not use these directives to (attempt to) make decisions while your program is actually running.

Programmers use conditional assembly for a variety of purposes in HLA programs. A very common use of conditional assembly is to compile different code sequences based on the environment in which the program is going to run. For example, some programmers use conditional assembly to embed non-portable code sequences into their programs and then they select between various code sequences by some boolean expression. For example, HLA programmers who are writing code that is to be portable between Linux and Windows might use conditional compilation sequences like the following:

```
   #if( os.win32 ) // os.win32 and os.linux are boolean constants provided by the
                   // HLA standard library that specifies the OS you're using.

      << code that is Windows specific >>

   #elseif( os.linux )

      << code that is Linux specific >>

   #endif
```

Of course, this book isn t dealing with code that could run under Windows or Linux, so this feature is probably of little use to the average reader of this book. However, this same idea is useful for writing code that makes advantage of features available only in certain versions of Windows. For example, if you want to make an API call that is available only in Windows XP or later, but you still want to be able to create a version of your application that runs under earlier versions of Windows, you can use this same trick, e.g.,

```
#if( WinXPorLater )

   << Code that uses WinXP-specific API functions >>

#else

   << code that deals with the fact that this API is missing >>

#endif
```

In this example, `WinXPorLater` is a compile-time boolean value that you ve defined in your program. A programmer will typically set this variable true or false depending upon whether they want to compile a version of their program for WinXP or an earlier version of Windows.

A traditional trick, inherited from the C/C++ programming community, is to test to see whether a symbol is defined or not defined to determine if code should be assembled one way or another. This technique is easy to use in HLA by employing the `@defined` compile-time variable, e.g.,

```
#if( @defined( WinXPorLater ))

   << Code that uses WinXP-specific API functions >>

#else

   << code that deals with the fact that this API is missing >>

#endif
```

Note that `WinXPorLater's` type and value are irrelevant. In fact, `WinXPorLater` doesn t even have to be a constant or compile-time variable (i.e., you can use any legal HLA identifier here, the `@defined` function returns true if the symbol is defined, false if it is not defined, and it completely ignores the other attributes the symbol may possess). HLA programmers who commonly use `@defined` in this manner generally define such symbols near the beginning of their programs and, although the type and value is irrelevant, they tend to define such compile-time variables as boolean with the value true.

Specifically to support the `@defined` compile-time function, the HLA provides a compile-time option that lets you define symbols. This command option takes the following form:

-d*SymbolToDefine*

This creates an HLA `const` definition for `SymbolToDefine`, setting the type to `boolean` and the value to false.

Here s a suggestion if you re going to use HLA s command-line feature to support compilation options via conditional assembly. Rather than use `@defined` all over your code, just put a short compile-time sequence like the following near the beginning of your source file:

```
#if( !@defined( SymbolToDefine ))
   ?SymbolToDefine := false;
#endif
```

This sequence guarantees that `SymbolToDefine` is defined in your source file. It s value (true or false) determines how conditional assembly will take place later on by simply using the `SymbolToDefine` symbol as a boolean expression in your code. The advantage of using this technique, rather than just using `@defined`, is that you can turn the feature on and off throughout the source code by injecting compile-time assignments like ?SymbolToDefine := true; and SymbolToDefine := false; into your source file.

Of course, conditional assembly has far more uses than simply letting you incorporate different program options in the source file that you can control via simple boolean expressions. Conditional assembly, of course, is what you use to make decisions in an HLA compile-time program. The examples thus far have used conditional assembly to select between one of several sequences of instructions to compile into the final run-time program. In fact, many HLA `#if..#endif` sequences don t contain any code emitting statements at all, instead they just contain sequence of other HLA compile-time language statements. Combined with HLA s compile-time functions, it s quite possible to do some very sophisticated processing. Here s a simple example that might be used to prevent problems when generating a table of log values:

```
#if( r > 0 )
   ?logVal := @log( r );
#else
   #error( "Illegal value passed to @log" )
#endif
```

Here s another example that you might find in a macro that tests a parameter s type to determine how to process the parameter:

```
#if( @typename( macroParm ) = "int32" )
   << do something if the macro parameter is a 32-bit integer >>
#elseif( @typename( macroParm ) = "int16" )
   << do something if the macro parameter is a 16-bit integer >>
#elseif( @typename( macroParm ) = "int88" )
   << do something if the macro parameter is an 8-bit integer >>
#else
   #error( "Expected an int32, int16, or int8 parameter" )
#endif
```

To see some complex examples of HLA s `#if..#elseif..#else..#endif` statement in action, check out the source code to many of the HLA Standard Library routines (especially in several of the HLA Standard Library header files). The HLA Standard Library stdout.put macro is an extreme example that you might want to take a look at.

2.5.5: The #for..#endfor Compile-Time Loop

The `#for..#endfor` loop can take one of the following forms:

```
#for( loop_control_var := Start_expr to end_expr )

   << Statements to execute as long as the loop control variable's >>
   << value is less than or equal to the ending expression.        >>

#endfor

#for( loop_control_var := Start_expr downto end_expr )
```

```
   << Statements to execute as long as the loop control variable’s >>
   << value is greater than or equal to the ending expression.     >>

#endfor
```

The HLA compile-time `#for..#endfor` statement is very similar to the for loops found in languages like Pascal and BASIC. This is a definite loop that executes some number of times determine when HLA first encounters the `#for` directive (this can be zero or more times, but HLA computes the number only once when HLA first encounters the `#for`). The loop control variable must be a `val` object or an undefined identifier (in which case, HLA will create a new `val` object with the specified name). Also, the number control variable must be an eight, sixteen, or thirty-two bit integer value (`uns8, uns16, uns32, int8, int16,` or `int32`). Also, the starting and ending expressions must be values that an `int32 val` object can hold.

The `#for` loop with the `to` clause initializes the loop control variable with the starting value and repeats the loop as long as the loop control variable s value is less than or equal to the ending expression s value. The `#for..to..#endfor` loop increments the loop control variable on each iteration of the loop.

The `#for` loop with the `downto` clause initializes the loop control variable with the starting value and repeats the loop as long as the loop control variable s value is greater than or equal to the ending expression s value. The `#for..downto..#endfor` loop decrements the loop control variable on each iteration of the loop.

Note that the `#for..to/downto..#endfor` loop only computes the value of the ending expression once, when HLA first encounters the `#for` statement. If the components of this expression would change as a result of the execution of the `#for` loop s body, this will not affect the number of loop iterations. If you need this capability, you will need to use HLA s compile-time indefinite loop (the `#while` loop, see the next section).

The `#for..#endfor` loop can also take the following form:

```
#for( loop_control_var in composite_expr )

   << Statements to execute for each element present in the expression >>

#endfor
```

The *`composite_expr`* in this syntactical form may be a string, a character set, an array, or a record constant.

This particular form of the `#for` loop repeats once for each item that is a member of the composite expression. For strings, the loop repeats once for each character in the string and the loop control variable is set to each successive character in the string. For character sets, the loop repeats for each character that is a member of the set; the loop control variable is assigned the value of each character found in the set (you should assume that the extraction of characters from the set is arbitrary, even though the current implementation extracts them in order of their ASCII codes). For arrays, this `#for` loop variant repeats for each element of the array and assigns each successive array element to the loop control variable. For record constants, the `#for` loop extracts each field and assigns the fields, in turn, to the loop control variable.

Examples:

```
   #for( c in "Hello" )
      #print( c )  // Prints the five characters ’H’, ’e’, ..., ’o’
   #endfor

   // The following prints a..z and 0..9 (not necessarily in that order):

   #for( c in {’a’..’z’, ’0’..’9’} )
      #print( c )
   #endfor

   // The following prints 1, 10, 100, 1000

   #for( i in [1, 10, 100, 1000] )
      #print( i )
   #endfor

   // The following prints all the fields of the record type r
   // (presumably, r is a record type you've defined elsewhere):

   #for( rv in r:[0, 'a', "Hello", 3.14159] )
      #print( rv )
   #endfor
```

The HLA compile-time `#for` loop is really useful for processing variable parameter lists in a macro. HLA creates an array of strings with each array element containing the text for each macro parameter. By using this latter form of the `#for` loop you can process each element of the macro in order. You ll see this feature used when we discuss HLAs macro facilities in a few sections.

One very useful purpose for the `#for..#endfor` loop is to construct data tables at compile time. For example, suppose you want to create a table of sine values for each of the angles in the range 0..359 degrees. You could easily do this in HLA as follows:

```
static
   sineTable: real32[ 360 ]; @nostorage;
      #for( angle := 0 to 359 )

         // Note: the HLA compile-time @sin function requires an angle in radians.
         // Conversion from degrees to radians is via the formula:
         // radians = angle * 2 * pi / 360

         real32 @sin( angle * 3.14159 / 180.0 );

      #endfor
```

HLAs `#for..#endfor` compile-time statement is sufficiently powerful that you can generate almost any type of data table you need at compile-time without having to enter in the data for the entire table yourself (or compute the table entries at run-time, consuming both space for the machine instructions and time to execute those instructions). For the few cases where the `#for` loop isn t entirely appropriate, you ll probably want to consider using the HLA `#while` loop that the next section describes.

2.5.6: The #while..#endwhile Compile-Time Loop

HLA provides an indefinite looping mechanism in the compile-time language via the `#while..#endwhile` loop. The `#while..#endwhile` compile-time loop takes the following form:

```
#while( constant_boolean_expression )

   << Statements to emit repeatedly as long >>
   << as the expression is true.            >>

#endwhile
```

While processing the #while..#endwhile loop, HLA evaluates the constant boolean expression. If it is false, HLA immediately skips to the first statement beyond the #endwhile directive.

If the expression is true, then HLA proceeds to compile the body of the #while loop. Upon encountering the #endwhile directive, HLA jumps back up to the #while clause in the source code and repeats this process until the expression evaluates false.

Warning: since HLA allows you to create loops in your source code that evaluation during the compilation process, HLA also allows you to create *infinite* loops that will lock up the system during compilation. If HLA seems to have gone off into la-la land during compilation and you re using #while loops in your code, it might not be a bad idea to put some #print directives[9] into your loop(s) to see if you ve created an infinite loop.

The HLA #while..#endwhile loop is great for repeatedly emitting sections of code whose loop control expression varies while processing the loop (unlike the #for..#endfor loop, for which HLA can compute the number of iterations when HLA first encounters the #for clause). Though nowhere near as popular in modern HLA code as the #for..#endfor loop, the #while..#endwhile loop is still quite useful in many situations. The #while..#endwhile loop, for example, is quite useful when you want to increment or decrement a non-integer loop control variable, e.g.,

```
? r :real64 := 10.2;
#while( r > 0 )
   << Statements to process while r is greater than zero >>
   ? r := r - 0.1;
#endwhile
```

If you look at some older HLA code (e.g., in the HLA Standard Library), you find the #while..#endwhile loop used in many cases where a #for..#endfor loop might be more appropriate. This is because the #for..#endfor loop is a fairly recent addition to the HLA language, appearing long after the creation of the HLA Standard library.

2.5.7: Compile-Time I/O and Data Facilities

HLAs compile-time language provides several facilities for printing messages during compilation, reading and writing text file data, executing system commands, and processing blocks of text as string data in the HLA source file. Though some of these features are rather esoteric, they are quite useful in some circumstances. We ll take a look at many of these HLA features in this section.

The #system directive requires a single string parameter. It executes this string as an operating system (shell/ command interpreter) operation via the C system function call. This call is useful, for example, to run a program during compilation that dynamically creates a text file that an HLA program may include immediately after the #system invocation.

Example of #system usage:

```
#system( "dir" )
```

Note that the #system directive is legal anywhere white space is allowable and doesn t require a semicolon at the end of the statement.

9. These will be described a little later in this chapter.

Do keep in mind that HLA does not have any control over the programs you run via the #system compile-time language statement. It is your responsibility to ensure that you don t run some program that interferes with the current assembly in process (e.g., deleting the current source file that HLA is compiling would be an example of interference that might leave the current compilation in an undefined state).

The `#print` directive displays its parameter values during compilation. The basic syntax is the following:

```
#print( comma, separated, list, of, constant, expressions, ... )
```

The `#print` statement is very useful for displaying messages during assembly (e.g., when debugging complex macros or compile-time programs). The items in the `#print` list must evaluate to constant (`const` or `val`) values at compile time. This directive is great for displaying status information during compilation. It s also great for tracking down bugs in a compile-time program (e.g., if HLA hangs up and you suspect that this is due to an infinite `#while` loop in your code, you can use `#print` to track down exactly where HLA is hanging up).

The `#error` directive behaves like `#print` insofar as it prints its parameter to the console device during compilation. However, this instruction also generates an HLA error message and does not allow the creation of an object file after compilation. This statement only allows a single string expression as a parameter. If you need to print multiple values of different types, use string concatenation and the `@string` function to achieve this. Example:

```
#error( "Error, unexpected value.  Value = " + #string( theValue ))
```

Notice that neither the `#print` nor the `#error` statements end with a semicolon.

The `#openwrite`, `#write`, and `#closewrite` compile-time statements let you do simple file output during compilation. The `#openwrite` statement opens a single file for output, `#write` writes data to that output file, and `#closewrite` closes the file when output is complete. These statements are useful for automatically generating *include* files that the source file will include later on during the compilation. These statements are also useful for storing bulk data for later retrieval or generating a log during assembly.

The `#openwrite` statement uses the following syntax:

#openwrite(*string_expression*)

This call opens a single output file using the filename specified by the string expression. If the system cannot open the file, HLA emits a compilation error. Note that `#openwrite` only allows one output file to be active at a time. HLA will report an error if you execute `#openwrite` and there is already an output file open. If the file already exists, HLA deletes it prior to opening it (so be careful!). If the file does not already exist, HLA creates a new one with the specified name.

The `#write` statement uses the same syntax as the `#print` directive. Note, however, that `#write` doesn t automatically emit a newline after writing all its operands to the file; if you want a newline output you must explicitly supply it as the last parameter to `#write`.

The `#closewrite` statement closes the file opened via `#openwrite`. HLA automatically closes this file at the end of assembly if you leave it open. However, you must explicitly close this file before attempting to use the data (via `#include` or `#openread`) in your program. Also, since HLA allows only one open output file at a time, you must use `#closewrite` to close the file before you can open another with `#openwrite`.

Here is an example of the `#openwrite`, `#write`, and `#closewrite` statements in action. This short compile-time program creates an array of integer values for inclusion elsewhere in the source file:

```
    #openwrite( “myDataInclude.hhf” )
    #write( “[ “ )
```

```
   #for( i := 0 to 2047 )

      #write( i )
      #if( i <> 2047 )
         #write( ", " )
      #elseif( ( i mod 16) == 15 )
         #write( nl )
      #endif
   #endfor
   #write( "];" )
   #closewrite
      .
      .
      .
const
   constArray :dword[ 2048] :=
      #include( "myDataInclude.hhf" )
```

Of course, initializing an array in this manner is rather silly, because we could have (more easily) initialized the array directly with an HLA compile-time program (e.g., the #for loop above with a few minor modifications would have been sufficient). However, it s easy enough to imagine building up the include file at various points throughout the code, thus making the use of `#openwrite/#write/#closewrite` more applicable.

The `#openread, @read,` and `#closeread` compile-time statements and function let you do simple file input during compilation. The `#openread` statement opens a single file for input, `@read` is a compile-time function that reads a line of text from the file, and `#closeread` closes the file when input is complete. These statements are useful for reading files produced by `#openwrite/#write/#closewrite` during compilation, or any other text file for that matter.

The `#openread` statement uses the following syntax:

#openread(*filename*)

The *`filename`* parameter must be a string expression or HLA reports an error. HLA attempts to open the specified file for reading; HLA prints an error message if it cannot open the file.

The `@read` function uses the following call syntax:

@read(*val_object*)

The *`val_object`* parameter must either be a symbol you ve defined in a `val` section (or via ?) or it must be an undefined symbol (in which case `@read` defines it as a `val` object). `@read` is an HLA compile-time function (hence the `@` prefix rather than `#` ; HLA uses `#` for compile-time statements). It returns either true or false, true if the read was successful, false if the read operation encountered the end of file. Note that if any other read error occurs, HLA will print an error message and return false as the function result. If the read operation is successful, then HLA stores the string it read (up to 4095 characters) into the `val` object specified by the parameter. Unlike `#openread` and `#closeread`, the `@read` function may not appear at an arbitrary point in your source file. It must appear within a constant expression since it returns a boolean result (and it is your responsibility to check for EOF).

The `#closeread` statement closes the input file. Since you may only have one open input file at a time, you must close an open input file with `#closeread` prior to opening a second file. Syntax:

#closeread

Example of using compile-time file I/O:

```
#openwrite( "hw.txt" )
#write( "Hello World", nl )
#closewrite
#openread( "hw.txt" )
?goodread := @read( s );
#closeread
#print( "data read from file = ", s )
```

The `#text` and `#endtext` directives surround a block of text in an HLA program from which HLA will create an array of string constants. The syntax for these directives is:

```
   #text( identifier )

      << arbitrary lines of text >>

   #endtext
```

The *`identifier`* must either be an undefined symbol or an object declared in the `val` section.

This directive converts each line of text between the `#text` and `#endtext` directives into a string and then builds an array of strings from all this text. After building the array of strings, HLA assigns this array to the identifier symbol. This is a `val` constant array of strings. The `#text..#endtext` directives may appear anywhere in the program where white space is allowed.

These directives provide an easy way to initialize a constant array of strings and they provide a convenient alternative to using `#openread/@read/#closeread` when you simply want to inject a fair amount of textual data into your HLA source file for further processing by the compile-time language. Here is an example that uses the `#text..#endtext` to initialize an array of strings in a program.

```
#text( constStringArray )
This goes into the first string.
This line goes into the second string.
The third string will contain this line.
And so on...
#endtext

static
   strArray :string[] := constStringArray;
```

By using `#text..#endtext` or the `#openread..@read..#closeread` compile-time statements, it is actually possible to create your own languages using the HLA compile-time language. Processing the text appearing in a #text..#endtext block, or the text read from a file via the `#openread..@read..#closeread` statements is one place where HLA s compile-time string and pattern matching functions come in real handy.

2.5.8: Macros (Compile-Time Procedures and Functions)

HLA provides one of the most sophisticated macro processing facilities of any assembler, indeed, any language, available. Macros are a great tool for Win32 programmers because a considerable amount of Win32 programming is the repetitive application of some common programming template. Learning how to properly use macros can help you reduce the drudgery often associated with assembly language programming. Unfortunately, HLA s macro sophistication comes with a price tag: if you want to learn the macros inside and out, there is a steep learning curve associated with them. Fortunately, you don t have to learn everything there is to know about

HLA macros in order to effectively employ them in your source files. This section will concentrate on those features of the HLA macro subsystem that a Win32 programmer will commonly use.

You can declare macros almost anywhere HLA allows whitespace in a program using the following syntax:

```
#macro identifier ( optional_parameter_list ) ;
   statements
#endmacro
```

Note that a semicolon does not follow the `#endmacro` clause.

The optional parameter list must be a list of one or more identifiers separated by commas. Unlike procedure declarations, you do not associate a type with macro parameters. HLA automatically associates the type text with all macro parameters (except for one special case noted below). Example:

```
   #macro MacroWParms( a, b, c );
      ?a = b + c;
   #endmacro
```

Optionally, the last (or only) name in the identifier list may take the form `identifier[]`. This syntax tells the macro that it may allow a variable number of parameters and HLA will create an array of string objects to hold all the parameters (HLA uses a string array rather than a text array because text arrays are illegal).

Example:

```
   #macro MacroWVarParms( a, b, c[] );
      ?a = b + text(c[0]) + text(c[1]);
   #endmacro
```

If the macro does not allow any parameters, then you follow the identifier with a semicolon (i.e., no parentheses or parameter identifiers). See the first example in this section for a macro without any parameters.

Occasionally you may need to define some symbols that are local to a particular macro invocation (that is, each invocation of the macro generates a unique symbol for a given identifier). The local identifier list allows you to do this. To declare a list of local identifiers, simply following the parameter list (after the parenthesis but before the semicolon) with a colon (:) and a comma separated list of identifiers, e.g.,

```
      #macro ThisMacro(parm1):id1,id2;
      ...
```

HLA automatically renames each symbol appearing in the local identifier list so that the new name is unique throughout the program. HLA creates unique symbols of the form `_XXXX_` where `XXXX` is some hexadecimal numeric value. To guarantee that HLA can generate unique symbols, you should avoid defining symbols of this form in your own programs (in general, symbols that begin and end with an underscore are reserved for use by the compiler and the HLA standard library).   Example:

```
   #macro LocalSym : i,j;

   j: cmp(ax, 0)
      jne( i )
      dec( ax )
      jmp( j )
   i:
   #endmacro
```

Without the local identifier list, multiple expansions of this macro within the same procedure would yield multiple statement definitions for i and j . With the local statement present, however, HLA substitutes symbols similar to _0001_ and _0002_ for i and j for the first invocation and symbols like _0003_ and _0004_ for i and j on the second invocation, etc. This avoids duplicate symbol errors if you do not use (poorly chosen) identifiers like _0001_ and _0004_ in your code.

The statements section of the macro may contain any legal HLA statements (including definitions of other macros). However, the legality of such statements is controlled by where you expand the macro.

To invoke a macro, you simply supply its name and an appropriate set of parameters. Unless you specify a variable number of parameters (using the array syntax) then the number of actual parameters must exactly match the number of formal parameters. If you specify a variable number of parameters, then the number of actual parameters must be greater than or equal to the number of formal parameters (not counting the array parameter).

During macro expansion, HLA automatically substitutes the text associated with an actual parameter for the formal parameter in the macro s body. The array parameter, however, is a string array rather than a text array so you will have force the expansion yourself using the @text function:

```
#macro example( variableParms[] );
   ?@text(variableParms[0]) := @text(variableParms[1]);
#endmacro
```

Actual macro parameters consist of a string of characters up to, but not including a separate comma or the closing parentheses, e.g.,

```
example( v1, x+2*y )
```

v1 is the text for parameter #1, x+2*y is the text for parameter #2. Note that HLA strips all leading whitespace and control characters before and after the actual parameter when expanding the code in-line. The example immediately above would expand do the following:

```
?v1 := x+2*y;
```

If (balanced) parentheses appear in some macro s actual parameter list, HLA does not count the closing parenthesis as the end of the macro parameter. That is, the following is perfectly legal:

```
example(  v1, ((x+2)*y) )
```

This expands to:

```
?v1 := ((x+2)*y);
```

If you need to embed commas or unmatched parentheses in the text of an actual parameter, use the HLA literal quotes #(and)# to surround the text. Everything (except surrounding whitespace) inside the literal quotes will be included as part of the macro parameter s text. Example:

```
example( v1, #( array[0,1,i] )# )
```

The above expands to:

```
?v1 := array[ 0,1,i];
```

Without the literal quote operator, HLA would have expanded the code to

```
?V1 := array[ 0;
```

and then generated an error because (1) there were too many actual macro parameters (four instead of two) and (2) the expansion produces a syntax error.

Of course, HLAs macro parameter parser does not consider commas appearing inside string or character constants as parameter separators. The following is perfectly legal, as you would expect:

```
example( charVar, ',' )
```

As you may have noticed in these examples, a macro invocation does not require a terminating semicolon. Macro expansion occurs upon encountering the closing parenthesis of the macro invocation. HLA uses this syntax to allow a macro expansion *anywhere* in an HLA source file. Consider the following:

```
#macro funny( dest )
   , dest );
#endmacro

   mov( 0 funny( ax )
```

This code expands to `mov( 0, ax );` and produces a legal machine instruction. Of course, the this is a truly horrible example of macro use (the style is really bad), but it demonstrates the power of HLA macros in your program. This expand anywhere philosophy is the primary reason macro invocations do not end with a semicolon.

HLA macros provide some very powerful facilities not found in other macro assemblers. One of the really unique features that HLA macros provides is support for multi-part (or context-free) macro invocations. This feature is accessed via the `#terminator` and `#keyword` reserved words. Consider the following macro declaration:

```
program demoTerminator;

#include( "stdio.hhf" );

#macro InfLoop:TopOfLoop, LoopExit;
   TopOfLoop:
#terminator endInfLoop;
   jmp TopOfLoop;
   LoopExit:
#endmacro;

static
   i:int32;

begin demoTerminator;

   mov( 0, i );
   InfLoop

      stdout.put( "i=", i, nl );
      inc( i );
```

```
    endInfLoop;

end demoTerminator;
```

The `#terminator` keyword, if it appears within a macro, defines a second macro that is available for a one-time use after invoking the main macro. In the example above, the `endInfLoop` macro is available only after the invocation of the `InfLoop` macro. Once you invoke the `EndInfLoop` macro, it is no longer available (though the macro calls can be nested, more on that later). During the invocation of the `#terminator` macro, all local symbols declared in the main macro (`InfLoop` above) are available (note that these symbols are not available outside the macro body. In particular, you could not refer to either `TopOfLoop` nor `LoopExit` in the statements appearing between the `InfLoop` and `endInfLoop` invocations above). The code above, by the way, emits code similar to the following:

```
_0000_:
        stdout.put( "i=", i, nl );
        inc( i );
        jmp _0000_;
_0001_:
```

The macro expansion code appears in italics. This program, therefore, generates an infinite loop that prints successive integer values.

These macros are called multi-part macros for the obvious reason: they come in multiple pieces (note, though, that HLA only allows a single `#terminator` macro). They are also referred to as *Context-Free macros* because of their syntactical nature. Earlier, this document claimed that you could refer to the `#terminator` macro only once after invoking the main macro. Technically, this should have said you can invoke the terminator once for each outstanding invocation of the main macro. In other words, you can nest these macro calls, e.g.,

```
    InfLoop

        mov( 0, j );
        InfLoop

            inc( i );
            inc( j );
            stdout.put( "i=", i, " j=", j, nl );

        endInfLoop;

    endInfLoop;
```

The term *Context-Free* comes from automata theory; it describes this nestable feature of these macros.

As should be painfully obvious from this `InfLoop` macro example, it would be really nice if one could define more than one macro within this context-free group. Furthermore, the need often arises to define limited-scope scope macros that can be invoked more than once (limited-scope means between the main macro call and the terminator macro invocation). The `#keyword` definition allows you to create such macros.

In the `InfLoop` example above, it would be really nice if you could exit the loop using a statement like `brkLoop` (note that `break` is an HLA reserved word and cannot be used for this purpose). The `#keyword` section of a macro allows you to do exactly this. Consider the following macro definition:

```
#macro InfLoop:TopOfLoop, LoopExit;
   TopOfLoop:
#keyword brkLoop;
   jmp LoopExit;
#terminator endInfLoop;
   jmp TopOfLoop;
   LoopExit:
#endmacro;
```

As with the `#terminator` section, the `#keyword` section defines a macro that is active after the main macro invocation until the terminator macro invocation. However, `#keyword` macro invocations to not terminate the multi-part invocation. Furthermore, `#keyword` invocations may occur more that once. Consider the following code that might appear in the main program:

```
   mov( 0, i );
   InfLoop

      mov( 0, j );
      InfLoop

         inc( j );
         stdout.put( "i=", i, " j=", j, nl );
         if( j >= 10 ) then

            brkLoop;

         endif

      endInfLoop;
      inc( i );
      if( i >= 10 ) then

         brkLoop;

      endif;

   endInfLoop;
```

The `brkLoop` invocation inside the `if( j >= 10)` statement will break out of the inner-most loop, as expected (another feature of the context-free behavior of HLAs macros). The `brkLoop` invocation associated with the `if( i >= 10 )` statement breaks out of the outer-most loop. Of course, the HLA (run-time) language provides the `forever..endfor` loop and the `break` and `breakif` statements, so there is no need for this `InfLoop` macro, nevertheless, this example is useful because it is easy to understand. If you are looking for a challenge, try creating a statement similar to the C/C++ `switch/case` statement; it is perfectly possible to implement such a statement with HLAs macro facilities, see the HLA Standard Library for an example of the `switch` statement implemented as a macro.

The discussion above introduced the `#keyword` and `#terminator` macro sections in an informal way. There are a few details omitted from that discussion. First, the full syntax for HLA macro declarations is actually:

```
#macro identifier ( optional_parameter_list ) :optional_local_symbols;
   statements
```

```
#keyword identifier ( optional_parameter_list ) :optional_local_symbols;
   statements

note: additional #keyword declarations may appear here

#terminator identifier ( optional_parameter_list ) :optional_local_symbols;
   statements
#endmacro
```

There are three things that should immediately stand out here: (1) You may define more than one `#keyword` within a macro. (2) `#keywords` and `#terminators` allow optional parameters. (3) `#keyword` and `#terminator` allow their own local symbols.

The scope of the parameters and local symbols isn t particularly intuitive (although it turns out that the scope rules are exactly what you would want). The parameters and local symbols declared in the main macro declaration are available to all statements in the macro (including the statements in the `#keyword` and `#terminator` sections). The `InfLoop` macro used this feature because the `jmp` instructions in the `brkLoop` and `endInfLoop` sections referred to the local symbols declared in the main macro. The `InfLoop` macro did not declare any parameters, but had they been present, the `brkLoop` and `endInfLoop` sections could have used those symbols as well.

Parameters and local symbols declared in a `#keyword` or `#terminator` section are local to that particular section. In particular, parameters and/or local symbols declared in a `#keyword` section are not visible in other `#keyword` sections or in the `#terminator` section.

One important issue is that local symbols in a multipart macro are visible in the main code between the start of the multipart macro and the terminating macro. That is, if you have some sequence like the following:

```
    InfLoop

       jmp LoopExit;

    endInfLoop;
```

Then HLA substitutes the appropriate internal symbol (e.g., `"_xxxx_"`) for the `LoopExit` symbol. This is somewhat unintuitive and might be considered a flaw in HLAs design. Future versions of HLA may deal with this issue; in the meantime, however, some code takes advantage of this feature (to mask global symbols) so it s not easy to change without breaking a lot of code. Be forewarned before taking advantage of this feature , however, that it will probably change in HLA v2.x. An important aspect of this behavior is that macro parameter names are also visible in the code section between the initial macro and the terminator macro. Therefore, you must take care to choose macro parameter names that will not conflict with other identifiers in your program. E.g., the following will probably lead to some problems:

```
static
   i:int32;

#macro parmi(i);
   mov( i, eax );
#terminator endParmi;
   mov( eax, i );
#endmacro
   .
   .
   .
   parmi( xyz );
   mov( i, ebx ); // actually moves xyz into ebx, since the parameter i
                  // overrides the global variable i here.
   endParmi;
```

As mentioned earlier, HLA treats all non-array macro parameters as text constants that are assigned a string corresponding to the actual parameter(s) passed to the macro. I.e., consider the following:

```
#macro SetI( v );
   ?i := v;
#endmacro

SetI( 2 );
```

The above macro and invocation is roughly equivalent to the following:

```
const
   v : text := "2";
   ?i := v;
```

When utilizing variable parameter lists in a macro, HLA treats the parameter object as a string array rather than a text array (because HLA v1.x does not currently support text arrays). For example, consider the following macro and invocation:

```
#macro SetI2( v[] );
   ?i := v[0];
#endmacro

SetI2( 2 );
```

Although this looks quite similar to the previous example, there is a subtle difference between the two. The former example will initialize the constant (value) `i` with the `int32` value two. The second example will initialize `i` with the string value 2 .

If you need to treat a macro array parameter as text rather than as a string object, use the HLA `@text` function that expands a string parameter as text. E.g., the former example could be rewritten as:

```
#macro SetI2( v[] );
   ?i := @text( v[0]);
#endmacro

SetI2( 2 );
```

In this example, the `@text` function tells HLA to expand the string value `v[0]` (which is 2) directly as text, so the "`SetI2( 2 )`" invocation expands as

```
?i := 2;
rather than as
?i := "2";
```

On occasion, you may need to do the converse of this operation. That is, you may want to treat a standard (non-array) macro parameter as a string object rather than as a text object. Unfortunately, text objects are expanded by the lexer in-line upon initial processing; the compiler never sees the text variable name (or parameter name, in this particular case). To overcome this problem, the lexer has a special feature to avoid expanding text constants if they appear inside an `@string` compile-time function. The following example demonstrates one possible use of this feature:

```
program demoString;

#macro seti3( v );
   #print( "i is being set to " + @string( v ))
   ?i := v;
#endmacro


begin demoString;

   seti3( 4 )
   #print( "i = " + string( i ) )
   seti3( 2 )
   #print( "i = " + string( i ) )

end demoString;
```

If an identifier is a `text` constant (e.g., a macro parameter or a const/value identifier of type `text`), special care must be taken to modify the string associated with that text object. A simple `val` expression like the following won t work:

?textVar:text := "SomeNewText";

The reason this doesn t work is subtle: if `textVar` is already a text object, HLA immediately replaces `textVar` with its corresponding string; this includes the occurrence of the identifier immediately after the ? in the example above. So were you to execute the following two statements:

```
?textVar:text := "x";
?textVar:text := "1";
```

the second statement would not change `textVar's` value from x to 1 . Instead, the second statement above would be converted to:

```
?x:text := "1";
```

and `textVar's` value would remain x . To overcome this problem, HLA provides a special syntactical entity that converts a text object to a string and then returns the text object ID. The syntax for this special form is "`@tostring:identifier`". The example above could be rewritten as:

```
?textVar:text := "x";
?@tostring:textVar:text := "1";
```

In this example, `textVar` would be a text object that expands to the string 1 .

As described earlier, HLA processes as parameters all text between a set of matching parentheses after the macro s name in a macro invocation. HLA macro parameters are delimited by the surrounding parentheses and commas. That is, the first parameter consists of all text beyond the left parenthesis up to the first comma (or up to the right parenthesis if there is only one parameter). The second parameter consists of all text just beyond the first comma up to the second comma (or right parenthesis if there are only two parameters). Etc. The last parameter consists of all text from the last comma to the closing right parenthesis.

Note that HLA will strip away any white space at the beginning and end of the parameter s text (though it does not remove any white space from the interior of the parameter s text).

If a single parameter must contain commas or parentheses, you must surround the parameter with the literal text macro quotes `#(` and `)#`. HLA considers everything but leading and trailing space between these macro quote symbols as a single parameter. Note that this applies to macro invocations appearing within a parameter list. Consider the following (erroneous) code:

```
CallToAMacro( 5, "a", CallToAnotherMacro( 6,7 ), true );
```

Presumably, the (6,7) text is the parameter list for the `CallToAnotherMacro` invocation. When HLA encounters a macro invocation in a parameter list, it defers the expansion of the macro. That is, the third parameter of `CallToAMacro` should expand to `CallToAnotherMacro( 6,7 )`, not the text that `CallToAnother-Macro` would expand to. Unfortunately, this example will not compile correctly because the macro processor treats the comma between the 6 and the 7 as the end of the third parameter to `CallToAMacro` (in other words, the third parameter is actually `CallToAnotherMacro( 6` and the fourth parameter is `7 )`. If you really need to pass a macro invocation as a parameter, use the `#(` and `)#` macro quotes to surround the interior invocation:

CallToAMacro(5, a , #(CallToAnotherMacro(6,7))#, true);

In this example, HLA passes all the text between the `#(` and `)#` markers as a single parameter (the third parameter) to the `CallToAMacro` macro.

This example demonstrates another feature of HLAs macro processing system - HLA uses *deferred macro parameter expansion*. That is, the text of a macro parameter is expanded when HLA encounters the formal parameter within the macro s body, *not* while HLA is processing the actual parameters in the macro invocation (which would be *eager* evaluation).

There are three exceptions to the rule of deferred parameter evaluation: (1) text constants are always expanded in an eager fashion (that is, the value of the text constant, not the text constant s name, is passed as the macro parameter). (2) The `@text` function, if it appears in a parameter list, expands the string parameter in an eager fashion. (3) The `@eval` function immediately evaluates its parameter; the discussion of `@eval` appears a little later.

In general, there is very little difference between eager and deferred evaluation of macro parameters. In some rare cases there is a semantic difference between the two. For example, consider the following two programs:

```
program demoDeferred;
#macro two( x, y ):z;
   ?z:text:="1";
   x+y
#endmacro

const
   z:string := "2";

begin demoDeferred;
```

```
   ?i := two( z, 2 );
   #print( "i=" + string( i ))

end demoDeferred;
```

In the example above, the code passes the actual parameter `z` as the value for the formal parameter `x`. Therefore, whenever HLA expands `x` it gets the value `z` which is a local symbol inside the `two` macro that expands to the value `1`. Therefore, this code prints `3` (`1` plus `y's` value which is `2`) during assembly. Now consider the following code:

```
program demoEager;
#macro two( x, y ):z;
   ?z:text:="1";
   x+y
#endmacro

const
   z:string := "2";

begin demoEager;

   ?i := two( @text( z ), 2 );
   #print( "i=" + string( i ))

end demoEager;
```

The only differences between these two programs are their names and the fact that `demoEager` invocation of `two` uses the `@text` function to eagerly expand `z's` text. As a result, the formal parameter `x` is given the value of `z's` expansion (`2`) and HLA ignores the local value for `z` in macro `two`. This code prints the value `4` during assembly. Note that changing `z` in the main program to a text constant (rather than a string constant) has the same effect:

```
program demoEager;
#macro two( x, y ):z;
   ?z:text:="1";
   x+y
#endmacro

const
   z:text := "2";

begin demoEager;

   ?i := two( z, 2 );
   #print( "i=" + string( i ))

end demoEager;
```

This program also prints `4` during assembly.

One place where deferred vs. eager evaluation can get you into trouble is with some of the HLA built-in functions. Consider the following HLA macro:

```
   #macro DemoProblem( Parm );
```

```
      #print( string( Parm ) )

   #endmacro
      .
      .
      .
   DemoProblem( @linenumber );
```

(The `@linenumber` function returns, as an `uns32` constant, the current line number in the file).

When this program fragment compiles, HLA will use deferred evaluation and pass the text `@linenumber` as the parameter `Parm`. Upon compilation of this fragment, the macro will expand to `#print( string( @linenumber ))` with the intent, apparently, being to print the line number of the statement containing the `DemoProblem` invocation. In reality, that is not what this code will do. Instead, it will print the line number, in the macro, of the `#print( string (Parm));` statement. By delaying the substitution of the current line number for the `@linenumber` function call until inside the macro, deferred execution produces the wrong result. What is really needed here is eager evaluation so that the `@linenumber` function expands to the line number string before being passed as a parameter to the `DemoProblem` macro. The `@eval` built-in function provides this capability. The following coding of the `DemoProblem` macro invocation will solve the problem:

```
      DemoProblem( @eval( @linenumber ) );
```

Now the compiler will execute the `@linenumber` function and pass that number as the macro parameter text rather than the string `@linenumber`. Therefore, the `#print` statement inside the macro will print the actual line number of the `DemoProblem` statement rather than the line number of the `#print` statement.

Of course, always having to type `@eval( @linenumber )` whenever you want to pass `@linenumber` as a macro parameter can get rather burdensome. Fortunately, you can create a text constant to ease this problem for you:

```
const
   evalLnNum :text := "@eval( @linenumber )";
      .
      .
      .
   DemoProblem( evalLnNum );
```

Because text constants expand in place (before any evaluation takes place), this code properly substitutes the current line number (of the `DemoProblem` statement) for the macro parameter.

2.5.9: Performance of the HLA Compile-Time Language

The HLA compile-time language was written to save development time by giving the assembly language programmer the ability to automate the creation of long or complex sequences of instructions. Because of HLAs design and the languages used to implement HLA (Flex, Bison, and C), HLA uses a pure interpreter implementation for the compile-time language. This means that HLA processes the text in your source file directly when executing a compile-time program. Unfortunately, pure interpretation is one of the slowest forms of program execution in common use. For the most part, the fact that HLA interprets compile-time language relatively slowly is a moot issue. Modern machines are sufficiently fast that even a pure interpretation scheme won t normally introduce a noticeable delay in the compilation time of your programs. However, the overuse of HLAs compile-time facilities can lead to slower development times, but consuming several seconds (or even minutes in some extreme cases) to interpret all the compile-time language statements in a source file.

Before discussing this subject any further, it s important for you to realize that we are talking about compilation times here, not the run-time of your actual application. The speed of HLA s compile-time language has nothing to do with the execution time of an application you write in HLA. So don t confuse the two and be concerned that HLA s compile-time language is somehow slowing down your applications.

As noted, modern machines are sufficiently fast that even HLA s pure interpretation execution model for compile-time programs won t have a tremendously significant impact on the compilation times of your programs. However, don t forget one thing - whenever HLA expands a macro the system isn t processing a single line of text in the source file, it s actually processing every line of text in that macro *for each invocation of the macro*. Likewise, whenever HLA processes a #while loop or a #for loop it isn t simply processing the number of lines in the loop s body, it s processing that number of lines *times the number of loop iterations*. It is very easy to create a very short HLA source file that takes a tremendous amount of time to compile. For example, on a 2GHz Pentium IV machine, the following trivial HLA program requires about a minute to compile:

```
program t;
   #for( i:= 0 to 100_000_000 )
   #endfor
begin t;
end t;
```

The fact that HLA is actually processing 1.7 million lines per second while compiling this program (a phenomenal number) is of little relief to the programmer waiting around a minute for this program to compile. The average programmer sees a short source file with only five statements and questions why the assembler would take nearly a minute to compile this file. They don t see the fact that as far as HLA is concerned, this source file is actually over one hundred million lines long. Compiling 100,000,000 lines of source code in a minute is very impressive. It s just not obvious to most people looking at this source code that this is what is going on.

Though examples like this one are rather rare, it is possible to create HLA macros that consume a fair amount of time, particularly if you invoke those macros many times within a source file. For example, the HLA stdout.put macro is between 400 and 500 source lines long (not counting all the loops present in the macro). If you stick a thousand invocations of stdout.put (each with multiple parameters) into a source file, you will measure compilation time in *minutes*, not seconds, on typical PCs. Counting loops and other features in the stdout.put macro, an invocation of this macro typically requires the interpretation of 500 or so HLA compile-time language statements per macro argument. A source file with 1,000 stdout.put macro invocations can easily expand to between 1,000,000 and 2,000,000 compile-time statements that HLA must interpret (which is slow, much slower than processing the empty `#for` loop in the previous example).

The moral of this story is that if you intend to write (or use) some extremely complex macros, and you intend to invoke that macro many, many, times in your source file, be prepared for slower than usual compilation times. If this loss of performance hinders your development, you might consider using separate compilation and splitting up the macro invocations across as many source files as possible so you don t have to recompile the entire program every time you make a minor change.

2.5.10: A Complex Macro Example: stdout.put

Before concluding this chapter, it would be a good idea to provide an example of a relatively complex HLA compile-time program. Probably the most common example of just such a compile-time program is the HLA Standard Library s `stdout.put` macro. The stdout.put macro is interesting because it parses the list of arguments you supply, determines the type of the argument, extracts optional formatting information (if present), and calls an appropriate HLA Standard Library routine to actually print the value to the standard output device. Using stdout.put is far more convenient than calling all the individual routines in the HLA Standard Library.

Note: although we won t use the stdout.put macro much in this book, it s mainly useful for writing console applications, not GUI apps, we will make use of the `fileio.put` and `str.put` macros which work in a similar fashion to `stdout.put`. Even if these other macros weren t useful to Win32 programmers, the concepts that stdout.put employs are quite useful for Win32-based macros.

To begin with, you should note that the HLA `stdout.put` macro is really the `put` macro that just happens to be defined in the `stdout` name space. We ll continue to call it `stdout.put` in this discussion to differentiate it from other `put` macros that appear in the HLA Standard Library. This macro s definition appears in the *stdout.hhf* header file; Those who would like to see the original source code can find *stdout.hhf* in the HLA include subdirectory.

The `stdout.put` macro allows zero or more arguments. To handle this, the put macro declaration defines a variable parameter list (using HLA s variable parameter list syntax -- an open ended array). In addition to this varying parameter list, the `put` macro uses several local symbols within the macro, they are all part of the `stdout.put` macro declaration:

```
 #macro put( _parameters_[] ):
     _curparm_, _pType_, _arg_, _width_,
     _decpts_, _parmArray_, _id_, _tempid_, _fieldCnt_;
```

In order to process each parameter the caller supplies, the `stdout.put` macro uses a compile-time variable and a `#while` loop to step through each of the elements of the `_parameters_` array. This could actually be done more conveniently with a `#for` loop, but the `stdout.put` macro was written long before the `#for` loop was added to the HLA language, hence the use of the `#while` loop to simulate a `#for` loop:

```
        ?_curparm_:uns32 := 0;


        // The following loop repeats once for each PUT parameter
        // we process.

        #while( _curparm_ < @elements( _parameters_ ))
```

The `stdout.put` macro allows operands to take the following form: `operand`, `operand:n`, or `operand:n:m`. The `n` and `m` items provide a minimum field width (`n`) and positions after the decimal point (`m`, for real values). In order to properly process each argument, HLA needs to split up each parameter into one, two, or three separate strings, depending on the presence of the field width and decimal point options. HLA maintains this information in an array of strings and uses the local symbol `_parmArray_` to hold these strings. On each iteration of the loop, HLA redefines `_parmArray_` as an `uns32` object in order to free the string storage used by the previous iteration of the loop. After doing this, the macro calls the @tokenize compile-time function to break up the current parameter string into various parts (see the *HLA Reference Manual* for a complete description of the `@tokenize` function; for our purposes, just assume that it puts the actual operand name into the first element of the `_parmArray_` array of strings, the field width into the second element, and the decimal point position into the third element).

```
            ?_parmArray_:uns32 := 0;

            ?_parmArray_ := @tokenize
```

```
(
    _parameters_[ _curparm_ ],
    0,
    { ':'},
    {
        '"',
        '''',
        '[ ',
        '] ',
        '(',
        ')',
        '{ ',
        '} '
    }
);
```

Next, the stdout.put macro does some processing on the first portion of the current parameter to determine if we ve got an identifier or an expression. First, this code strips away any leading and trailing spaces and then checks to see if the string begins with an identifier. If so, the code invokes the stdio._GetID_ macro that extracts the identifier from the start of the string.

```
?_arg_ := @trim( _parmArray_[ 0 ], 0 );
#if( char( _arg_ ) in stdio._idchars_ )

    // If this parameter begins with an id character,
    // then strip away any non-ID symbols from the
    // end of the string.  Then determine if we've
    // got a constant or some other class (e.g.,
    // variable or procedure).  If not a constant,
    // keep only the name.  If a constant, we need
    // to keep all trailing characters as well.

    ?_id_ := stdio._GetID_( _arg_ );
    #if
    (
            @class( _id_ ) = hla.cConstant
        |   @class( _id_ ) = hla.cValue
    )

        ?_id_ := _arg_;

    #endif

#else

    // If it's not an ID, we need to keep everything.

    ?_id_ := _arg_;

#endif
```

After extracting the first operand, the next step is to process the optional field width and fractional width components. The `stdout.put` macro determines if these fields are present by checking the number of elements in the `_parmArray_` object. If the number of elements is two or greater, then `stdout.put` assumes that the second array element holds the minimum field width. If the number of elements is three or greater, then `stdout.put` assumes that the third element contains the fractional width information.

```
        // Okay, determine if the caller supplied a field width
        // value with this parameter.

        ?_fieldCnt_ := @elements( _parmArray_ );
        #if( _fieldCnt_ > 1 )

            ?_width_ := @trim( _parmArray_[ 1 ], 0 );

        #else

            ?_width_ := "-1";       // Default width value.

        #endif

        // Determine if the user supplied a fractional width
        // value with this parameter.

        #if( _fieldCnt_ > 2 )

            ?_decpts_ := @trim( _parmArray_[ 2 ], 0 );

        #else

            ?_decpts_ := "-1";  // Default fractional value.

        #endif

        // Quick check to see if the user supplied too many
        // width fields with this parameter.

        #if( _fieldCnt_ > 3 )

            #error
            (
                "<<" + _parameters_[ _curparm_ ] + ">>" +
                " has too many width fields"
            );

        #endif
```

After extracting the components of the current parameter, the stdout.put macro now goes about its business of determining the type of the current parameter so it can determine which HLA Standard Library function to call to actually print the value. This code sequence uses the HLA `@pType` and `@typename` compile-time functions to determine the symbol s type (see the *HLA Reference Manual* for more details). Note that this code also handles arrays of these objects by determining the base address of the array object (this is what the `#while` loop is doing in the following code sequence).

```
        // Determine the type of this parameter so we can
        // call the appropriate routine to print it.

        ?_pType_ := @pType( @text( _id_ ));
        ?_tempid_ := _id_;
```

```
#while( _pType_ = hla.ptArray )

    ?_tempid_ := @typename( @text( _tempid_ ));
    ?_pType_ := @pType( @text( _tempid_ ));

#endwhile
```

Once stdout.put has a handle on the parameter s type, it invokes the stdout._put_ macro (which we ll describe in a moment) to actually emit the code. Note that the `@pType` compile-time function returns an integer value that specifies a primitive HLA type (hence the name `pType`) and the corresponding type values have been given meaningful names in the *hla.hhf* header file (which this code is using). Another interesting aspect to this code is that if the argument is a class object, then it will automatically call a put method for that class, if such a method exists. This is how `stdout.put` extends its ability to print user-defined data types.

```
// Based on the type, call the appropriate library
// routine to print this value.

#if( _pType_ = hla.ptBoolean )
    stdout._put_( stdout.putbool, boolean )

#elseif( _pType_ = hla.ptUns8 )
    stdout._put_( stdout.putu8, uns8 )

#elseif( _pType_ = hla.ptUns16 )
    stdout._put_( stdout.putu16, uns16 )

#elseif( _pType_ = hla.ptUns32 )
    stdout._put_( stdout.putu32, uns32 )

#elseif( _pType_ = hla.ptUns64 )
    stdout._put_( stdout.putu64, uns64 )

#elseif( _pType_ = hla.ptUns128 )
    stdout._put_( stdout.putu128, uns128 )

#elseif( _pType_ = hla.ptByte )
    stdout._put_( stdout.putb, byte )

#elseif( _pType_ = hla.ptWord )
    stdout._put_( stdout.putw, word )

#elseif( _pType_ = hla.ptDWord )
    stdout._put_( stdout.putd, dword )

#elseif( _pType_ = hla.ptQWord )
    stdout._put_( stdout.putq, qword )

#elseif( _pType_ = hla.ptLWord )
    stdout._put_( stdout.putl, lword )

#elseif( _pType_ = hla.ptInt8 )
    stdout._put_( stdout.puti8, int8 )

#elseif( _pType_ = hla.ptInt16 )
    stdout._put_( stdout.puti16, int16 )
#elseif( _pType_ = hla.ptInt32 )
    stdout._put_( stdout.puti32, int32 )

#elseif( _pType_ = hla.ptInt64 )
    stdout._put_( stdout.puti64, int64 )

#elseif( _pType_ = hla.ptInt128 )
    stdout._put_( stdout.puti128, int128 )

#elseif( _pType_ = hla.ptChar )
    stdout._put_( stdout.putc, char )

#elseif( _pType_ = hla.ptCset )
    stdout._put_( stdout.putcset, cset )

#elseif( _pType_ = hla.ptReal32 )
    stdout._put_( stdout.putr32, real32 )

#elseif( _pType_ = hla.ptReal64 )
    stdout._put_( stdout.putr64, real64 )

#elseif( _pType_ = hla.ptReal80 )
    stdout._put_( stdout.putr80, real80 )

#elseif( _pType_ = hla.ptString )
    stdout._put_( stdout.puts, string )

#elseif( @isclass( @text( _parameters_[  _curparm_ ] )))

    #if
    (
        @defined
        (
            @text( _parameters_[  _curparm_ ] + ".toString" )
        )
    )

        push( eax );
        push( esi );
        push( edi );
        @text
        (
            _parameters_[  _curparm_ ] +
            ".toString()"
        );
        puts( eax );
        strfree( eax );
        pop( edi );
        pop( esi );
        pop( eax );

    #else

        #error
        (
            "stdout.put: Class does not provide a toString " +
                "method or procedure"
            );

        #endif

    #else

        #error
        (
            "stdout.put: Unknown data type (" +
            _parameters_[  _curparm_ ] +
            ":" +
            @typename( @text( _id_ )) +
            ")"
        );

    #endif
    ?_curparm_ := _curparm_ + 1;

#endwhile

// The following is stuck here just to require
// that the user end the stdout.put(--) invocation
// with a semicolon.

static
    ;
endstatic


#endmacro;
```

The `stdout.put` macro actually invokes a couple of macros. One of these is the `stdio._GetID_` macro (which we ll not cover here, see the source code in the *stdio.hhf* header file found in the HLA include subdirectory for details). The second macro that `stdout.put` invokes is the `stdout._put_` macro that expands into a call to a specific HLA Standard Library function depending on the type of the parameter that `stdout.put` is processing. This macro calculates the size information and fills in the parameter to the actual HLA Standard Library call.

```
#macro _put_( _routine_, _typename_ ):
                      _func_, sizeParms, _realsize_, _typ_;

    ?_func_:string := @string(_routine_);
    ?sizeParms:string := "";
    ?_typ_:string := @string(_typename_)

    // Real values allow two size parameters (width & decpts).

    #if( @substr( _typ_, 0, 4 ) = "real" )

        // Note: on entry, typename = real32, real64, or real80 and
        // routine = putr32, putr64, putr80, fputr32, fputr64, or
        // fputr80.

        ?_realsize_:string := @substr( _typ_, 4, 2 );
        #if( _width_ <> "-1" )


            // If decpts is <> -1, print in dec notation,
            // else print in sci notation.

            #if( _decpts_ <> "-1" )

                ?sizeParms:string := "," + _width_ + "," + _decpts_;

            #else

                ?_func_:string := "stdout.pute" + _realsize_;
                ?sizeParms:string := "," + _width_;

            #endif

        #else

            // If the user did not specify a format size,
            // then use the puteXX routines with default
            // sizes of: real32=15, real64=22, real80=28.

            ?_func_:string := "stdout.pute" + _realsize_;
            #if( _realsize_ = "32" )

                ?sizeParms:string := ",15";

            #elseif( _realsize_ = "64" )

                ?sizeParms:string := ",20";

            #else

                ?sizeParms:string := ",23";

            #endif

        #endif

    #else //It's not a real type.

        #if( _decpts_ <> "-1" )

            #error
            (
                "Fractional width specification is not supported here"
            )

        #elseif( _width_ <> "-1" )

            // Width specifications are only supported for
            // certain types.  Check for that here.

            #if
            (
                    _typ_ <> "uns8"
                &   _typ_ <> "uns16"
                &   _typ_ <> "uns32"
                &   _typ_ <> "uns64"
                &   _typ_ <> "uns128"
                &   _typ_ <> "int8"
                &   _typ_ <> "int16"
                &   _typ_ <> "int32"
                &   _typ_ <> "int64"
                &   _typ_ <> "int128"
                &   _typ_ <> "char"
                &   _typ_ <> "string"
                &   _typ_ <> "byte"
                &   _typ_ <> "word"
                &   _typ_ <> "dword"
                &   _typ_ <> "qword"
                &   _typ_ <> "lword"
            )

                #error
                (
                    "Type " +
                    _typ_ +
                    " does not support width format option"
                )

            #else

                ?_func_:string := _func_ + "Size";
                ?sizeParms:string := "," + _width_ + ", ' '";

            #endif

        #endif

    #endif

    // Here's the code that calls the appropriate function based on the parameter's
    // type and format information:

    #if
    (
            @isconst( @text( _arg_ ))
        &   _typ_ = "string"
        &   _arg_ = "#13 #10"
    )
        stdout.newln();

    #elseif( @isconst( @text( _arg_ )) )

        @text( _func_ )( @text( _arg_ ) @text( sizeParms ));

    #else

        @text( _func_ )
            ( (type _typename_ @text( _arg_ )) @text( sizeParms ));

    #endif
```

```
    #endmacro;
```

2.6: Even More Advanced HLA Programming...

This chapter could go on and on forever. HLA is a very sophisticated assembly language and provides tons of features that this chapter doesn t even touch upon. However, we do need to get on with the real purpose of this book, learning Win32 assembly language programming. We ve covered enough advanced HLA programming to deal with just about everything you will encounter the need for when writing Win32 assembly applications. Of course, if you want to learn more about HLA, here are two suggestions: (1) Read the HLA Reference manual and (2) write lots of HLA code and experiment.