OpenClovis Software Development Kit (SDK) Service Description and API Reference for Interface Definition Language (IDL) Service

For OpenClovis SDK Release 2.3 V0.4

Document Revision Date: March 27, 2007

Contents

Chapter 1

Functional Overview

OpenClovis ASP is a middleware that provides high availability and system manageability solutions in a distributed system. The components on various nodes interact with each other to provide the required services. Thus, it is imperative to provide a simple and efficient means of communication between components, which has to be agnostic to platform architecture issues and native data formats (32/64 bit architecture).

To provide such a communication process, a message passing mechanism with Remote Procedure Call (RPC) semantics is necessary. RPC provides the ability to call functions that are spread across remote components as if the functions are local. This functionality is provided by the OpenClovis service called Remote Method Dispatch (RMD).

RMD has an engine that provides the ability to send messages from the source component to a destination component. The source component, where the call originates, is called the Client and the destination component is called the Server. The RMD engine provides an interface to send and receive messages. Typically, the application is responsible for handling cross-platform issues such as endianness and function calling semantics such as synchronous calls, asynchronous calls, and function identification.

The OpenClovis Interface Definition Language (IDL) service, generates a code that handles cross-platform issues, function calling semantics, and uses RMD to send and receive messages. The IDL generator accepts the functions or interface exposed by a component, relevant programmer-defined data type definitions in XML format, and generates the code which can be used by the applications.

Chapter 2

Service Model

2.1 Usage Model

TBD

2.2 Functional Description

The IDL Library provides an IDL handle that defines the communication semantics and provides APIs that perform the following:

- Initialize the handle with RMD parameters
- Update the handle
- Finalize the handle

While initializing the IDL library, you must specify the structure, `ClIdlHandleObjT`. This contains the RMD parameters such as RMD options, destination address, and flags. The IDL engine is a Python script that takes two arguments and generates the code. The arguments are:

- XML file - Provides a specification of the interface exposed by the components and data structures. The interface is a set of operations that a client can invoke.
- Directory - Contains the generated client, server, and xdr routines.

IDL generates a Makefile. The server, client, and XDR libraries are generated when the Makefile is run. You have to link the code with the client and XDR libraries and include the client and XDR headers in the code, to invoke remote functions. At the server-side, you have to provide function definitions corresponding to the specification in the XML file. You must install the server stubs so that the client application can use them and uninstall them after they are used.

The directory contains subdirectory for every service specified in the XML file, which in turn contains the following:

- Top level Makefile - This recursively descends into the client, server, and XDR directories.
- XDR - This directory contains:

  - .h file generated for every user-defined data structure in the XML file.
  - `xdr<type name>Marshall.c` and `xdr<type name>Unmarshall.c` files generated for every user-defined data type.
  - Makefile that generates a library from the files in this XDR directory.

- Client - This directory contains:
  - .c and .h files generated for every service defined in the XML file.
  - Definition and declaration of the synchronous and asynchronous functions defined in the XML file.
  - Definition and declaration of the asynchronous callback function defined in the XML file.
  - Makefile that generates a library from the files in this Client directory.

- Server - This directory contains:
  - .c and.h files generated for every service defined in the XML file.
  - Definition and declaration of a server function for every operation defined in the XML file.
  - Makefile that generates a library from the files in this Server directory.
  - `responseSend` function for every server stub defined.
  - `cl<Eo>IdlSyncDefer` function, which is generated in the `<Eo>server.h` and `<Eo>server.c file`.

2.2 Functional Description

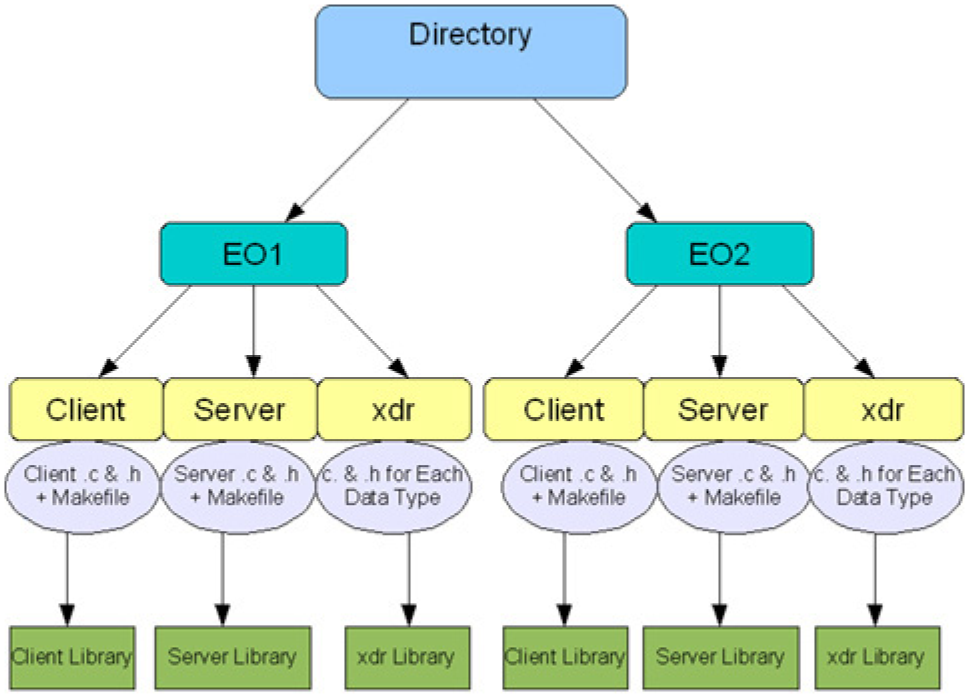
Directory
EO1
EO2
Client
Server
xdr
Client
Server
xdr
Client .c & .h + Makefile
Server .c & .h + Makefile
.c & .h for Each Data Type
Client .c & .h + Makefile
Server .c & .h + Makefile
.c & .h for Each Data Type
Client Library
Server Library
xdr Library
Client Library
Server Library
xdr Library

Figure 2.1: IDL Directory Structure

The IDL uses XDR to store data in a message. This is called marshalling of data. XDR stands for eXternal Data Representation Standard (described in RFC 1832). It is used to store data in a machine independent format. The reverse process of retrieving data from the message into the native format is called unmarshalling of data. IDL recognizes the following data types:

- `ClCharT`
- `ClInt8T`
- `ClUint8T`
- `ClInt16T`
- `ClUint16T`
- `ClInt32T`
- `ClUint32T`
- `ClInt64T`
- `ClUint64T`
- `ClNameT`
- `ClVersionT`
- `struct` formed from other data types
- `union` formed from other data types
- enum

- Pointers and arrays of the mentioned data types.

All data structures map to relevant C structures except for `union`. A union is implemented as a structure that aggregates the respective C-style `union` and a member called discriminant that has the appropriate values for every member of the C-style `union`.

2.2.0.1 Relevant XDR Details

In XDR, data is stored as multiples of 4 bytes (by padding, if required) in the network byte order (big endian) format. A byte is padded with 3 bytes and the 4-byte word is converted to big endian, if required.

Arrays are stored as a sequence of N elements. The sequence is then padded and converted to the network byte order.

Pointers are stored as 4-byte length of the number of elements being pointed to, followed by a sequence of elements (padded as required) in the network byte order.

Structures are stored as an aggregation of the individual members. They are always multiples of 4-bytes.

Unions are stored as a 4-byte discriminant (an identification of the member of the union being stored) and the member being used.

If there are pointers present inside structures, they must be allocated from the heap. When the heap is used for allocation/de-allocation, destruction of variables becomes non-trivial as arbitrary levels of nesting for data structures are allowed. It is logical to embed the code for the destruction of structures in the marshalling code. Thus, marshalling code can also be used for destroying variables, if required.

The algorithm for marshalling/unmarshalling arrays and pointers is separate from the data structure to which the array/pointer belongs. Marshalling/unmarshalling functions for arrays and pointers are wrappers which call the marshalling/unmarshalling functions for the relevant data type multiple times.

2.2.0.2 Design of XDR APIs

The marshalling API requires the following information:

- Reference to the data structure to marshall, with relevant information such as discriminant, length variable, and so on.
- Message, in which the marshalled data is to be stored.
- Flag that indicates if a variable needs to be destroyed.

Thus the API signature is as follows:

`ClRcT clXdrMarshall<Struct>(void* pGenVar, ClBufferMessageHandleT msg, ClUint32T isDelete)` where:

- `pGenVar` is the reference to the data structure to be marshalled.
- `msg` is the message in which the marshalled data is stored (marshalling is performed only if msg != 0).
- `isDelete` is the flag that indicates if `pGenVar` should be destroyed.

The corresponding unmarshalling API is:

`ClRcT clXdrUnmarshall<Struct>(ClBufferMessageHandleT msg, void* pGenVar)` where:

- `pGenVar` is the reference to the data structure, where the marshalled data is to be stored.
- `msg` is the message that is to be unmarshalled.

This ensures that the individual elements of a data structure are handled appropriately. However, pointers or arrays must be handled differently. The marshalling/unmarshalling algorithms for a structure containing pointers/arrays are:

```
#define clXdrMarshallArray<Struct>(pointer, multiplicity, msg,
isDelete) clXdrMarshallArray((pointer), sizcomponentf(<Struct>),
(multiplicity), clXdrMarshall<Struct>, (msg), (isDelete))
```

```
#define clXdrUnmarshallArray<Struct>(pointer, multiplicity, msg)
clXdrUnmarshallArray((pointer), sizcomponentf(<Struct>),
(multiplicity), clXdrUnmarshall<Struct>, (msg))
```

```
#define clXdrMarshallPointer<Struct>(pointer, multiplicity, msg,
isDelete) clXdrMarshallPointer((pointer), sizcomponentf(<Struct>),
(multiplicity), clXdrMarshall<Struct>, (msg), (isDelete))
```

```
#define clXdrUnmarshallPointer<Struct>(msg,pointer)
clXdrUnmarshallPointer(msg, sizcomponentf(<Struct>),
clXdrUnmarshall<Struct>, pointer)
```

To marshall pointers/arrays, it is not enough to pass the pointer to the data structure alone. It is necessary to pass the multiplicity of the elements being pointed to. These generic algorithms call the element marshalling/unmarshalling algorithms appropriately, after performing the necessary work on pointers/arrays.

2.2.0.3 XDR Algorithms

Marshalling Algorithms

In all marshalling algorithms, the message is written only if it is non-zero. The message should be 0, if the variable under consideration needs to be deleted. Data is converted to the network byte order before it is written to the message.

Pointer:

parameters:

```
void* pointer,

ClUint32T typeSize,

ClUint32T multiplicity,

ClXdrMarshallFuncT func,

ClBufferMessageHandleT msg,

ClUint32T isDelete
```

Algorithm

1. Write the `multiplicity` into `msg`

2. Return, if `multiplicity` is zero
3. Call the array algorithm with relevant parameters such as `pointer`, `typeSize`, `multiplicity`, `func`, `msg`, and `isDelete`.
4. If `isDelete`, delete the memory being pointed to

Array:

parameters:

`void* array,`

`ClUint32T typeSize,`

`ClUint32T multiplicity,`

`ClXdrMarshallFuncT func,`

`ClBufferMessageHandleT msg,`

`ClUint32T isDelete`

Algorithm

1. Return if `multiplicity` is 0
2. Call `func` for every element in `array`
3. Increment array by `typeSize`.

Struct:

parameters:

`void* struct,`

`ClBufferMessageHandleT msg,`

`ClUint32T isDelete`

Algorithm

1. Call the appropriate marshalling functions (`andstruct-><member>, msg, isDelete`) for every member of the structure.

Union:

parameters:

`void* union,`

`ClBufferMessageHandleT msg,`

`ClUint32T isDelete`

Algorithm

1. The `union->discriminant` is written to `msg`
2. Call the marshalling function (`andunion-><union>.<member>, msg, isDelete`) based on `union->discriminant`.

The marshalling function for basic types is predefined and writes the variable into the message. The functions for array and pointer to data types, that are smaller than 4 bytes, are also predefined as they require additional effort of padding.

Unmarshalling Algorithms

In unmarshalling algorithms, the message should always be non-zero. Data is converted to the host order before it is read from a message.

Pointer:

parameters:

`void** pointer,`

`ClUint32T typeSize,`

`ClXdrUnmarshallFuncT func,`

`ClBufferMessageHandleT msg,`

Algorithm

1. Read the `multiplicity` from `msg`
2. If `multiplicity` is zero, set `*pointer` to NULL and return
3. Allocate `*pointer` as per `multiplicity` from heap
4. Call the array algorithm with relevant parameters such as `*pointer`, `multiplicity`, `func`, and `msg`.

Array:

parameters:

`void* array,`

`ClUint32T typeSize,`

`ClUint32T multiplicity,`

`ClXdrMarshallFuncT func,`

`ClBufferMessageHandleT msg,`

Algorithm

1. Return if `multiplicity` is 0
2. Call `func` for every element in `array`
3. Increment array by `typesize`

Struct:

parameters:

`void* struct,`

`ClBufferMessageHandleT msg,`

Algorithm

1. Call the appropriate unmarshalling functions (`andstruct-><member>, msg`), for every member of the structure.

Union:

parameters:

`void* union,`

`ClBufferMessageHandleT msg`

Algorithm

1. The `union->discriminant` is read from `msg`
2. Call the unmarshalling function (`andunion-><union>.<member>, msg, isDelete`) based on `union->discriminant`

The unmarshalling function for basic types is predefined and reads the variable from the message. The functions for arrays and pointers to sub-4-byte data types are also predefined as they require the additional effort of padding.

2.2.1 IDL Stubs

IDL generates the following for every function installed in the component interface:

- Synchronous client stub
- Asynchronous client stub
- Server stub
- Asynchronous callback

In addition, it generates two functions to install and uninstall the server functions into the component. A client function is defined by it's arguments. Arguments or parameters are marshalled into a message that is passed to the functions residing on the server and vice versa. It is necessary to understand the type of arguments that can be available and how marshalling/unmarshalling can be performed. The arguments can be classified as follows:

Classification based on direction of transfer:

- `CL_IN` - These are provided as an input to a function. They are sent to the server through an input message.
- `CL_OUT` - These are used to return results to the caller. These parameters are sent from the server to the client in the output message.
- `CL_INOUT` - These act as both input and output arguments. Consequently, they are sent in both directions

Classification based on passing parameters by reference or by value:

- By value - This is passed as a copy. The original variable is not modified.
- By reference - This is passed as an address. The original variables are modified. As IDL marshalls/unmarshalls the parameters, these are further categorized as:
  - Without count - When pointers are passed without a count (size of the array), they are assumed to point to a single element.

  – With count - When a reference is passed with a count variable (an input basic type parameter), the count variable is assumed to contain the number of elements being pointed to.

An argument can be classified into one of the following categories:

- `CL_IN`, by value (as `CL_INOUT` and `CL_OUT` type parameters returns results to the caller, they must be passed by reference.)
- `CL_IN`, by reference without count
- `CL_IN`, by reference with count
- `CL_INOUT`, by reference without count
- `CL_INOUT`, by reference with count
- `CL_OUT`, by reference without count
- `CL_OUT`, by reference with count

2.2.2 IDL Synchronous Client Stub

The IDL synchronous client stub is generated from the component definition provided in the XML file. A stub is generated for every function installed in the client component. The first argument of the stub is always the handle that contains RMD parameters and the IDL handle needs to be preinitialized.

the signature of the IDL synchronous client stub is as specified in the XML file with the exception that a rearrangement of parameters in `CL_IN`, `CL_INOUT` and `CL_OUT` order is performed.

Guidelines for designing synchronous client stub are:

- `in` parameters must not be deleted. These parameters are marshalled into a message. The prototype for marshalling these parameters is:
  ```
  clXdrMarshall<Struct>(param, inmsg, 0);
  ```
- `inout` parameters must be destroyed and prepared to return values. The prototype for marshalling these parameters is:
  ```
  clXdrMarshall<Struct>(param, inmsg, 1);
  ```
- After the call, the `inout/out` parameters are populated with the `outmsg`. The prototype for unmarshalling these parameters is:
  ```
  clXdrUnmarshall<Struct>(param, outmsg);
  ```
- `out` parameters can be assumed to be devoid of heap pointers. `inout` parameters are made devoid of heap pointers during the marshaling call.
- Parameters by reference are passed to marshalling functions as is.
- Parameters by value are passed by reference.

Error handling

It is the caller's responsibility to manage the variables. Variables are deleted when as error is detected and the call returns immediately.

Handling Name Address

RMD recognizes numeric addresses. Hence, it is the responsibility of IDL stub to convert the name address to a numeric address. This is performed at the beginning of the stub when it detects that the address passed is a name address.

Algorithm

1. Resolve address, if it is of type name.
2. Create input message, if `in/inout` parameters are present.
3. Marshall `in` parameters into input message.
4. Marshall `inout` parameters with destruction flag into input message.
5. Create output message, if `out/inout` parameters are present.
6. Call RMD synchronously.
7. Unmarshall `inout` parameters from output message.
8. Unmarshall `out` parameters from output message.
9. Return.

For a function `<func>`, client stub `<func>SyncClient` is generated. The client header will contains:

```
#define <func> <func>SyncClient
```

The application can now make a call to `<func>` and if the application includes the client header, it is replaced with `<func>SyncClient`.

2.2.3 IDL Asynchronous Client Stub

The asynchronous stub is similar to the sync stub with the following exceptions:

- There are two additional arguments at the end of the argument list:
  - `void *cookie` - Pointer provided by the application and returned to it in the callback
  - `<component><func>AsyncCallbackT func` - Pointer to the callback function that must be invoked when the server returns.
- The RMD call is an asynchronous RMD call.
- The output message does not contain the `out/inout` arguments after the RMD call.
- The stub allocates a cookie of it's own from the heap, stores the application's cookie and the callback, and passes a reference to this cookie (stub cookie) to the RMD call.
- It is named `<func>AsyncClient` and has to be invoked likewise with the two additional parameters.

Algorithm

1. Resolve address, if it is of type name.

2. Create input message, if `in/inout` parameters are present.
3. Marshall `in` parameters into input message.
4. Marshall `inout` parameters with destruction variable into input message.
5. Create output message, if `out/inout` parameters are present.
6. Allocate IDL cookie and store user's cookie and callback into the IDL cookie.
7. Call the RMD function asynchronously with IDL cookie and IDL callback as asynchronous options.

2.2.3.1 IDL Asynchronous Callback

This is the function that is called when an asynchronous call is made and the context from the server returns. It is similar to synchronous client stub that is present after the synchronous RMD call returns. This function is invoked from the context of RMD. It's signature is:

```
void <func>AsyncCallback(ClRcT rc, void *pIdlCookie,
ClBufferMessageHandleT inMsgHdl, ClBufferMessageHandleT outMsgHdl)
```

Where:

- `rc` - Return value of the server function.
- `pIdlCookie` - Cookie passed by the asynchronous IDL client call.
- `inMsgHdl` - Message containing the input, `inout` parameters (XDR'ed).
- `outMsgHdl` - Message containing the `inout`, output parameters (XDR'ed).

Error Handling

It is the responsibility of the callback to destroy the variables it creates. When an error is detected, the function returns only after destroying the variables it created as part of the unmarshalling operation. The call to destroy variables is:

```
clXdrMarshall<Struct>(param, 0, 1);
```

The variable is destroyed by creating a system of `goto` labels. When a failure is detected at any point, the `goto` takes the code execution to an alternate point in the function from where all variables that were created are destroyed. For example:

```
rc = createVar v1; if (error) goto L0;

rc = createVar v2; if (error) goto L1;

--

return; /*end of normal flow*/

--

/*error flow*/

L1:  delete v2;

L0:  delete v1;

return;
```

The callback function obtains the following and invokes the application's callback.

- Cookie provided by the application
- Callback from the IDL cookie

It unmarshalls the input parameters from the input message and the `inout` and output parameters from the output message before calling the application's callback.

Algorithm

1. Set all local variables to be passed to the application's callback to 0.
2. Unmarshall `in` parameters from input message.
3. Unmarshall `inout` parameters from output message.
4. Unmarshall `out` parameters from output message.
5. Return.

2.2.4 IDL Server-side Code

2.2.4.1 Server Stubs

The IDL server stub is invoked by RMD server library in response to a request from the client. This stub is installed in the client component's tables. This stub performs a reverse operation compared to that of the sync client stub. It extracts the `in` and `inout` parameters from the input message and calls the actual server function with these parameters. When the function returns, it packs the `inout` and `out` parameters in the output message and destroys the variables it created.

Error handling

This is complex than the callback stub. Error handling can be viewed from the following perspectives:

- Errors can occur when variables are being unmarshalled. All variables created have to be destroyed. This is achieved through the `goto` system.
- Errors can occur after the server function is called while unmarshalling the variables. Unmarshalled/undeleted variables have to be destroyed. This can be achieved through stack unrolling simulated by multiple labels and the `goto` system.

Algorithm

1. `memset` all local variables to be passed to the server function to 0.
2. Unmarshall input and `inout` variables from the input message.
3. Call the server function.
4. Marshall `inout` variables into output message.
5. Marshall `out` variables into output message.
6. Destroy the `in` variables.
7. Return.

2.2.4.2 New Functions

`ClRcT cl<service>IdlSyncDefer(ClIdlHandleT *phDefer);`

This call is used to defer the response from a function call. For example, a client makes a call to server A, to complete that request the server A needs to make calls to servers, B and C. These inter-server calls cannot be made synchronous as they would block the server. Hence, these calls need to be asynchronous, but we do not know when the response will be received. Server A cannot return to the client if synchronous call is made from the client or invoke the callback if an asynchronous call is made from the client.

So, server A needs to defer the response it sends to the client. It can do this by making as a call to the function, `cl<service>IdlSyncDefer(ClIdlHandleT *phDefer)`. This function populates the defer handle that is used later to send a response to the client. The response to the client is deferred and should be sent explicitly through the `clIdlSyncResponseSend()` function.

`ClRcT clIdlSyncResponseSend();`

When the response to a function is deferred, it should be sent explicitly through this function. This function is generated for every server function. The first argument of this function is the `IdlHandle` returned from a call to `SyncDefer`, the second argument is the return code of the server function invoked by the client. Following this are the `INOUT` and `OUT` parameters of the server function for which this `clIdlSyncResponseSend()` function is generated.

Where the client receives this call depends on the method the client used to invoke the server function. If the call was synchronous, the client is unblocked only when the server executes the `clIdlSyncResponseSend()` function. If the call was asynchronous with callback, the callback is invoked only when the ResponseSend is executed.

2.2.5 Clean-up

For Asynchronous Methods

- All the `CL_IN` parameters that are dynamically allocated can be deleted after a call to the asynchronous method. This cannot be performed during the callback as the argument passed here is dummy (not allocated from the heap). The caller must use the free function, that is equivalent to the allocation function used for allocating this memory. For example, if `clHeapAllocate`, `clHeapCalloc`, or `clHeapRealloc` was used to allocate these variables, `clHeapFree` must be used. If `malloc/realloc/calloc` was used to allocate, `free()` must be used.
- All the `CL_INOUT` parameters that are dynamically allocated must be deleted only in the callback function. For freeing these variables, only `clHeapFree()` must be irrespective of what was used to allocate those variables.
- All the `CL_OUT` parameters that are dynamically allocated must be deleted only in the callback function. For freeing these variables, only `clHeapFree()` must be used irrespective of what was used to allocate those variables.

For Synchronous Methods

All the dynamically allocated variables can be freed after the call and the function that performs this are `clHeapFree` and `free()`.

Chapter 3

Service APIs

3.1 Type Definitions

3.1.1 ClIdlHandleT

typedef ClHandleT ClIdlHandleT;

The type of handle supplied by IDL to a process or component when it is initialized. This handle is used in successive calls to IDL library functions so that the process can be recognized.

3.1.2 ClIdlHandleObjT

```
typedef struct {
        ClIdlAddressT address;
        ClUint32T flags;
        ClRmdOptionsT options;
} ClIdlHandleObjT;
```

The structure, `ClIdlHandleObjT`, represents the IDL handle object. This object contains the details passed by the component during `clIdlHandleInitialize()`. This structure is used in all IDL functions. The attributes of this structure are:

- *address* - Specifies if the address is of type name or IOC. It is the destination address of the server side stub.
- *flags* - Flags that can be passed to IDL.
- *options* - RMD options such as `retries`, `timeout`, and `priority`.

3.1.3 ClIdlAddressT

```
typedef struct {
        ClUint32T addressType;
        ClIdlAddressTypeT address;
} ClIdlAddressT;
```

The structure, `ClIdlAddressT`, contains the address passed to IDL. This is the address where the server can be found. The structure has two attributes:

- *addressType* - Type of the address that can be one of the following:
  - CL_IDL_ADDRESSTYPE_IOC: IOC address.
  - CL_IDL_ADDRESSTYPE_NAME: Name service address.
- *address* - This is the placeholder for the address that contains the actual value of the address.

3.1.4 ClIdlAddressTypeT

```
typedef union ClIdlAddressTypeT {
        ClIocAddressT iocAddress;
        ClIdlNameAddressT nameAddress;
} ClIdlAddressTypeT;
```

This is the union that contains the IDL address. IDL accepts either IOC address or a name service address. The address currently in use, if defined by the addressType variable, is contained in the `ClIdlAddressT` structure. This union has two attributes:

- *iocAddress* - A union that contains the physical, logical, or multicast address. Refer to the Service Description and API Reference for IOC Service document for details about `ClIocAddressT`.
- *nameAddress* - The name service address.

3.1.5 ClIdlNameAddressT

```
typedef struct ClIdlNameAddressT {
        ClUint32T contextCookie;
        ClNameT name;
        ClUint32T attrCount;
        ClNameSvcAttrEntryT attr[CL_NS_MAX_NO_ATTR];
} ClIdlNameAddressT;
```

This structure, `ClIdlNameAddressT`, contains the name service address. This address needs to be resolved through the name service. This structure contains four attributes:

- *contextCookie* - This is the context cookie that Name Service expects to be passed.
- *name* - The name of the service requested.
- *attrCount* - The number of attributes being passed
- *attr* - The attributes being passed.

3.2 Library Life Cycle APIs

3.2.1 clIdlHandleInitialize

clIdlHandleInitialize

Synopsis:
Initializes the IDL handle with the RMD parameters and the server destination.

Header File:
clIdlApi.h

Syntax:

```
ClRcT clIdlHandleInitialize(
                         CL_IN  ClIdlHandlcomponentbj *pIdlObj,
                         CL_OUT ClIdlHandleT   *pHandle);
```

Parameters:

pIdlObj: (in) Object of the IDL handle. This contains information required to communicate with the destination component.

pHandle: (out) Handle of the IDL object used in subsequent calls to IDL.

Return values:

CL_OK: IDL initialized successfully.

CL_IDL_RC(CL_ERR_NULL_POINTER): `pIdlObj` or `pHandle` contains a NULL pointer.

CL_IDL_RC(CL_ERR_NO_MEMORY): IDL or a module of IDL is out of memory. Thus, the service cannot be provided at this time. This can be a transient problem.

Description:
This function is used to initialize the IDL handle with various RMD parameters. After it is initialized, a handle is returned that is used for all subsequent calls to IDL API functions. RMD options such as server address and flags are set to the values passed in this function.

Library File:
libClIdl.a

Related Function(s):
clIdlHandleFinalize, clIdlHandleUpdate.

3.2.2 clIdlHandleFinalize

clIdlHandleFinalize

Synopsis:
Removes the IDL handle and frees all the resources allocated during initialization of IDL handle.

Header File:
clIdlApi.h

Syntax:

```
ClRcT clIdlHandleFinalize(
                  CL_IN ClIdlHandleT handle );
```

Parameters:
handle: (in) Handle of the IDL object. This handle is obtained from a previous call to the `clIdlInitialize()` function.

Return values:
CL_OK: The function executed successfully. The handle is destroyed and cannot be used.

Description:
This function is used to destroy the IDL handle. This deletes the handle created by the library and frees the resources allocated to it during initialization.

Library File:
libClIdl.a

Related Function(s):
clIdlHandleInitialize.

3.3 Functional APIs

3.3.1 clIdlHandleUpdate

clIdlHandleUpdate

Synopsis:
Updates the IDL handle. This function can be used to change the address of the server stub, RMD options, or RMD flags.

Header File:
clIdlApi.h

Syntax:

```
        ClRcT clIdlHandleUpdate(
                             CL_IN ClIdlHandleT    handle,
                                CL_IN ClIdlHandlcomponentbj *pIdlObj);
```

Parameters:
pHandle: (in) Handle of the IDL object obtained from a previous call to the `clIdlInitialize()` function.

pIdlObj: (in) Object of the IDL handle used in subsequent calls to IDL.

Return values:
CL_OK: The function executed successfully and the handle is updated with the new IDL object.

CL_IDL_RC(CL_ERR_NULL_POINTER): `pIdlObj` contains a NULL pointer.

CL_IDL_RC(CL_ERR_NO_MEMORY): IDL or a module of IDL is out of memory. Thus, the service cannot be provided at this time. This can be a transient problem.

Description:
This function is used to update the IDL handle with various IDL parameters.

Library File:
libClIdl.a

Related Function(s):
clIdlHandleInitialize, clIdlHandleFinalize.

3.3.2 clIdlVersionCheck

clIdlVersionCheck

Synopsis:
Verifies if the current version of IDL service is the version expected by the client.

Header File:
clIdlApi.h

Syntax:
```
ClRcT clIdlVersionCheck(
                            CL_INOUT     ClVersionT  *pVersion);
```

Parameters:
pVersion: (in/out) As an input parameter, this acts as a pointer to the required IDL Service version. As an output parameter, it contains the version of IDL service supported by current implementation of OpenClovis ASP.

Return values:
CL_OK: The function executed successfully.

CL_IDL_RC(CL_ERR_NULL_POINTER): `pVersion` contains a NULL pointer.

Description:
This function is used to check the version of the IDL library. If it is invalid, the function returns the supported version.

Library File:
libClIdl.a

Related Function(s):
None.

Chapter 4

Service Management Information Model

TBD

Chapter 5

Service Notifications

TBD

Chapter 6

Configuration

TBD

Chapter 7

Debug CLI

TBD

Index