Web Services Integration Toolkit for OpenVMS

Interface Definition File (IDL) Reference

June 2012

This document contains information that will help you to read or manually modify the WSIT IDL file.

Software Version
Web Services Integration Toolkit
Version 3.4-1

Hewlett-Packard Company
Palo Alto, California

About Web Services Integration Toolkit for OpenVMS Documentation

This *IDL Reference* describes the layout of the WSIT Interface Definition Language (IDL) file in order to make it as easy as possible for a developer to manually read and modify it.

The *Installation Guide and Release Notes* includes system requirements and installation instructions for OpenVMS, as well as release notes for the current release of the Web Services Integration Toolkit for OpenVMS.

The *Developer's Guide* contains information about how to use the tools in the Web Services Integration Toolkit for OpenVMS, and things to consider as you prepare your legacy application.

For the latest release information, refer to the Web Services Toolkit for OpenVMS web site at http://www.hp.com/products/openvms/webservices/.

Contents

1 O V E R V I E W

The Web Services Integration Toolkit (WSIT) provides tools to expose the API of an existing non-java application as a java based API. The first step in this process is to create an Interface Definition Language (IDL) file which describes the API to be wrapped. This IDL is an XML formatted file which may be created by a WSIT provided tool or it may be created by hand. In either case, it is often necessary to modify this file during the development of the WSIT application. This document will describe the XML tags and their relationship to each other.

A WSIT IDL file has an easy to understand nested layout that allows a developer to completely describe their application's interface in a language-neutral way. It does this by allowing the definition of all routines and structures that are to be exposed by the application. Within these routine and structure definitions, all parameter and field datatypes are mapped (translated) into their OpenVMS equivalent datatypes (DSC$K_DTYPE_*).

In general, the mapping of parameters within routines, and fields within structures, take one of the following two forms:

"User datatype specification" -> typedef translation[n] -> Primitive translation -> OpenVMS primitive (datatype)

-or-

"User datatype specification" -> typedef translation[n] -> Structure definition

Structure definitions contain an ordered set of fields which in turn go through the above translations until all definitions have been mapped into their equivalent OpenVMS datatypes.

Note: The `[n]` states that zero or more *typedef* translations may occur before the final translation to an OpenVMS datatype (primitive) or structure definition.

The sections below describe how each component (routine, structure, and so on), including translation (mapping), is defined within the WSIT IDL file.

2 <OpenVMSInterface> BLOCK

The <`OpenVMSInterface`> block s is the main block that encapsulates all of the blocks that collectively describe the application's interface. It has the following format:

```
<OpenVMSInterface
xmlns="hp/openvms/integration"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="hp/openvms/integration openvms-integration.xsd"
ModuleName="DISK:[MYDIR.STOCK]stock.OBJ"
Language="C89">
.
<Enumerations>
</Enumerations>
.
<Typedefs>
</Typedefs>
.
<Primitives>
```

```
</Primitives>
.
<Structures>
</Structures>
.
<Routines>
</Routines>
.
</OpenVMSInterface>
```

Except for the first line in the XML file, which is `<?xml version="1.0" encoding="UTF-8"?>`, all lines within the WSIT IDL file reside within the `<OpenVMSInterface>` block.

The properties within the `<OpenVMSInterface>` tag (in bold above) are primarily generic header information that is required, and is identical, for all WSIT IDL files. The two notable exceptions are the **ModuleName** and **Language** properties, described below:

Property Name	Description
`ModuleName`	For 3GL based applications, this is the fully qualified file specification of the OBJ module that contains the application's interface. For ACMS based applications, this is the ACMS application name.
`Language`	This is the language in which the interface module was written in, such as C89, BASIC, COBOL, and ACMS, …

The blocks nested within the `<OpenVMSInterface>` block contain the collection of definitions corresponding to that component or translation type. For example:

- The `<Routines>` block contains all of the individual <Routine> blocks that describe the exposed routines of an application.
- The `<Structures>` block contains the list of <Structure> definitions.
- The `<Primitives>` block contains the list of <Primitive> or OpenVMS datatype translations.
- The `<Typedefs>` block contains the list of <Typedef> definitions.
- The `<Enumerations>` block contains the list of <Enumeration> definitions.

All of the individual definition blocks and tags are described in greater detail below.

2.1 <Enumeration> Block

The collection of `<Enumeration>` blocks contains all of the constant definitions that are defined as enumerations within the application. An `<Enumeration>` is made up of a name, an OpenVMS datatype, an optional size in bytes, and the list of Name/Value pairs. The format of the block is as follows:

```
<Enumerations>
    <Enumeration Name = "myenums"
                 VMSDataType = "DSC$K_DTYPE_L"
                 ByteSize = "4">
        […see Enumerator block for more information…]
    </Enumeration>
</Enumerations>
```

The properties of the <Enumeration> tag are defined as follows:

Property Name	Description
Name	The name given to this collection of enumerators.
VMSDataType	The equivalent OpenVMS datatype of the enumeration. The DSC$K_DTYPE_* values are used to specify them in a language and application independent way.
ByteSize	The size, in bytes, of the specified datatype.

2.1.1 <Enumerator> Block

Each <Enumerator> within an enumerator collection (an Enumeration) specifies a name/constant value pair. These pairs make up the set of valid values for the Enumeration. The format of an Enumerator is:

```
<Enumeration …>
    <Enumerator Name = "PIC$SIZE1"  ConstantValue = "1"/>
    <Enumerator Name = "PIC$SIZE2"  ConstantValue = "2"/>
</Enumeration>
```

The properties within the <Enumerator> tag are as follows:

Property Name	Description
Name	The name given to the specified constant value.
ConstantValue	The constant value associated with the specified name.

2.2 <Typedef> Block

The collection of <Typedef> blocks contains all of the typedef translations used within the application. Each <Typedef> tag describes a user defined mapping of a type name to an equivalent type. In C, this would look something like:

```
typedef myint unsigned int;
```

Each <Typedef> tag has the following format:

```
<Typedef Name = "myint"
         TargetName = "unsigned int"/>
```

Where each property is defined below:

Property Name	Description
Name	The user defined name associated with the typedef within the application.
TargetName	The equivelant datatype that this typedef maps to. This may specify another typedef, a primitive, or a structure definition.

2.3 <Primitive> Block

The collection of <Primitive> blocks contains the datatype translations to their OpenVMS equivalents for all datatypes used within an application. Each <Primitive> mapping contains the datatype, the OpenVMS datatype (primitive) that it maps to, along with any other information needed to completely describe that primitive. The <Primitive> tag has the following formats.

The following describes a **Packed Decimal**:

```
<Primitive Name = "DSC$K_DTYPE_P_5_2"
           Size = "5"
           Scale = "2"
           VMSDataType = "DSC$K_DTYPE_P" />
```

The following describes a simple **Longword**:

```
<Primitive Name = "int"
           Size = "4"
           VMSDataType = "DSC$K_DTYPE_L" />
```

The following describe different string types (a **dynamic string** and a **varying string**):

```
<Primitive Name = "String_Dynamic"
           Size = "0"
           FixedFlag = "0"
           NullTerminatedFlag = "1"
           VMSDataType = "DSC$K_DTYPE_T" />
<Primitive Name = "Varying_String_20"
           Size = "20"
           FixedFlag = "0"
           NullTerminatedFlag = "1"
           VMSDataType = "DSC$K_DTYPE_VT" />
```

The properties of the <Primitive> tag are described below:

Property Name	Description
Name	The application or language specific name for the specified datatype, such as `unsigned int` or `PIC 9(8)`.**
VMSDataType	The equivalent OpenVMS datatype specification. The `DSC$K_DTYPE_*` values are used to specify them in a language neutral way.
Size	The size of the primitive being defined.*** This is ignored for datatypes whose size is constant, such as `DSC$K_DTYPE_L`. If the datatype is a string and the size is 0, then the string is considered dynamically sized.
Scale	Only used with scaled numeric datatypes, this property specifies the scale factor for the primitive being defined. Note that a positive scale factor specifies that the decimal point moves to the left. (The example above would represent a number with the format of 123.45.)

FixedFlag	Only used with string datatypes, this property specifies that the string being defined is of fixed size. A value of 1 specifies fixed size, while a 0 specifies that the string is dynamically sized.
NullTerminatedFlag	Only used with string datatypes, this property specifies if a null terminator should be appended to the end of the string. For fixed length strings, the string will be truncated if needed in order to append the null terminator. A value of 1 says to append a null, while a value of 0 specifies no null.
MemoryFreeByWSIT	Only used with BLOB datatypes. If set to 1, it specifies that WSIT should deallocate the memory that the user allocated once the call is complete. (Refer to Section 5.)

****** All Primitive Names must be unique. For primitives that are the same but differ in size and/or scale, one way force uniqueness is to embed the size and scale values into the Primitive Name itself. (See the Packed Decimal example above.)

******* The size is specified in bytes for all datatypes except Scaled Numerics, where the size specifies the number of digits. Note that for Varying Strings, the size specifies the maximum length of the string. (The actual length of the varying string is determined at runtime. If the varying string size is specified as 0, then the actual length at runtime is used as the maximum size.)

2.4 <Structure> Block

Although a language may refer to them as Records, Workspaces, or Structures, all languages support the concept of a structure. The collection of `<Structure>` blocks contains all of the user defined structure definitions that will be passed in or out of the application's interface. Each `<Structure>` block represents a single user defined structure definition. All parameters and fields must eventually map to an OpenVMS primitive, or one of these structure definitions. The format of the `<Structure>` block is:

```
<Structures>
      <Structure Name = "MyStruct"
                 TotalPaddedSize = "128">
            […See Field Block below for more information…]
      </Structure>
</Structures>
```

The properties of the `<Structure>` tag are:

Property Name	Description
Name	The user specified name given to this structure (record, workspace, …) definition.
TotalPaddedSize	The size of the structure, including any padding added for alignment purposes.

2.4.1 <Field> Block

Each `<Field>…</Field>` block describes a single field within a structure. The format of a `<Field>` block is as follows:

```
<Structure … >
     <Field Name = "Fld1"
            Type = "signed int"
            Offset = "0"/>
     <Field Name = "Fld2"
            Type = "FixedString16"
            Offset = "4"/>
     <Field Name = "AryFld3"
            Type = "signed int"
            Offset = "20"
            ArrayDimension = "1"
            RowByColumn = "0">
            <Array LowerBound = "0"
                   UpperBound = "9"/>
     </Field>
</Structure>
```

The properties of the `<Field>` tag are:

Property Name	Description
`Name`	The user specified name given to this field
`Type`	The language dependant or application specific datatype associated with this field. (Primitive, Typedef, Enumeration, or Structure.)
`Offset`	The offset (within the structure) to the start of this field.
`ArrayDimension`	If this field is an array of elements, this property specifies the number of dimensions within the array.
`RowByColumn`	If this field is a multi-dimensional array of elements, this property specifies the ordering of the dimensions within memory. All languages, except FORTRAN, use a `RowByColumn` layout. Use a 1 to specify `RowByColumn`, and a 0 to specify `ColumnByRow` (FORTRAN).
`<Array> Tag`	See below.

2.4.1.1 <Array> Tag

For fields and parameters that are arrays, the `<Array>` tag is used to specify dimension information for a single dimension. The number of `<Array>` tags must match the number specified in the `ArrayDimension` Field property above. The format of the `<Array>` tag is shown above.

The properties of the `<Array>` tag are:

Property Name	Description
`LowerBound`	The user specified lower bound of this dimension.
`UpperBound`	The user specified upper bound of this dimension. Note that the upper bound must be larger than the lower bound.

Note: **T**he size of each array dimension is `UpperBound - Lowerbound + 1`.

2.5 <Routine> Block

The collection of <Routine> blocks contains all of the definitions for the routines being exposed by the application. Each <Routine> block contains the complete description of a single exposed routine call, including all parameter and return information. The format for the <Routine> block is as follows:

```
<Routines>
      <Routine Name = "MyRoutine"
               ReturnType = "unsigned int"
               Description = "This is the description for my routine">
          […Refer to the Parameter block section below for more information…]
      <\Routine>
<\Routines>
```

The properties of the <Routine> tag are:

Property Name	Description
Name	The user specified name for this exposed routine.
ReturnType	The language dependant or application specific datatype associated with this routine's return type. (*The return type can not be a structure, string, or scaled numeric.)
Description	A user specified description to be associated with this routine definition.
MethodID	Species a value to use as the internal method ID instead of the one automatically generated for this routine. This is only useful in rare cases for backwards compatibility within the generated interface.
<Parameter> Tag	See below for more information.

2.5.1 <Parameter> Block

The <Parameter> block is used to describe a single parameter within a routine's parameter list. There will be one <Parameter> tag for each parameter passed in or out of the routine. The <Parameter> block has the following format:

```
<Routine …>
   <Parameter Name = "Param1"
              Type = "unsigned int"
              PassingMechanism = "Value"
              Usage = "IN"/>
   <Parameter Name = "AryParam2"
              Type = "__int16"
              PassingMechanism = "Reference"
              Usage = "IN/OUT"
              ArrayDimension = "1"
              RowByColumn = "1"
              ArrayDescriptorType = "DSC$K_CLASS_A">
              <Array LowerBound = "0"
                     UpperBound = "10"/>
   </Parameter>
   <Parameter Name = "AryParam3"
              Type = "__int16"
              PassingMechanism = "Descriptor"
```

```
                Usage = "IN/OUT"
                ArrayDimension = "1"
                RowByColumn = "1"
                ArrayDescriptorType = "DSC$K_CLASS_A">
    </Parameter>
</Routine>
```

The <Parameter> tag has the following properties:

Property Name	Description
Name	The user specified name for this parameter.
Type	The language dependant or application specific datatype associated with this parameter type. (Primitive, Typedef, Enumeration, or Structure.)
PassingMechanism	The OpenVMS based passing mechanism used to pass this parameter. It can be Value, Reference, or Descriptor.
Usage	This property specifies how this parameter will be effected by the called routine. It is either IN, which specifies that it doesn't modify the value, or IN/OUT which specifies that it does modify this value.
ArrayDimension	If this parameter is an array of elements, this property specifies the number of dimensions within the array.
RowByColumn	If this parameter is a multi-dimensional array of elements, this property specifies the ordering of the dimensions within memory. All languages, except FORTRAN, use a RowByColumn layout. Use a 1 to specify RowByColumn, and a 0 to specify ColumnByRow (FORTRAN).
ArrayDescriptorType	If this parameter is an array passed by descriptor, this property specifies the descriptor class that should be used when passing this array. The valid values for this property are DSCK_CLASS_A, DSCK_CLASS_NCA, DSC$K_CLASS_VSA.
<Array> Tag	See below for more information.

2.5.1.1 <Array> Tag

For field arrays, and parameter arrays that are passed by reference, the <Array> tag is used to specify dimension information for a single dimension. For Parameter arrays that are passed by reference, the number of <Array> tags must match the number specified in the ArrayDimension Parameter property above. The format of the <Array> tag is shown above.

The properties of the <Array> are:

Property Name	Description
LowerBound	The user specified lower bound of this dimension.
UpperBound	The user specified upper bound of this dimension. Note that the upper bound must be larger than the

	lower bound.

Note: The size of each array dimension is `UpperBound - Lowerbound + 1`. The `<Array>` tags are not needed when the array parameter is being passed by descriptor. (The dimension information is pulled from the descriptor at runtime.)

2.6 Valid Property Values

The following table lists all of the datatypes supported by WSIT, along with their associated property values. Note that if the property box is empty, this property need not be specified for that datatype within the IDL.

Type Description	OpenVMS Datatype	Size Value	Scale Required	Null Terminated Flag Value	Fixed Flag Value
Binary Types					
Byte (8 bits)	DSC$K_DTYPE_B				
Unsigned Byte	DSC$K_DTYPE_BU				
Word (16 bits)	DSC$K_DTYPE_W				
Unsigned Word	DSC$K_DTYPE_WU				
Longword (32 bits)	DSC$K_DTYPE_L				
Unsigned Longword	DSC$K_DTYPE_LU				
Quadword (64 bits)	DSC$K_DTYPE_Q				
Unsigned Quadword	DSC$K_DTYPE_QU				
Octaword (128 bits)	DSC$K_DTYPE_O				
Unsigned Octaword	DSC$K_DTYPE_OU				
Float (32 bits)	DSC$K_DTYPE_F				
Double(D) Float (64 bits)	DSC$K_DTYPE_D				
Double(G) Float (64 bits)	DSC$K_DTYPE_G				
Quad Float (128 bits)	DSC$K_DTYPE_H				
IEEE Float (32 bits)	DSC$K_DTYPE_FS				
IEEE Double Float (64 bits)	DSC$K_DTYPE_FT				
IEEE Quad Float (128 bits)	DSC$K_DTYPE_FX				
String Types					
Static String	DSC$K_DTYPE_T	length		0	1
C Style String**	DSC$K_DTYPE_T	0		1	0
Dynamic String**	DSC$K_DTYPE_T	0		1	0
Varying String	DSC$K_DTYPE_VT	length		0	0
Scaled Numeric Types					
Packed Numeric	DSC$K_DTYPE_P	digits	scale		
Unsigned Numeric	DSC$K_DTYPE_NU	digits	scale		
Left Separate Sign Numeric	DSC$K_DTYPE_NL	digits	scale		
Left Overpunch Sign Numeric	DSC$K_DTYPE_NLO	digits	scale		
Right Separate Sign Numeric	DSC$K_DTYPE_NR	digits	scale		
Right Overpunch Sign Numeric	DSC$K_DTYPE_NRO	digits	scale		
Zoned Decimal Numeric	DSC$K_DTYPE_NZ	digits	scale		

Special Types					
Structures	Structure Definition Name				
Binary Large Objects (BLOB)	DSC$K_DTYPE_ BLOB				

****** Dynamic Strings differ from C Style strings in the mechanism used to pass them within a parameter. Dynamic Strings are passed exclusively by Descriptor, while C Style Strings are passed by Reference.

******* Binary Large Objects (BLOBs) are not valid OpenVMS datatypes. This type is handled internally by WSIT and is discussed in Section 5.

3 EXAMPLE WSIT IDL FILE

The following code is an example of a WSIT IDL file.

```
<?xml version="1.0" encoding="UTF-8"?>
<OpenVMSInterface
       xmlns="hp/openvms/integration"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="hp/openvms/integration openvms-integration.xsd"
       ModuleName="disk$:[workshop.lab1]math.obj"
       Language="C89">
       <Primitives>
         <Primitive Name = "unsigned int"
                    Size = "4"
                    VMSDataType = "DSC$K_DTYPE_LU"/>
         <Primitive Name = "signed int"
                    Size = "4"
                    VMSDataType = "DSC$K_DTYPE_L"/>
         <Primitive Name = "FixedString16"
                    Size = "16"
                    FixedFlag = "1"
                    NullTerminatedFlag = "1"
                    VMSDataType = "DSC$K_DTYPE_T" />
       </Primitives>
       <Structures>
                <Structure Name = "MyStruct"
                              TotalPaddedSize = "60">
                <Field Name = "Fld1"
                               Type = "signed int"
                               Offset = "0"/>
                       <Field Name = "Fld2"
                              Type = "FixedString16"
                              Offset = "4"/>
                       <Field Name = "AryFld3"
                              Type = "signed int"
                               Offset = "20"
                               ArrayDimension = "1"
                               RowByColumn = "0">
                               <Array LowerBound = "0"
                                      UpperBound = "9"/>
                       </Field>
                </Structure>
       </Structures>
```

```
        <Routines>
          <Routine Name = "sum"
                   ReturnType = "unsigned int">
                   <Parameter Name = "number1"
                              Type = "signed int"
                              PassingMechanism = "Value"
                              Usage = "IN"/>
                   <Parameter Name = "number2"
                              Type = "signed int"
                              PassingMechanism = "Value"
                              Usage = "IN"/>
          </Routine>
          <Routine Name = "product"
                   ReturnType = "unsigned int">
                   <Parameter Name = "number1"
                              Type = "signed int"
                              PassingMechanism = "Value"
                              Usage = "IN"/>
                   <Parameter Name = "number2"
                              Type = "signed int"
                              PassingMechanism = "Value"
                              Usage = "IN"/>
                   <Parameter Name = "structparam"
                              Type = " MyStruct "
                              PassingMechanism = "Reference"
                              Usage = "IN"
                              ArrayDimension = "1"
                              RowByColumn = "1"
                              ArrayDescriptorType = "DSC$K_CLASS_A">
                              <Array LowerBound = "0"
                                     UpperBound = "9"/>
                   </Parameter>
          </Routine>
        </Routines>
</OpenVMSInterface>
```

4 MAPPING LANGUAGE DEFINITIONS TO IDL

The following sections describe the various types of language interfaces in the IDL.

4.1 Mapping Binary Types

The easiest way to define an interfaces for routines is to pass simple binary datatypes. A routine with a function prototype that looks similar to the following:

```
        int MyAdd (int p1, int p2);
```

Would be defined in the WSIT IDL as follows:

```
…
<Primitives>
 <Primitive Name = "signed int"
            VMSDataType = "DSC$K_DTYPE_L"/>
</Primitives>
```

```
<Routines>
 <Routine Name = "MyAdd"
          ReturnType = "int">
          <Parameter Name = "p1"
                     Type = "int"
                     PassingMechanism = "Value"
                     Usage = "IN"/>
          <Parameter Name = "p2"
                     Type = "int"
                     PassingMechanism = "Value"
                     Usage = "IN"/>
 </Routine>
</Routines>
```

The function definition above simply passes in longwords by value. Since the parameters are passed in by value, the called routine cannot modify them so that the caller can pick up the new values. However, if you want the caller to pick up the new values, you can change the routine to have the parameters passed in by reference, as follows:

```
        Int MyAdd (int *p1, int *p2);
```

The corresponding change to the WSIT IDL file are as follows (the modified lines are bolded):

```
…
<Primitives>
 <Primitive Name = "signed int"
            VMSDataType = "DSC$K_DTYPE_L"/>
</Primitives>
<Routines>
 <Routine Name = "MyAdd"
          ReturnType = "int">
          <Parameter Name = "p1"
                     Type = "int"
                     PassingMechanism = "Reference"
                     Usage = "IN/OUT"/>
          <Parameter Name = "p2"
                     Type = "int"
                     PassingMechanism = "Reference"
                     Usage = "IN/OUT"/>
 </Routine>
</Routines>
```

4.2 Mapping Decimal Types

For languages that support them, the second easiest datatypes to define and use within a routine are the Scaled Decimal Numeric datatypes. For example, a BASIC subroutine that is defined as:

```
           SUB MYADD (DECIMAL(5,2) P1, DECIMAL(5,2) P2, DECIMAL(5,2) SUM)
                 …
           END SUB
```

Would be defined within the WSIT IDL file as follows:

```
…
<Primitives>
```

```
 <Primitive Name = "decimal_5_2"
        Size = "5"
        Scale = "2"
        VMSDataType = "DSC$K_DTYPE_P"/>
</Primitives>

<Routines>
 <Routine Name = "MYADD">
          <Parameter Name = "P1"
                     Type = "decimal_5_2"
                     PassingMechanism = "Reference"
                     Usage = "IN"/>
          <Parameter Name = "P2"
                     Type = "decimal_5_2"
                     PassingMechanism = "Reference"
                     Usage = "IN"/>
          <Parameter Name = "SUM"
                     Type = "decimal_5_2"
                     PassingMechanism = "Reference"
                     Usage = "IN/OUT"/>
 </Routine>
</Routines>
```

Note: Although all three parameters are passed by reference, the usage for P1 and P2 are defined to be `IN` only. This can happen when the called routine does not modify these parameters, or when the modified values in the client are unimportant. Specifying `IN` only where possible creates a cleaner interface in the generated JavaBean interface.

Scaled Decimal Numeric datatypes have the following limitations:

- The Size (in digits) cannot be larger than 31.

4.3 Mapping String Types

Of all the primitive datatypes, the String datatypes are the most challenging to define. This is because (depending on the language) it is their usage, and not their interface definition, that determines what kind of string it is. For example, these function prototypes all represent different uses of the string datatype, all of which need to be defined differently within the WSIT IDL file.

```
int fixedStrRoutine ( char *P1 );    \\ P1 is a 20 character fixed string
int CStrRoutine ( char *P1 );        \\ P1 is a null terminated C string
int varyingStrRoutine ( char *P1 );  \\ P1 is a length prefixed varying
                                        string (max 20)
int charAryRoutine ( char *P1 );      \\ P1 is an array of chars (20 elements)
```

The associated WSIT IDL definitions would look similar to the following:

```
…
<Primitives>
<Primitive Name = "Fixed_String_20"
           Size = "20"
           FixedFlag = "1"
           NullTerminatedFlag = "0"
           VMSDataType = "DSC$K_DTYPE_T" />
```

```
<Primitive Name = "C_String"
           Size = "0"
           FixedFlag = "0"
           NullTerminatedFlag = "1"
           VMSDataType = "DSC$K_DTYPE_T" />

<Primitive Name = "Varying_String_20"
           Size = "20"
           FixedFlag = "0"
           NullTerminatedFlag = "1"
           VMSDataType = "DSC$K_DTYPE_VT" />

<Primitive Name = "char"
           VMSDataType = "DSC$K_DTYPE_B"/>

<Primitive Name = "int"
           VMSDataType = "DSC$K_DTYPE_L"/>
</Primitives>

<Routines>
 <Routine Name = "fixedStrRoutine"
                 ReturnType = "int">
                 <Parameter Name = "P1"
                            Type = "Fixed_String_20"
                            PassingMechanism = "Reference"
                            Usage = "IN/OUT"/>
 </Routine>

 <Routine Name = "CStrRoutine"
                  ReturnType = "int">
                  <Parameter Name = "P1"
                             Type = "C_String"
                             PassingMechanism = "Reference"
                             Usage = "IN/OUT"/>
 </Routine>

 <Routine Name = "varyingStrRoutine"
                  ReturnType = "int">
                  <Parameter Name = "P1"
                             Type = "Fixed_String_20"
                             PassingMechanism = "Reference"
                             Usage = "IN/OUT"/>
 </Routine>

 <Routine Name = "charAryRoutine"
                  ReturnType = "int">
                  <Parameter Name = "P1"
                             Type = "char"
                             PassingMechanism = "Reference"
                             Usage = "IN/OUT"
                             ArrayDimension = "1"
                             RowByColumn = "1"
                             ArrayDescriptorType = "DSC$K_CLASS_A">
                             <Array LowerBound = "0"
                                    UpperBound = "19"/>
                  </Parameter>
```

```
 </Routine>
</Routines>
```

String Datatypes have the following limitations:

- The Size must be 0 <= Size <= 65535

4.4 Mapping Arrays

Arrays can be easily defined within the WSIT IDL file as long as you know the following minimum information about the arrays being passed in/out of your routines:

- What is the datatype of the array? (int, float, string, …)
- How many dimensions does the array have? (Usually 1)

However, you may also need to know more information about the array, depending on the usage of the array.

If the array is a parameter passed by reference, or is a field within a structure, then you need to know:

- The lower and upper bounds of each dimension. (1 set of values for each dimension)

If the array parameter is passed by descriptor, then you do not need to specify lower and upper bounds, but you must correctly specify:

- The Array Descriptor Class to use when passing the array. (DSC$K_CLASS_A, …)

If the array is multi-dimensional, you must correctly specify:

- If the multi-dimensional array is RowByColumn or not. (True for all languages except Fortran?)

For example, the following C function prototype, which defines an array of fixed length strings:

```
int myAryRoutine (char P1[100][20]);
```

Would be defined as follows within the WSIT IDL file:

```
 …
<Primitives>
…
<Primitives>
<Primitive Name = "Fixed_String_20"
            Size = "20"
            FixedFlag = "1"
            NullTerminatedFlag = "0"
            VMSDataType = "DSC$K_DTYPE_T" />

<Primitive Name = "int"
           VMSDataType = "DSC$K_DTYPE_L"/>
</Primitives>

<Routines>
 <Routine Name = "myAryRoutine"
```

```
          ReturnType = "int">
          <Parameter Name = "P1"
                     Type = "Fixed_String_20"
                     PassingMechanism = "Reference"
                     Usage = "IN/OUT"
                     ArrayDimension = "1"
                     RowByColumn = "1"
                     ArrayDescriptorType = "DSC$K_CLASS_A">
                     <Array LowerBound = "0"
                            UpperBound = "99"/>
          </Parameter>
 </Routine>
</Routines>
```

Note: Because P1 is a single dimensional array passed by Reference in this case, neither the `RowByColumn` nor the `ArrayDescriptorType` properties are important. (However, because the WSIT IDL Schema requires their use, you must specify them in the IDL.)

4.5 Mapping Structures

Describing the structures that are passed in and out of your routine(s) within WSIT IDL is straightforward. The only items that you need to keep careful watch over are the field offsets. These define the starting byte offset of each field within the structure. Note that the following items must be taken into careful consideration in order to define these offsets correctly.

- Alignment requirements of each field (appropriate padding must be added into offsets.)
- Size differences of the different Scaled Numeric datatypes.

You must also correctly specify the structure's overall `TotalPaddedSize`. This value is referenced whenever the structure is used within an array, or nested within another structure.

Assuming *natural alignment* is being used, the following C structure examples:

```
struct Struct1 {
       char    f1;
       int     f2;
}

struct Struct2 {
       short   f1;
       int     f2;
       struct Struct1 f3;
       char    f4[9];
}
```

Would be defined as follows within the WSIT IDL file:

```
<Primitives>
  <Primitive Name = "char"
             VMSDataType = "DSC$K_DTYPE_B"/>
  <Primitive Name = "short"
             VMSDataType = "DSC$K_DTYPE_W"/>
  <Primitive Name = "int"
             VMSDataType = "DSC$K_DTYPE_L"/>
```

```
   <Primitive Name = "Fixed_String_9"
              Size = "9"
              FixedFlag = "1"
              NullTerminatedFlag = "0"
              VMSDataType = "DSC$K_DTYPE_T" />
</Primitives>

  <Structures>
     <Structure Name = "Struct1"
                TotalPaddedSize = "8">
                <Field Name = "f1"
                       Type = "char"
                       Offset = "0"/>
                <Field Name = "f2"
                       Type = "int"
                       Offset = "4"/>
     </Structure>

     <Structure Name = "Struct2"
                TotalPaddedSize = "28">
                <Field Name = "f1"
                       Type = "short"
                       Offset = "0"/>
                <Field Name = "f2"
                       Type = "int"
                       Offset = "4"/>
                <Field Name = "f3"
                       Type = "Struct1"
                       Offset = "8"/>
                <Field Name = "f4"
                       Type = " Fixed_String_9"
                       Offset = "16"/>
     </Structure>
  </Structures>
```

5 MAPPING BLOBS AND OTHER UNFORMATTED DATA

As discussed in previous sections, WSIT gives you a clean way to define almost every OpenVMS primitive and aggregate type within WSIT's IDL file. However, some applications may require a large non-typed chunk of memory to be exchanged with it. This may be needed if you want to:

- Exchange a string larger than 65535 with your application
- Exchange a non-standard datatype with your application

In these cases, you treat the parameter as a **Binary Large Object (BLOB)** which you can describe in the WSIT IDL as well. Conceptually, a BLOB is a large chunk of memory whose contents is in a format unknown to the underlying runtime. WSIT will not attempt to interpret it when passed into a routine. (The contents only have meaning to the application's java client(s) and user routine(s).) Internally, WSIT handles a BLOB like a resizable array of bytes passed by descriptor. Because of this, the application routine that is to be passed a BLOB must also have the BLOB parameter defined as an array of bytes passed by descriptor.

An example C function prototype is as follows:

```
            int myStringRtn (struct dsc$descriptor_a *p1);
```

Defining the BLOB parameter within the WSIT IDL is a simple matter of creating a BLOB primitive type, then assigning the parameter to this new type:

```
…
<Primitives>
  <Primitive Name = "myblob"
             MemoryFreeByWSIT = "1"
             VMSDataType = "DSC$K_DTYPE_BLOB" />
  <Primitive Name = "int"
             VMSDataType = "DSC$K_DTYPE_L"/>
</Primitives>

<Routines>
  <Routine Name = "myStringRoutine"
           ReturnType = "int">
           <Parameter Name = "p1"
                      Type = "myblob"
                      PassingMechanism = "Descriptor"
                      Usage = "IN/OUT"/>
  </Routine>
</Routines>
```

Notice that there is a new property called `MemoryFreeByWSIT` and it is set to 1. This tells the WSIT runtime to deallocate this memory once it has finished returning the contents to Java. Unless you plan to explicitly deallocate the new memory in a later call, you should let WSIT deallocate this memory for you on return. If you are planning on handling deallocation yourself at a later time, then specify 0 for `MemoryFreeByWSIT`, or don't specify this property at all.

Note that when working with BLOBs, it is the responsibility of the user's routine to correctly modify the array descriptor which is passed in. If the array descriptor isn't correctly updated to reflect the new size and memory location, the WSIT runtime will not pass the BLOB back correctly. A sample C routine that handles this is as follows:

```
int myStringRoutine ( struct dsc$descriptor_a *adx )
{
    char    *somestring =
            "ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890ABCDEFGHIJKLMNOPQR";
    int     status = 0;
    int     newarysize = 50;
    char    *newmem = NULL;

    // Allocate memory for the new BLOB
    newmem = malloc(newarysize);

    // Fill in the new BLOB with some information
    memcpy(newmem, somestring, newarysize-1);
    newmem[newarysize-1] = '\0';

    //Tell WSIT about the memory...
    adx->dsc$a_pointer = newmem;
    adx->dsc$l_arsize = newarysize;

    return status;
}
```

One benefit of treating BLOBs as byte arrays is that in Java, the String class contains constructors and methods that make converting from a byte array to a String and back again a straightforward process.

*Note that WSIT requires the client to always pass in a valid array for BLOB parameters. If an empty *ObjectHolder*, or a null, is passed in to the routine WSIT will throw an exception.
