8 HLA Data Types

8.1 Data Types in HLA

Unlike traditional x86 assemblers that tend to work only with bytes, words, double-words, quad-words, and long- (oct-) words, HLA provides a rich set of basic primitive types. This chapter discusses all the built-in and user-definable types that HLA supports.

8.2 Native (Primitive) Data Types in HLA

HLA provides the following basic primitive types:

boolean	One byte; zero represents false, one represents true (any non-zero value also represents true).
Enum	One, two, or four bytes (program selectable, default is one byte); user defined IDs with unique values.
Uns8	Unsigned values in the range 0..255.
Uns16	Unsigned integer values in the range 0..65535.
Uns32	Unsigned integer values in the range 0..4,204,967,295.
Uns64	Unsigned 64-bit integer.
Uns128	Unsigned 128-bit integer.
Byte	Generic eight-bit value.
Word	Generic 16-bit value.
DWord	Generic 32-bit value.
QWord	Generic 64-bit value.
TByte	Generic 80-bit value.
LWord	Generic 128-bit value.
Int8	Signed integer values in the range -128..+127.
Int16	Signed integer values in the range -32768..+32767.
Int32	Signed integer values in the range -2,147,483,648..+2,147,483,647.
Int64	Signed 64-bit integer values.
Int128	Signed 128-bit integer values.
Char	Character values.
WChar	Unicode character values.
Real32	32-bit floating-point values.
Real64	64-bit floating-point values.
Real80	80-bit floating-point values.
Real128	128-bit floating-point values (for SSE/2 instructions).
String	Dynamic length string constants. (Run-time implementation: four-byte pointer.)
ZString	Zero-terminated dynamic length strings (run-time implementation: four-byte pointer).
Unicode	Unicode strings.
CSet	A set of up to 128 different ASCII characters (16-byte bitmap).

Text	Similar to string, but text constants expand in-place (like #define in C/C++).
Thunk	A set of machine instructions to execute.

Often, it is convenient to discuss the types above in various groups. The HLA language reference manual will often use the following terms:

Ordinal:	boolean, enum, uns8, uns16, uns32, byte, word, dword, int8, int16, int32, char.
Unsigned:	uns8, uns16, uns32, byte, word, dword.
Signed:	int8, int16, int32, byte, word, dword.
Number:	uns8, uns16, uns32, int8, int16, int32, byte, word, dword
Numeric:	uns8, uns16, uns32, int8, int16, int32, byte, word, dword, real32, real64, real80

8.2.1 Enumerated Data Types

HLA provides the ability to associate a list of identifiers with a user-defined type. Such types are known as enumerated data types (because HLA enumerates, or numbers, each of the identifiers in the list to give them a unique value). The syntax for an enumerated type declaration (in an HLA `type` section, see the description a little later) takes the following form:

```
typename : enum{ list_of_identifiers };
```

Here is a typical example:

```
type
   color_t :enum{ red, green, blue, magenta, yellow, cyan, black, white };
```

Internally, HLA treats enumerated types as though they were unsigned integer values (though **`enum`** types are not directly compatible with the unsigned types). HLA associates the value zero with the first identifier in the **`enum`** list and then attaches sequentially increasing values to the following identifiers in the list. For example, HLA will associate the following values with the **`color_t`** symbolic constants:

```
red          0
green     1
blue      2
magenta3
yellow    4
cyan      5
black     6
white     7
```

Because each enumerated constant in a given **`enum`** list is unique, you may compare these values, use them in computations, etc. Also note that, because of the way HLA assigns internal values to these constant names, you may compare objects in an enumerated list for less than and greater than in addition to equal or not equal.

Note that HLA uses zero as the internal representation for the first symbol of every **`enum`** list. HLA only guarantees that the values it associates with **`enum`** types are unique for a single type; it does not make this guarantee across different enumerated types (in fact, you're guaranteed that different **enum** types *do not* use unique values for their symbol sets). In the following example, HLA uses the value zero for both the internal representation of *`const0`* and *`c0`*. Likewise, HLA uses the value one for both *`const1`* and *`c1`*. And so on...

```
type
   enumType1 :enum{ const0, const1, const2 };
   enumType2 :enum( c0, c1, c2 };
```

Note that the enumerated constants you specify are not "private" to that particular type. That is, the constant names you supply in an enumerated data type list must be unique within the current scope (see the definition of identifier scope elsewhere in the HLA documentation). Therefore, the following is *not* legal:

```
type
    enumType1 :enum{ et1, et2, et3, et4 };
    enumType2 :enum{ et2, et2a, et2b, et2c }; //et2 is a duplicate symbol!
```

The problem here is that both type lists attempt to define the same symbol: *et2*. HLA reports an error when you attempt this.

One way to view the enumerated constant list is to think of it as a list of constants in an HLA **const** section (see the description of declaration sections a little later in this document), e.g.,

```
const
    red       : color_t := 0;
    green     : color_t := 1;
    blue      : color_t := 2;
    magenta: color_t := 3;
    yellow    : color_t := 4;
    cyan      : color_t := 5;
    black     : color_t := 6;
    white     : color_t := 7;
```

By default, HLA uses 8-bit values to represent enumerated data types. This means that you can represent up to 256 different symbols using an enumerated data type. This should prove sufficient for most applications. HLA provides a special "compile-time variable" that lets you change the size of an enumerated type from one to two or four bytes. All you have to do is assign the value two or four to this variable and HLA will automatically resize the storage for enumerated types to handle longer lists of objects. Example:

```
?@enumSize := 4;// Use dword size for enum types

type
    enumDword:enum{ d0, d1, d2, d3};

var
    ed  :enumDword;// Reserves four bytes of storage
```

8.2.2 HLA Type Compatibility

HLA is unusual among assembly language insofar as it does some serious type checking on its operands. While the type checking isn't quite as "strong" as some high-level languages, HLA clearly does a lot more type checking than other assemblers, even those that purport to do type checking on operands (e.g., MASM). The use of strong type checking can help you locate logical errors in your code that would otherwise go unnoticed (except via a laborious and time consuming testing/debug session).

The downside to strong type checking is that experienced assembly programmers may become somewhat annoyed with HLA's reports that they are doing something wrong when, in fact, the programmer knows exactly what they are doing. There are two solutions to this problem: use type coercion (described a little bit later) or use the "untyped" types that reduce type checking to simply ensuring that the sizes of the operands match. However, before discussing how to override HLA's type checking system, it's probably a good idea to first describe how HLA uses data types.

Fundamentally, HLA divides the data types into classes based on the size of their underlying representation. Unless you explicitly override a type with a type coercion operation, attempting to mix object sizes in a memory or register operand will produce an error (in constant expressions, HLA is a bit more forgiving; it will automatically promote between certain types and adjust the type of the result accordingly). With most of HLA's data types, it's obvious what the size of the underlying representation is, because most HLA type names incorporate the size (in bits) in the type's name. For example, the **uns16** data type is a 16-bit (two-byte) type. Nevertheless, this rule isn't true for all data types, so it's a good idea to begin this discussion by looking at the underlying sizes of each of the HLA types.

8 bits: boolean, byte, char, enum, int8, uns8

16 bits: int16, uns16, wchar, word

32 bits:	dword, int32, pointer types, real32, string, zstring, unicode, uns32
64 bits:	int64, qword, real64, uns64
80 bits:	real80, tbyte
128 bits:	cset, int128, lword, uns128, real128

The **byte, word, dword, qword, tbyte,** and **lword** types are somewhat special. These are known as *untyped data types*. They are directly compatible with any scalar, ordinal, data type that is the same size as the type in question. For example, a **byte** object is directly compatible with any object of type **boolean, byte, char, enum** (assuming **@enumSize** is 1), **int8**, or **uns8**. No special coercion is necessary when assigning a **byte** value to an object that has one of these other types; likewise, no special coercion operation is necessary when assigning a value of one of these other types to a **byte** object.

Note that **cset, real32, real64, real80,** and **real128** objects are not ordinal types. Therefore, you cannot directly mix these types with **lword, dword, qword, tbyte,** or **lword** objects without an explicit type coercion operation. Also, keep in mind that composite data types (see the next section) are not directly compatible with **bytes, words, dwords, qwords, tbytes,** and **lwords**, even if the composite data type has the same number of bytes (the only exception is the pointer data type, which is compatible with the **dword** type).

8.3 Composite Data Types

In addition to the primitive types above, HLA supports pointers, arrays, records (structures), unions, and classes of the primitive types (except for text objects).

8.4 Array Data Types

HLA allows you to create an array data type by specifying the number of array elements after a type name. Consider the following HLA type declaration that defines *intArray* to be an array of int32 objects:

```
type intArray : int32[ 16 ];
```

The "[16]" component tells HLA that this type has 16 four-byte integers. HLA arrays use a zero-based index, so the first element is always element zero. The index of the last element, in this example, is 15 (total of 16 elements with indices 0..15).

HLA also supports multidimensional arrays. You can specify multidimensional arrays by providing a list of indices inside the square brackets, e.g.,

```
type intArray4x4 : int32[ 4, 4 ];
type intArray2x2x4 : int32[ 2,2,4 ];
```

The mechanism for accessing array elements differs depending upon whether you are accessing compile-time array constants or run-time array variables. A complete discussion of this will appear in later sections.

8.5 Union Data Types

HLA implements the discriminate 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;
```

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 declaration section (see the discussion of the **var** section in the chapter on *HLA Program Structure* for more details).

Given a union object, say *i* of type *allInts*, you access the fields of the union using the familiar dot-notation. The following 80x86 **mov** instructions demonstrate how to access each of the fields of the *i* variable:

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

Unions also support a special field type known as an anonymous record (see the next section for a description of records). The syntax for an anonymous record in a union is the following:

```
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.

8.6 Record Data Type[1]s

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.

1. For C/C++ programmers: an HLA record is similar to a C struct. In language design terminology, a record is often referred to as a "cartesian product."

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**. As for unions, you use the dot-notation to access fields of a **record** object.

In addition to the **var**-like declarations, you may also declare anonymous unions within a record. An anonymous union is a **union** declaration without a fieldname 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.

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* inherits all the fields from the *Pt2D* 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:

```
    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:

```
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 the same name. That is, perhaps the programmer intends to actually create the following record:

```
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. While the previous declarations for *DerivedRecord* produce errors, the following is acceptable to HLA:

```
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 can only use values that are integral powers of two in the **align** directive and the value must be 16 or less.

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 that is a power of two in the range 1..16 inside these parentheses. If you specify the value 1 (which is the default), then all fields are packed (aligned on a byte boundary). For values greater than 1, 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.

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 8-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 8-byte boundary, then *d* and *e* will not actually be on 8-byte boundaries; they will, however be on 8-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 (because 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 within the outermost record on a boundary that is compatible with the size of the largest element of the nested array, union, or record.

HLA 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.

When declaring record variables in a **var, static, readonly,** or **storage** declaration section, HLA associates the offset zero with the first field of a record. Each additional field in the record is assigned an offset corresponding to the sum of the sizes of all the prior fields. So in the following example, *x* would have the offset zero, *y* would have the offset four, and *z* would have the offset eight.

```
Pt3D:
    record

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

    endrecord;
```

If you would like to specify a different starting offset, you can use the following syntax for a record declaration:

```
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 advanced assembly language programmers should use this option.

8.7 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 (near32) pointers.

HLA v1.x allowed 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;
```

However, this pointer syntax has been deprecated as of HLA v2.0 and this syntax will disappear sometime in HLA v2.x. The modern way to declare pointers that are compatible with a particular procedure is to use the "new style" procedure declarations in the HLA proc section. This is done as follows:

```
type
    p : procedure( parameter_list );
        .
        .
        .
proc
    someProc:p {optional procedure options};

        .
        .
        .
```

See the HLA reference manual chapter on *HLA Procedures* for more details about the **proc** section.

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).

8.8 Thunks

A "thunk" is an 8-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 uses the activation record of the surrounding 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. To declare a **thunk** variable is easy, just use a declaration like the following in a **var**, **static, readonly,** or **storage** section:

```
thunkVar: thunk;
```

This declaration reserves eight bytes of storage. The first double-word holds the address of the code to execute, the second double-word holds a pointer to the activation record to load into EBP when the thunk executes.

Of course, like almost any pointer variable, declaring a **thunk** variable is the easy part; the hard part is making sure the **thunk** variable is initialized before attempting to call the thunk. While you could manually load the address of some code and the frame pointer value into a **thunk** variable, HLA provides a better syntax for initializing thunks with small code fragments: the **thunk** statement. The **thunk** statement uses the following syntax:

```
thunk thunkVar := #{ sequence_of_statements }#;
```

Consider the following example:

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

procedure proc1;
var
    i:      int32;
    p1Thunk: thunk;

    procedure proc2( t:thunk );
    var
        i:int32;
    begin proc2;

        mov( 25, i );
        t();
        stdout.put( "Inside proc2, i=", i, nl );

    end proc2;

begin proc1;

    thunk p1Thunk := #{ mov( 0, i ); }#;

    mov( 1, i );
    proc2( p1Thunk );
    stdout.put( "i=", i, nl );

end proc1;

begin ThunkDemo;

    proc1();

end ThunkDemo;
```

In this example, proc1 has two local variables, *i* and *p1Thunk*. The **thunk** statement initializes the *p1Thunk* variable with the address of some code that moves a zero into the *i* variable. The **thunk** statement also initializes *p1Thunk* with a pointer to the current activation record (that is, a pointer to *proc1's* activation record). Then *proc1* calls *proc2* passing *p1Thunk* as a parameter.

The *proc2* routine has its own local variable named *i*. Of course, this is a different variable from the *i* in *proc1*. *Proc2* begins by setting its variable *i* to the value 25. Then *proc2* invokes the thunk (passed to it as a parameter). This thunk sets the variable *i* to zero. However, because the thunk uses the current activation record when the **thunk** statement was executed, this statement sets *proc1's i* variable to zero rather than *proc2's i* variable. This program produces the following output:

```
Inside proc2, i=25
i=0
```

Although you probably won't use thunks that often, they are quite nice for deferred execution. This is especially useful in AI (Artificial Intelligence) programs.

8.9 Class Types

Classes and object-oriented programming are the subject of a different HLA Reference Manual Document. See the chapter on *HLA Classes* for more details.

8.10 Regular Expression Types

The HLA compile-time language supports a special data type known as a "compiled regular expression". Please see the section on regular expression macros in the chapter on the *HLA Compile-Time Language* for more details on this data type.