17 HLA Related Macros and Constants (hla.hhf)

The HLA module contains numeric constants produced by some of the HLA symbol-table compile-time functions. It also contains various macros to extend the HLA compile-time language and provide support for other HLA stdlib modules.

17.1 The HLA Module

To use the HLA macros and constants in your application, you will need to include one of the following statements at the beginning of your HLA application:

```
#include( "hla.hhf" )
or
#include( "stdlib.hhf" )
```

17.2 Classification Macros

The hla.hhf module contains some macros that test the type of an identifier at compile time. Here is a typical invocation of these macros:

```
#if( hla.IsUns( uVar ))
    // do something if Uns object
#else
    // Do something if not unsigned object
#endif
```

#macro hla.IsUns(identifier);

This macro returns a compile-time expression that evaluates true if the specified identifier is an uns8, uns16, uns32, uns64, or uns128 object.

#macro hla.IsInt(identifier);

This macro returns a compile-time expression that evaluates true if the specified identifier is an int8, int16, int32, int64, or int128 object.

#macro hla.IsHex(identifier);

This macro returns a compile-time expression that evaluates true if the specified identifier is a byte, word, dword, qword, tbyte, or lword object.

#macro hla.IsNumber(identifier);

This macro returns a compile-time expression that evaluates true if the specified identifier is an uns8, uns16, uns32, uns64, un128, int8, int16, int32, int64, int128, byte, word, dword, qword, tbyte, or lword object.

#macro hla.IsReal(identifier);

This macro returns a compile-time expression that evaluates true if the specified identifier is a real32, real64, or real80 object.

#macro hla.IsNumeric(identifier);

This macro returns a compile-time expression that evaluates true if the specified identifier is an uns8, uns16, uns32, uns64, uns128, int8, int16, int32, int64, int128, byte, word, dword, qword, tbyte, lword, real32, real64, or real80 object.

#macro hla.IsOrdinal(identifier);

This macro returns a compile-time expression that evaluates true if the specified identifier is an uns8, uns16, uns32, uns64, uns128, int8, int16, int32, int64, int128, boolean, char, byte, word, dword, qword, tbyte, lword, or enumerated data type object.

17.3 String to Integer Macros

The HLA module provides two macros, hla.asWord and hla.asDword, that let you treat a one to four-character string as a 16-bit or 32-bit number. Specifically, these macros cram the 1-4 bytes of the strings into a two-byte word constant or a 4-byte dword constant.

#macro hla.asWord("1 or 2 character string");

This macro places the first character of the string in the L.O. byte of the 16-bit result, and the second character of the string in the H.O. byte of the result. If there is only one character in the string, the H.O. byte of the result will be zero.

#macro hla.asDword("1 to 4 character string");

This macro places copies the first through fourth characters of the string to the L.O. to H.O. bytes of the dword results. If there are fewer than 4 characters, the H.O. byte(s) of the result are filled with zeros.

17.4 Label Generation Macro

The hla.genlabel macro generates a sequence of strings that are unique, legal, HLA identifiers (within the current compilation, do not use these as public symbols). Typically, you would take the string that this macro returns and convert that string to an actual symbol using the @TEXT function.

Here's the definition of *hla.getLabel* in the HLA header file:

```
val
    _hla_labelCnt_ := 0;


#macro genLabel;

    "_genLabel_" + string( hla._hla_labelCnt_ ) + "_"
    ?hla._hla_labelCnt_ := hla._hla_labelCnt_ + 1;

#endmacro;
```

17.5 Procedure Overloading Macro

The *hla.overload* macro allows you to provide the equivalent of "overloaded functions" in your HLA programs. Note that this macro is somewhat obsolete as the HLA language now provides built-in overloading (see the HLA Reference Manual for details). This macro is provided for compatibility with old code that is still using it.

Overloaded functions are called by "signature" rather than by name. A signature is the combination of the name of a function, the number of parameters, and the types of each of the parameters. As long as two separate functions have unique signatures, they can be called using the same name. Of course, HLA requires each

procedure/iterator/method/macro to have a unique name within the current scope, but the overload macro lets you define the name to use when calling these different functions.

As an example, consider the following three function prototypes:

```
procedure min8( val1:uns8; val2:uns8 ); @external;
procedure min16( val1:uns16; val2:uns16 ); @external;
procedure min32( val1:uns32; val2:uns32 ); @external;
```

We would like to call these three functions using the single name "min" and letting the different signatures (the differing parameter types in this case) select the actual function to call. The hla.overload macro will write a "min" macro for us that will determine the parameter types and call one of the functions above based on the signature of the call. An overloaded definition looks like the following:

```
#macro min( _parms_[] );// Define "min", our overloaded procedure
   hla.overload( _parms_ );
     hla.signature( min8( uns8, uns8 ));
     hla.signature( min16( uns16, uns16 ));
     hla.signature( min32( uns32, uns32 ));
   hla.endoverload
#endmacro
```

The first thing to note is that you must create a macro that will be used as the "overloaded procedure". In this example, that macro is the "min" macro. This macro must always have a single array (variable) parameter. The name isn't important, though "_parms_" is the conventional name to use here.

The body of the overloaded macro is actually going to be written by the hla.overload macro. The hla.overload macro is an HLA context-free macro that begins with "hla.overload", ends with "hla.endoverload", and contains a single "hla.signature" keyword macro invocation for each signature we want our overloaded procedure to support. In this example, we want our min function to call one of three different functions based on the types of the parameters passed to min, so there will be three signature invocations.

A signature keyword macro invocation takes the following form:

```
hla.signature( actualProcedureName( list_of_parameter_types ));
```

where *actualProcedureName* is the name of the function we want to call if the signature matches and *list_of_parameter_types* is a comma-separated list of HLA data type names, that correspond to the data types for the signature.

The hla.overload macro will write the body of the min macro so that it will parse the parameters passed in (via _parms_) and determine which of the three functions to call. For example:

```
min( u8, u8a ); // calls min8( u8, u8a );
min( u16, u16a );// calls min16( u16, u16a );
min( u32, u32a ); // calls min32( u32, u32a );
```

Here are some additional examples of overloaded macro definitions from the HLA stdlib:

```
#macro catsub( parms[] );

      hla.overload( parms )

         hla.signature( str.catsub4(string, dword, dword, string) )
         hla.signature( str.catsub5(string, dword, dword, string, string))

      hla.endoverload

   #endmacro

  procedure catsub4( src:string; start:dword; len:dword; dest:string );
     external( "STR_CATSUB4" );

  procedure catsub5
  (
     src2:string;
```

```
        start:dword;
        len    :dword;
        src1:string;
        dest:string
    );
        @returns( "(type string eax)" );
        external( "STR_CATSUB5" );


    #macro first( parms[] );

        hla.overload( parms )

            hla.signature( str.first2(string, dword) )
            hla.signature( str.first3(string, dword, string) )

        hla.endoverload

    #endmacro


    procedure first2
    (
        s      :string;
        len    :dword
    ); external( "STR_FIRST2" );

    procedure first3
    (
        s      :string;
        len    :dword;
        dest:string
    ); external( "STR_FIRST3" );
```

When you use an overloaded function (that is, you invoke the macro whose body was filled in by the *hla.overload* macro), the code attempts to match the actual parameters against the signatures you've provided. If no possible signature can match the actual parameter list, then the system will report an error. If two or more signatures can be matched by the actual parameter list, the system will also report an error. Unfortunately, if the actual parameter list provides an ambiguous footprint, then the system will report an error. Consider the following overloaded function and a couple invocations of that function:

```
static
  u8     :uns8;
  u16 :uns16;
  s      :string;

#macro abc( _parms_[] );
  hla.overload( _parms_ )

      hla.signature( abc1( uns8, string ))
      hla.signature( abc2( uns16, string ))

  hla.endoverload;
#endmacro
  .
  .
  .
abc( u8, s ); // Okay, calls abc1
abc( u8, "abc1" );// Okay, calls abc1
abc( u16, s );// Okay, calls abc2
```

```
abc( 0, s );// Illegal – matches multiple signatures
```

The last example doesn't work because the literal constant zero matches both the uns8 and uns16 types, so this is an ambiguous signature match. Keep this limitation in mind when creating your signatures.

17.6 Generic PUT Macro

The *hla.put* macro provides a mechanism for creating generic "put" functions like the *stdout.put, stderr.put, fileio.put,* and *str.put* macros found in the HLA standard library. Indeed, these macros were built using the *hla.put* macro. By using the hla.put macro, you can easily create your own "put" macro that calls a variety of different functions based on the types of the parameters passed to your "put" macro.

First of all, it's important to realize that a user-defined "put" macro must appear inside a namespace. The *hla.put* macro expects to find several objects defined in a namespace whose name you provide. The namespace will contain all the procedures that the user-defined macro will ultimately call as well as a few additional compile-time data structures that the *hla.put* macro will reference.

Within the namespace you're defining a "put" macro for, there must be six constant array objects with *hla.sizePTypes* elements each. These arrays must have the following names and types:

```
validPutType:boolean [ @global:hla.sizePTypes ];
validPutSizeType:boolean [ @global:hla.sizePTypes ];
validPutSize2Type:boolean [ @global:hla.sizePTypes ];
putFunc     :string [ @global:hla.sizePTypes ];
putSizeFunc:string [ @global:hla.sizePTypes ];
putSize2Func:string [ @global:hla.sizePTypes ];
```

Assuming you've created a namespace called "myns", then *myns.validPutType* will tell the *hla.put* macro which built-in data types it can process when you specify an unadorned argument to *myns.put*. For example, if i8 is an int8 object, then myns.put(i8); will call some function if the *myns.validPutType* entry indexed by *hla.ptInt8* contains true. Here is the validPutType table for the stdout namespace; most validPutType tables will be a copy of this one (assuming you want your new "put" macro to handle all the same data types as *stdout.put*):

```
const
  validPutType :boolean[ @global:hla.sizePTypes ] :=
     [
        @global:false,// Undefined
        @global:true,// tBoolean //1
        @global:false,// enum//2
        @global:true,// tUns8,//3
        @global:true,// tUns16,//4
        @global:true,// tUns32,//5
        @global:true,// tUns64,//6
        @global:true,// tUns128,//7
        @global:true,// tByte,//8
        @global:true,// tWord,//9
        @global:true,// tDWord,//10
        @global:true,// tQWord,//11
        @global:true,// tTByte,//12
        @global:true,// tLWord,//13
        @global:true,// tInt8,//14
        @global:true,// tInt16,//15
        @global:true,// tInt32,//16
        @global:true,// tInt64,//17
        @global:true,// tInt128,//18
        @global:true,// tChar, //19
        @global:false,// tWChar,//20
        @global:true,// tReal32,//21
        @global:true,// tReal64, //22
        @global:true,// tReal80,//23
        @global:false,// tReal128,//24
        @global:true,// tString,//25
        @global:false,// tZString,//26
```

```
            @global:false,// tWString,//27
            @global:true,// tCset,//28
            @global:false,// tArray,//29
            @global:false,// tRecord,//30
            @global:false,// tUnion,//31
            @global:false,// tRegex//32
            @global:true,// tClass,//33
            @global:false,// tProcptr,//34
            @global:false,// tThunk,//35
            @global:true,// tPointer//36
            @global:false,// tLabel//37
            @global:false,// tProc//38
            @global:false,// tMethod//39
            @global:false,// tClassProc//40
            @global:false,// tClassIter//41
            @global:false,// tIterator//42
            @global:false,// tProgram//43
            @global:false,// tMacro//44
            @global:false,// tText//45
            @global:false,// tRegExMac//46
            @global:false,// tNamespace//47
            @global:false,// tSegment//48
            @global:false,// tAnonRec//49
            @global:false,// tAnonUnion//50
            @global:false,// tVariant//51
            @global:false// tError//52
        ];
```

Notice that each element of the array is indexed by the pType value for the data type.

The validPutSizeType array is very similar to the validPutType array (indeed, it is structurally identical to the validPutType array). The difference between the two is that the hla.put macro uses the validPutSizeType array to determine if it can call a "*Size" function when the "put" macro encounters an operand of the form "xx:ss", where "ss" is a print width specfication. For example, if you write "myns.put(i8:4);" and i8 is an int8 variable, then the hla.put macro will check the hla.ptInt8 element of the validPutSizeType array to determine whether it supports a field width for int8 output. Here is the stdout version of this array (again, most uses of the *hla.put* macro will copy this, assuming they provide all the same output functionality as *stdout.put*):

```
validPutSizeType :boolean[ @global:hla.sizePTypes ] :=
      [
            @global:false,// Undefined
            @global:true,// tBoolean //1
            @global:false,// enum//2
            @global:true,// tUns8,//3
            @global:true,// tUns16,//4
            @global:true,// tUns32,//5
            @global:true,// tUns64,//6
            @global:true,// tUns128,//7
            @global:true,// tByte,//8
            @global:true,// tWord,//9
            @global:true,// tDWord,//10
            @global:true,// tQWord,//11
            @global:true,// tTByte,//12
            @global:true,// tLWord,//13
            @global:true,// tInt8,//14
            @global:true,// tInt16,//15
            @global:true,// tInt32,//16
            @global:true,// tInt64,//17
            @global:true,// tInt128,//18
            @global:true,// tChar, //19
            @global:false,// tWChar,//20
            @global:true,// tReal32,//21
```

```
        @global:true,// tReal64, //22
        @global:true,// tReal80,//23
        @global:false,// tReal128,//24
        @global:true,// tString,//25
        @global:false,// tZString,//26
        @global:false,// tWString,//27
        @global:false,// tCset,//28
        @global:false,// tArray,//29
        @global:false,// tRecord,//30
        @global:false,// tUnion,//31
        @global:false,// tRegEx//32
        @global:false,// tClass,//33
        @global:false,// tProcptr,//34
        @global:false,// tThunk,//35
        @global:true,// tPointer//36
        @global:false,// tLabel//37
        @global:false,// tProc//38
        @global:false,// tMethod//39
        @global:false,// tClassProc//40
        @global:false,// tClassIter//41
        @global:false,// tIterator//42
        @global:false,// tProgram//43
        @global:false,// tMacro//44
        @global:false,// tText//45
        @global:false,// tRegExMac//46
        @global:false,// tNamespace//47
        @global:false,// tSegment//48
        @global:false,// tAnonRec//49
        @global:false,// tAnonUnion//50
        @global:false,// tVariant//51
        @global:false// tError//52
    ];
```

The validPutSize2Type array is very similar to the validPutSizeType array. The difference between the two is that the *hla.put* macro uses the validPutSize2Type array to determine if it can call a "*Size" function when the "put" macro encounters an operand of the form "xx:ww:dd", where "ww" is a print width specfication and "dd" is a "number of decimal positions" value. This functionality is typically used for outputting real values in decimal form. For example, if you write "myns.put(r80:14:2);" and r80 is an real80 variable, then the *hla.put* macro will check the hla.ptReal80 element of the validPutSize2Type array to determine whether it supports a field width and decimal count for real80 output. Here is the stdout version of this array (again, most uses of the *hla.put* macro will copy this, assuming they provide all the same output functionality as *stdout.put*):

```
validPutSize2Type :boolean[ @global:hla.sizePTypes ] :=
    [
        @global:false,// Undefined
        @global:false,// tBoolean //1
        @global:false,// enum//2
        @global:false,// tUns8,//3
        @global:false,// tUns16,//4
        @global:false,// tUns32,//5
        @global:false,// tUns64,//6
        @global:false,// tUns128,//7
        @global:false,// tByte,//8
        @global:false,// tWord,//9
        @global:false,// tDWord,//10
        @global:false,// tQWord,//11
        @global:false,// tTByte,//12
        @global:false,// tLWord,//13
        @global:false,// tInt8,//14
        @global:false,// tInt16,//15
        @global:false,// tInt32,//16
```

```
            @global:false,// tInt64,//17
            @global:false,// tInt128,//18
            @global:false,// tChar, //19
            @global:false,// tWChar,//20
            @global:true,// tReal32,//21
            @global:true,// tReal64, //22
            @global:true,// tReal80,//23
            @global:false,// tReal128,//24
            @global:false,// tString,//25
            @global:false,// tZString,//26
            @global:false,// tWString,//27
            @global:false,// tCset,//28
            @global:false,// tArray,//29
            @global:false,// tRecord,//30
            @global:false,// tUnion,//31
            @global:false,// tRegEx//32
            @global:false,// tClass,//33
            @global:false,// tProcptr,//34
            @global:false,// tThunk,//35
            @global:false,// tPointer//36
            @global:false,// tLabel//37
            @global:false,// tProc//38
            @global:false,// tMethod//39
            @global:false,// tClassProc//40
            @global:false,// tClassIter//41
            @global:false,// tIterator//42
            @global:false,// tProgram//43
            @global:false,// tMacro//44
            @global:false,// tText//45
            @global:false,// tRegExMac//46
            @global:false,// tNamespace//47
            @global:false,// tSegment//48
            @global:false,// tAnonRec//49
            @global:false,// tAnonUnion//50
            @global:false,// tVariant//51
            @global:false// tError//52
        ];
```

The *putFunc, putSizeFunc,* and *putSize2Func* arrays in your namespace must contain the names of the functions to call if the corresponding entries in *validPutType, validPutSizeType*, and *validPutSize2Type* contain true (respectively). These strings must be the name of the procedure to call without the namespace prefix. That is, if you want to tell hla.put to call "myns.puti8" to print an 8-bit integer, then the entry at index *hla.ptInt8* in *myns.putFunc* should contain the string "puti8". Note that if the corresponding entry in the in *validPutType, validPutSizeType*, or *validPutSize2Type* tables contain false, then it doesn't matter what string appears in the array as *hla.put* will never use it; by convention, the empty string is always put in unused entries. Here is the *stdout.putFunc* table:

```
putFunc :string[ @global:hla.sizePTypes ] :=
      [
          "",           // Undefined
          "putbool",// tBoolean //1
          "",           // enum   //2
          "putu8",  // tUns8,//3
          "putu16",// tUns16,//4
          "putu32",// tUns32,//5
          "putu64",// tUns64,//6
          "putu128",// tUns128,//7
          "putb",   // tByte,//8
          "putw",   // tWord,//9
          "putd",   // tDWord,//10
```

```
            "putq",   // tQWord,//11
            "puttb",  // tLWord,//12
            "putl",   // tLWord,//13
            "puti8",  // tInt8,//14
            "puti16",// tInt16,//15
            "puti32",// tInt32,//16
            "puti64",// tInt64,//17
            "puti128",// tInt128,//18
            "putc",   // tChar, //19
            "",           // tWChar,//20
            "_pute32",// tReal32,//21
            "_pute64",// tReal64, //22
            "_pute80",// tReal80,//23
            "",           // tReal128,//24
            "puts",   // tString,//25
            "putz",   // tZString,//26
            "",           // tWString,//27
            "putcset",// tCset,//28
            "",           // tArray,//29
            "",           // tRecord,//30
            "",           // tUnion,//31
            "",           // tRegEx//32
            "",           // tClass,//33
            "",           // tProcptr,//34
            "",           // tThunk,//35
            "putd",   // tPointer//36
            "",           // tLabel//37
            "",           // tProc  //38
            "",           // tMethod//39
            "",           // tClassProc//40
            "",           // tClassIter//41
            "",           // tIterator//42
            "",           // tProgram//43
            "",           // tMacro//44
            "",           // tText  //45
            "",           // tRegExMac//46
            "",           // tNamespace//47
            "",           // tSegment//48
            "",           // tAnonRec//49
            "",           // tAnonUnion//50
            "",           // tVariant//51
            ""               // tError//52
        ];
```

ut.putFunc table:

Please see the stdout.hhf header file for examples of the other two tables.

Once you have set up the six constant arrays in your namespace, defining your own put macro using *hla.put* is almost trivial. The *hla.put* macro processes a single "put" operand. The syntax for this macro is the following:

```
hla.put( <namespaceID>, <first param as string>, <single argument> );
```

where <namespace> is your namespace identifier, <first param as string> is the first parameter to be passed to all your functions (this can be the empty string if you don't have a first parameter [e.g., *stdout* or *stderr*], or it can be whatever your output functions require; for example, the *fileio.put* macro specifies a file handle here, the *str.put* macro specifies a string variable name here).

Because *hla.put* only handles a single output object, you must provide a simple put macro within your namespace that calls *hla.put* for each of the actual arguments passed to your *put* macro. Here's the *stdout* version of that macro:

```
val
```

```
    stdoutParm:string;

#macro put( _parameters_[] );

    #for( @global:stdout.stdoutParm in _parameters_ )

        @global:hla.put( stdout, "", @eval(@global:stdout.stdoutParm) )

    #endfor

#endmacro
```

Here's the fileio version of this macro:

```
val
    _v_       :string;
    _curparm_:string;

#macro put( _ileVar_, _parameters_[] );

    ?@global:fileio._v_ := @string:_ileVar_;
    #for( @global:fileio._curparm_ in _parameters_ )

        @global:hla.put
        (
            fileio,
            @global:fileio._v_,
            @eval(@global:fileio._curparm_)
        )

    #endfor

#endmacro
```

17.7 @class Constants

The hla.hhf module contains some macros that test the type of an identifier at compile time. Here is a typical invocation of these macros:

The HLA compile-time @class function returns the following values to denote the classification of an identifier. If a symbol appears more than once in a program, the @class function returns the classification value for the symbol currently in scope.

Ταβλε 1 **@Class Return Values**

Name	Value	Description
hla.cIllegal	0	Symbol doesn't have a legal HLA classification.
hla.cConstant	1	Symbol was defined in the CONST section.
hla.cValue	2	Symbol was defined in the VAL section.
hla.cType	3	Symbol was defined in the TYPE section.
hla.cVar	4	Symbol was defined in the VAR section

hla.cParm	5	Symbol is a parameter.
hla.cStatic	6	Symbol was defined in a STATIC, READONLY, or STORAGE section.
hla.cLabel	7	Symbol is a statement label.
hla.cProc	8	Identifier is the name of a (non-class) procedure.
hla.cIterator	9	Identifier is the name of a (non-class) iterator.
hla.cClassProc	10	Identifier is the name of a class procedure.
hla.cClassIter	11	Identifier is the name of a class iterator.
hla.cMethod	12	Identifier is the name of a class method.
hla.cMacro	13	Symbol is a macro.
hla.cKeyword	14	Symbol is an HLA reserved word.
hla.cTerminator	15	Symbol is an HLA TERMINATOR macro.
hla.cRegex	16	Symbol is a regular expression macro
hla.cProgram	17	PROGRAM or UNIT identifier.
hla.cNamespace	18	Identifier is a name space ID.
hla.cSegment	19	Identifier is a segment name.
hla.cRegister	20	Identifier is an 80x86 register name.
hla.cNone	21	Reserved.

17.8 HLA pType Constants

The HLA @ptype compile-time function returns the values in the following table for the symbol you pass as a parameter to the function. You should always use these symbol names rather than hard-coding the constants in your programs. These values have changed in the past and they will likely change in the future with improvements to the HLA language.

Ταβλε 2**: @pType Return Values**

Symbol	Value	Description
hla.ptIllegal	0	Symbol is undefined or is not an object to which a type can be applied.
hla.ptBoolean	1	Symbol is of type boolean.
hla.ptEnum	2	Symbol is an enumerated type.
hla.ptUns8	3	Symbol is an UNS8 object.
hla.ptUns16	4	Symbol is an UNS16 object.

hla.ptUns32	5	Symbol is an UNS32 object.
hla.ptUns64	6	Symbol is an UNS64 object.
Hla.ptUns128	7	Symbol is an UNS128 object.
hla.ptByte	8	Symbol is a BYTE object.
hla.ptWord	9	Symbol is a WORD object.
hla.ptDWord	10	Symbol is a DWORD object.
hla.ptQWord	11	Symbol is a QWORD object.
hla.ptTByte	12	Symbol is a TBYTE object.
hla.ptLWord	13	Symbol is a LWORD object.
hla.ptInt8	14	Symbol is an INT8 object.
hla.ptInt16	15	Symbol is an INT16 object.
hla.ptInt32	16	Symbol is an INT32 object.
hla.ptInt64	17	Symbol is an INT64 object.
hla.ptInt128	18	Symbol is an INT128 object.
hla.ptChar	19	Symbol is of type CHAR.
hla.ptWChar	20	Symbol is of type WCHAR.
hla.ptReal32	21	Symbol is a REAL32 object.
hla.ptReal64	22	Symbol is a REAL64 object.
hla.ptReal80	23	Symbol is a REAL80 object.
hla.ptReal128	24	Symbol is a REAL128 object.
hla.ptString	25	Symbol has the STRING type.
hla.ptZString	26	Symbol has the ZSTRING type.
hla.ptWString	27	Symbol has the WSTRING type.
hla.ptCset	28	Symbol’s type is CSET.
hla.ptArray	29	The symbol is an array object.
hla.ptRecord	30	The symbol is a record object.
hla.ptUnion	31	The symbol is a union object.
hla.ptRegex	32	The symbol is a regular expression object.
hla.ptClass	33	The symbol is a class object.
hla.ptProcptr	34	The symbol’s type is "pointer to a procedure".
hla.ptThunk	35	The symbol is a THUNK type.
hla.ptPointer	36	The symbol is a POINTER object.

hla.ptLabel	37	The symbol is a statement label object.
hla.ptProc	38	The symbol denotes a procedure.
hla.ptMethod	39	The symbol denotes a method.
hla.ptClassProc	40	The symbol is a procedure within a class.
hla.ptClassIter	41	The symbol denotes an iterator within a class.
hla.ptIterator	42	The symbol is an iterator name.
hla.ptProgram	43	The symbol is the program's or unit's identifier.
hla.ptMacro	44	The identifier is a macro.
hla.ptText	45	The identifier is a text object (note: @ptype does not return this value since HLA expands the text prior to processing by @ptype).
Hla.ptRegExMac	46	The symbol is a regular expression macro.
hla.ptNamespace	47	The identifier is a namespace ID.
hla.ptSegment	48	The identifier is a segment ID.
hla.ptAnonRec	49	The identifier is an anonymous record within a union (internal use only, @ptype will never return this value).
hla.ptAnonUnion	50	The identifier is an anonymous union within a record (internal use only, @ptype will never return this value).
hla.ptVariant	51	This value is reserved for internal use by HLA.
hla.ptError	52	This value indicates a cascading error in an expression. Generally, you will not get this value from @ptype unless there was some sort of error in the parameter to pass to @ptype.

Note that the HLA module provides a constant array, hla.ptypeStrs, that returns the associated HLA type name when indexed by one of the above constants. Note that this is a compile-time constant array, not a run-time array of strings. If you want a run-time string array, you can define one thusly:

```
static
  ptypeStrs:string [@elements( hla.ptypeStrs )] := hla.ptypeStrs;
```

Do note that not every ptype value maps to a valid HLA data type. For example, hla.ptRecord maps to the string "(record)". You cannot use this as a type in an HLA program.

17.9 @pclass Return Values

The HLA @pClass function expects a procedure's parameter name as its sole parameter. It returns one of the following constants that denotes the parameter passing mechanism for the parameter. Note that @pClass' return values are defined only for parameter identifiers. These values have changed in the past and they will likely change in the future with improvements to the HLA language, so always use these symbolic names rather than hard-coded values.

Ταβλε 3 **@pClass Return Values**

Symbol	Value	Description
hla.illegal_pc	0	May be returned if the symbol is not a parameter.
hla.valp_pc	1	Returned if parameter is passed by value.
hla.refp_pc	2	@pClass returns this value if you pass the parameter by reference.
hla.vrp_pc	3	Denotes that you've passed the parameter by value/result.
hla.result_pc	4	This value means that you've passed the parameter by result.
hla.name_pc	5	This value indicates that you've passed the parameter by name.
hla.lazy_pc	6	This value indicates that you've passed the parameter by lazy evaluation.

17.10 @section Return Values

The following constants correspond to bits in the value returned by @section. They denote the current position of the compiler in the code. These values have changed in the past and they will likely change in the future with improvements to the HLA language, so always use these symbolic names rather than hard-coded values.

Ταβλε 4 **@section Constants**

Symbol	Value	Description
hla.inConst	1	Bit zero is set if HLA is current processing definitions in a CONST section.
hla.inVal	2	Bit one is set if HLA is current processing definitions in a VAL section.
hla.inType	4	Bit two is set if HLA is current processing definitions in a TYPE section.
hla.inVar	8	Bit three is set if HLA is current processing definitions in a VAR section.
hla.inStatic	$10	Bit four is set if HLA is current processing definitions in a STATIC section.
hla.inReadonly	$20	Bit five is set if HLA is current processing definitions in a READONLY section.
hla.inStorage	$40	Bit six is set if HLA is current processing definitions in a STORAGE section.

hla.inMain	$1000	Bit 12 is set if HLA is current processing statements in the main program.
hla.inProcedure	$2000	Bit 13 is set if HLA is current processing statements in a procedure.
hla.inMethod	$4000	Bit 14 is set if HLA is current processing statements in a method.
hla.inIterator	$8000	Bit 15 is set if HLA is current processing statements in an iterator.
hla.inMacro	$1_0000	Bit 16 is set if HLA is current processing statements in a macro.
hla.inKeyword	$2_0000	Bit 17 is set if HLA is current processing statements in a keyword macro.
hla.inTerminator	$4_0000	Bit 18 is set if HLA is current processing statements in a terminator macro.
hla.inThunk	$8_0000	Bit 19 is set if HLA is current processing statements in a thunk's body.
hla.inUnit	$80_0000	Bit 23 is set if HLA is current processing statements in a unit.
hla.inProgram	$100_0000	Bit 24 is set if HLA is current processing statements in a program (not a unit).
hla.inRecord	$200_0000	Bit 25 is set if HLA is current processing declarations in a record definition.
hla.inUnion	$400_0000	Bit 26 is set if HLA is current processing declarations in a union.
hla.inClass	$800_0000	Bit 27 is set if HLA is current processing declarations in a class.
hla.inNamespace	$1000_0000	Bit 28 is set if HLA is current processing declarations in a union.