TR-069

CPE WAN Management Protocol

How to Add the TR-069 Parameters

And

How to Set the TR-069 Web page

CONFIDENTIAL

Contents

Version History

Version	Date	Editor	Changes
1.0	2007/08/02	Jiunming Chen	The initial version

1 Introduction

This document will show you how to add the TR-069 parameters and how to set the TR-069 Web page. This includes several sessions to describe these topics. First, we will show you what are the TR-069 parameters, and will describe our implementation architecture of the TR-069 parameters in our system. Then, define the C structures and functions used in our implementation. We will use the example to explain how to add a new parameter, how to add a new instance of a multi-instance object, and how to apply the new values. Finally, we will explain how to configure the TR-069 Web page.

2 TR-069 parameters

The TR-069 Parameters are defined in the TR-069 specification by DSL Forum. "Parameter" means a name-value pair representing a manageable CPE parameter made accessible to an ACS for reading and/or writing. The name identifies the particular Parameter, and has a hierarchical structure similar to files in a directory, with each level separated by a "." (dot). The value of a Parameter may be one of data types, string, int, unsigned int, boolean, dateTime, and base64. Besides, there is a special Parameter type, object, which is a container for parameters and/or other objects.

Parameter may be defined as read-only or read-write. Read-only Parameters may be used to allow an ACS to determine specific CPE characteristics, observe the current state of the CPE, or collect statistics. Writeable Parameters allow an ACS to customize various aspects of the CPE's operation. All writeable Parameters must also be readable although those that contain confidential user information, e.g. passwords, may return empty values when read.

Because there are many parameters defined by DSL Forum, we use some parameters listed below for our example. The detail definitions for each parameter can refer to TR-069 AnnexB.

InternetGatewayDevice.

InternetGatewayDevice.LANDeviceNumberOfEntries

InternetGatewayDevice.WANDeviceNumberOfEntries

InternetGatewayDevice.DeviceInfo.

InternetGatewayDevice.DeviceInfo.Manufacturer

InternetGatewayDevice.DeviceInfo.ManufacturerOUI

InternetGatewayDevice.DeviceInfo.ModelName

InternetGatewayDevice.DeviceInfo.Description

InternetGatewayDevice.DeviceInfo.SerialNumber

InternetGatewayDevice.DeviceInfo.HardwareVersion

InternetGatewayDevice.DeviceInfo.SoftwareVersion

InternetGatewayDevice.DeviceInfo.SpecVersion

InternetGatewayDevice.DeviceInfo.ProvisioningCode

InternetGatewayDevice.DeviceInfo.UpTime

InternetGatewayDevice.DeviceInfo.DeviceLog

InternetGatewayDevice.Layer3Forwarding.

InternetGatewayDevice.Layer3Forwarding.DefaultConnectionService

InternetGatewayDevice.Layer3Forwarding.ForwardNumberOfEntries

InternetGatewayDevice.Layer3Forwarding.Forwarding.

InternetGatewayDevice.Layer3Forwarding.Forwarding.{i}.

InternetGatewayDevice.Layer3Forwarding.Forwarding.{i}.Enable

InternetGatewayDevice.Layer3Forwarding.Forwarding.{i}.Status

InternetGatewayDevice.Layer3Forwarding.Forwarding.{i}.Type

InternetGatewayDevice.Layer3Forwarding.Forwarding.{i}.DestIPAddress

InternetGatewayDevice.Layer3Forwarding.Forwarding.{i}.DestSubnetMask

InternetGatewayDevice.Layer3Forwarding.Forwarding.{i}.SourceIPAddress

InternetGatewayDevice.Layer3Forwarding.Forwarding.{i}.SourceSubnetMask

InternetGatewayDevice.Layer3Forwarding.Forwarding.{i}.GatewayIPAddress

InternetGatewayDevice.Layer3Forwarding.Forwarding.{i}.Interface

InternetGatewayDevice.Layer3Forwarding.Forwarding.{i}.ForwardingMetric

3 Implementation Architecture

The figure below shows the implementation architecture of TR-069 parameters defined above. Because parameters are like a directory structure, the implementation architecture also uses a directory-like structure and the root node is "*InternetGatewayDevice.*".

There are 3 main data structures defined in this architecture, including "**struct CWMP_LEAF**", "**struct CWMP_NODE**", and "**struct CWMP_LINKNODE**". "**struct CWMP_NODE**" and "**struct CWMP_LINKNODE**" present those parameters whose data type is object, and "**struct CWMP_LEAF**" presents those parameters whose data types are not object . The difference of "**struct CWMP_NODE**" and "**struct CWMP_LINKNODE**" is that "**struct CWMP_NODE**" means a static object, but "**struct CWMP_LINKNODE**" means a dynamic object that we can have operations on (ex. add, delete, and etc). In other words, these parameters of "**struct CWMP_NODE**" and "**struct CWMP_LINKNODE**" are like directories, and these parameters of "**struct CWMP_LEAF**" are like files in directories. The detail data structures will be described in session 4.

For example, the root node, "*InternetGatewayDevice.*", belongs to "**struct CWMP_NODE**" (a static object). Its *leaf* points to a "**struct CWMP_LEAF**" array including 2 leaves, "*LANDeviceNumberOfEntries*" and "*WANDeviceNumberOfEntries*". And its *next*

points to a "**struct CWMP_NODE**" array including 2 nodes, "*DeviceInfo*" and "*Layer3Forwarding*". Hence these data structures present these parameters below:

InternetGatewayDevice.

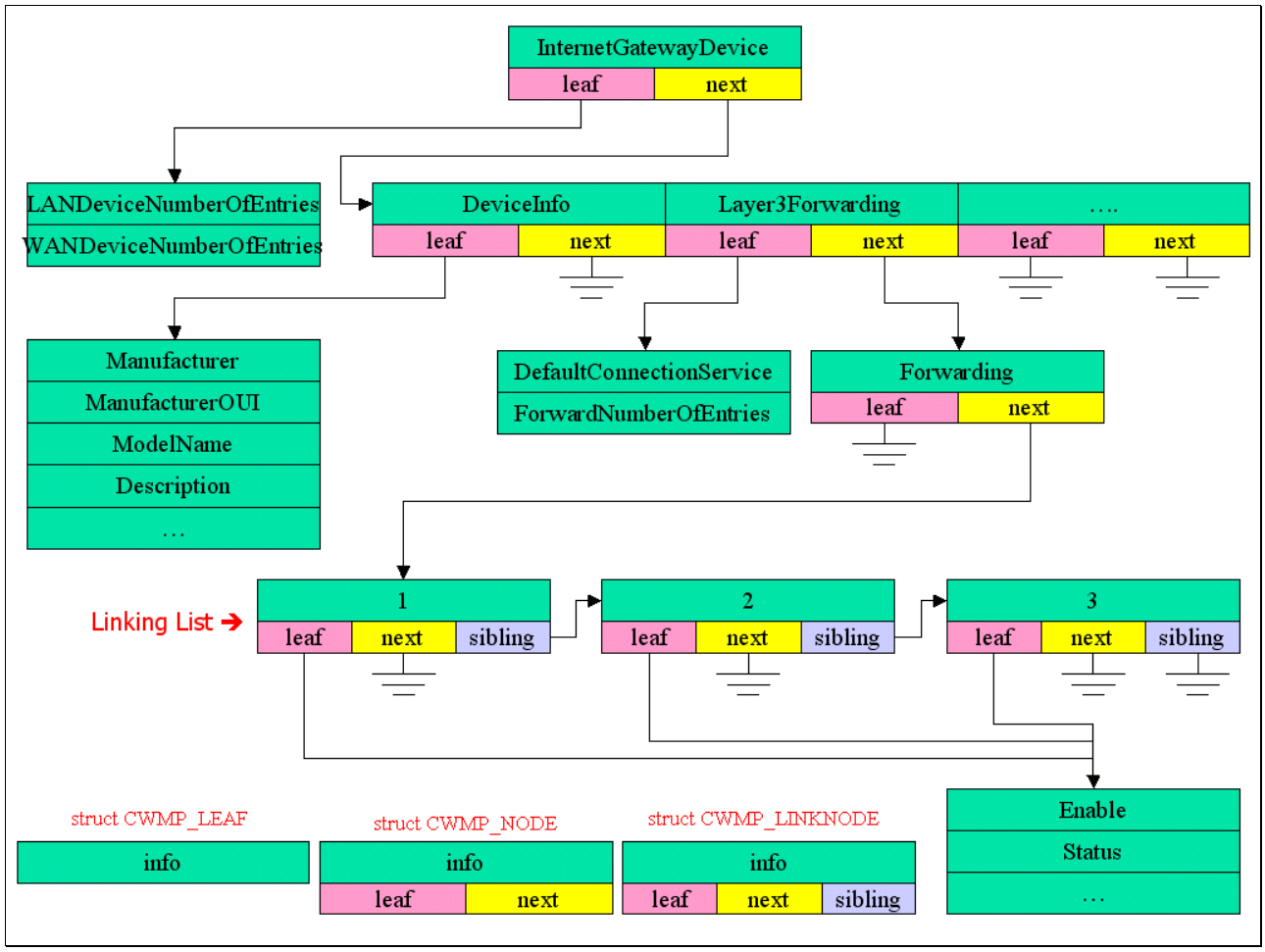
InternetGatewayDevice.LANDeviceNumberOfEntries

InternetGatewayDevice.WANDeviceNumberOfEntries

InternetGatewayDevice.DeviceInfo.

InternetGatewayDevice.Layer3Forwarding.

Giving another example, "*InternetGatewayDevice.Layer3Forwarding.*" is a "**struct CWMP_NODE**". Its *leaf* points to "*DefaultConnectionServie*" and "*ForwardingNumberOfEntries*", and its *next* points to "*Forwarding.*" node. "*Forwarding.*" is defined read-write object that means we can use AddObject RPC Method to create a new Forwarding instance, use DeleteObject RPC Method to delete a specific Forwarding instance, and synchronize these Forwarding instances for internal management. Because these operations for "*Forwarding.*" node, the *next* of the "*Forwarding.*" structure uses a "**struct CWMP_LINKNODE**" linking list to link all the Forwarding instances instead of using a static/fixed-size array to present. "**struct CWMP_LINKNODE**" has one extra field called *sibling* and uses the *sibling* to link different forwarding instances into a linking list.



4 Structure and Function Definitions

These structures and functions described below are defined in the parameter_api.h

4.1 struct CWMP_OP

4.1.1 Definition

```
struct CWMP_OP
{
    int   (*getvalue)(char *name, struct CWMP_LEAF *entity, int *type, void **data);
    int   (*setvalue)(char *name, struct CWMP_LEAF *entity, int type, void *data);
};
```

4.1.2 Description

Define the get-callback function and the set-callback function to get and set the value of a specific parameter.

4.1.3 Arguments

getvalue: the get-callback function
setvalue: the set-callback function

4.2 struct CWMP_PRMT

4.2.1 Definition

```
struct CWMP_PRMT
{
    char              *name;
    unsigned short    type;
    unsigned short    flag;
    struct CWMP_OP    *op;
};
```

4.2.2 Description

The structure is used to contain all the information about a specific parameter including its name, data type, special flag (attributes), and callback functions for get and set operations.

4.2.3 Arguments

name: the name of this parameter
type: the data type of this parameter, and the data type is defined in session 4.6

flag: the flag (attributes) of this parameter, and the flag is defined in session 4.7

op: the get and set operation for this parameter, and the structure is defined in session 4.1.

4.3 struct CWMP_LEAF

4.3.1 Definition

```
struct CWMP_LEAF
{
    struct CWMP_PRMT    *info;
};
```

4.3.2 Description

The structure is used to contain information about a specific parameter whose data type is not object.

4.3.3 Arguments

info: include the information about a parameter whose data type is not object, and the data structure is defined in session 4.2.

4.4 struct CWMP_NODE

4.4.1 Definition

```
struct CWMP_NODE
{
    struct CWMP_PRMT    *info;
    struct CWMP_LEAF    *leaf;
    struct CWMP_NODE    *next;
};
```

4.4.2 Description

The structure is used to contain the information about a specific parameter that is a static object.

4.4.3 Arguments

info: include the information about a parameter whose data type is static object, and the data structure is defined in session 4.2.

leaf: point to a leaf array that is the next level of this object.

next: point to a object array/linking list that is the next level of this object.

4.5 struct CWMP_LINKNODE

4.5.1 Definition

```
struct CWMP_LINKNODE
{
    struct CWMP_PRMT        *info;
    struct CWMP_LEAF        *leaf;
    struct CWMP_NODE        *next;
    struct CWMP_LINKNODE    *sibling;
    unsigned int            instnum;
};
```

4.5.2 Description

The structure is used to contain the information about a specific parameter that is a dynamic object.

4.5.3 Arguments

info: include the information about a parameter whose data type is dynamic object, and the data structure is defined in session 4.2.

leaf: point to a leaf array that is the next level of this object.

next: point to a object array/linking list that is the next level of this object.

sibling: point to the next sibling node in this linking list.

instnum: the instance number for this instance of the multi-instance object

4.6 eCWMP_TYPE

4.6.1 Definition

```
typedef enum
{
    eCWMP_tNONE         =0,
    eCWMP_tSTRING       = SOAP_TYPE_string,
    eCWMP_tINT          = SOAP_TYPE_int,
    eCWMP_tUINT         = SOAP_TYPE_unsignedInt,
    eCWMP_tBOOLEAN      = SOAP_TYPE_xsd__boolean,
    eCWMP_tDATETIME     = SOAP_TYPE_time,
//  eCWMP_tBASE64       = SOAP_TYPE_xsd__base64,
    eCWMP_tFILE         = 200,
    eCWMP_tOBJECT,
    eCWMP_tINITOBJ,
    eCWMP_tADDOBJ,
    eCWMP_tDELOBJ,
```

eCWMP_tUPDATEOBJ
} eCWMP_TYPE;

4.6.2 Description

This enumeration shows the data types used by TR-069, and actions used internally by cwmpClient program.

4.6.3 Arguments

eCWMP_tNONE:	none
eCWMP_tSTRING:	data type of string
eCWMP_tINT:	data type of integer
eCWMP_tUINT:	data type of unsigned integer
eCWMP_tBOOLEAN:	data type of boolean
eCWMP_tDATETIME:	data type of dateTime
eCWMP_tBASE64:	data type of base64 (do not support this now)
eCWMP_tFILE:	data type of string. This string data is put in a file for internal use.
eCWMP_tOBJECT:	data type of object
eCWMP_tINITOBJ:	action to initialize the instances of a dynamic object
eCWMP_tADDOBJ:	action to add a new instance of a dynamic object
eCWMP_tDELOBJ:	action to delete an instance of a dynamic object
eCWMP_tUPDATEOBJ:	action to synchronize the instances of a dynamic object with the latest configuration.

4.7 Flag definition

4.7.1 Definition

```
#define   CWMP_WRITE      0x01
#define   CWMP_READ       0x02
#define   CWMP_LNKLIST    0x04
#define   CWMP_DENY_ACT   0x08
```

4.7.2 Description

These define the flags used by struct CWMP_PRMT defined in session 4.2.

4.7.3 Arguments

CWMP_WRITE:	this parameter is writable
CWMP_READ:	this parameter is readable
CWMP_LNKLIST:	this parameter is an instance of a dynamic object, and can be deleted dynamically.

CWMP_DENY_ACT: this parameter cannot be set to active notification by SetParameterAttributes RPC method call.

4.8 Error-code definition

4.8.1 Definition

```
#define ERR_9000  -9000    /*Method not supported*/
#define ERR_9001  -9001    /*Request denied*/
#define ERR_9002  -9002    /*Internal error*/
#define ERR_9003  -9003    /*Invalid arguments*/
#define ERR_9004  -9004    /*Resources exceeded*/
#define ERR_9005  -9005    /*Invalid parameter name*/
#define ERR_9006  -9006    /*Invalid parameter type*/
#define ERR_9007  -9007    /*Invalid parameter value*/
#define ERR_9008  -9008    /*Attempt to set a non-writable parameter*/
#define ERR_9009  -9009    /*Notification request rejected*/
#define ERR_9010  -9010    /*Download failure*/
#define ERR_9011  -9011    /*Upload failure*/
#define ERR_9012  -9012    /*File transfer server authentication failure*/
#define ERR_9013  -9013    /*Unsupported protocol for file transfer*/
```

4.8.2 Description

These define the CPE fault codes used by TR-069

4.8.3 Arguments

ERR_9000: Method not supported

ERR_9001: Request denied (no reason specified)

ERR_9002: Internal error

ERR_9003: Invalid arguments

ERR_9004: Resources exceeded (when used in association with SetParameterValues, this MUST NOT be used to indicate parameters in error)

ERR_9005: Invalid parameter name (associated with Set/GetParameterValues, GetParameterNames, Set/GetParameterAttributes, AddObject, and DeleteObject)

ERR_9006: Invalid parameter type (associated with SetParameterValues)

ERR_9007: Invalid parameter value (associated with SetParameterValues)

ERR_9008: Attempt to set a non-writable parameter (associated with SetParameterValues)

ERR_9009: Notification request rejected (associated with SetParameterAttributes)
ERR_9010: Download failure (associated with Download or TransferComplete)
ERR_9011: Upload failure (associated with Upload or TransferComplete)
ERR_9012: File transfer server authentication failure (associated with Upload, Download, or TransferComplete)
ERR_9013: Unsupported protocol for file transfer (associated with Upload and Download)

4.9 init_ParameterTable()

4.9.1 Definition

*int init_ParameterTable(struct CWMP_NODE **root, struct CWMP_NODE table[], char *prefix);*

4.9.2 Description

This function is used to initialize a parameter tree. We use a parameter tree to record what parameters we are supported.

4.9.3 Arguments

root: the root node
table: the next node that needs to be initialized.
prefix: the parameter path name.

4.9.4 Return Value

0: successful.
-1: error

4.10 destroy_ParameterTable()

4.10.1 Definition

*int destroy_ParameterTable(struct CWMP_NODE *table);*

4.10.2 Description

This function is used to destroy a parameter tree that is pointed by *table* variable

4.10.3 Arguments

table: the node that will be destroyed.

4.10.4 Return Value

0: successful.
-1: error

4.11 create_Object()

4.11.1 Definition

*int create_Object(struct CWMP_LINKNODE **table,*
struct CWMP_LINKNODE ori_table[], int size, unsigned int num, unsigned int from);

4.11.2 Description

This function is only called when initializing the instances of a dynamic object to create a linking list that consists of the current instances

4.11.3 Arguments

table: the root node of the linking list
ori_table: the instance definition table
size: the size of ori_table
num: the number of how many instances that need to initialize together.
from: the starting number of the instance number for this object

4.11.4 Return Value

0: successful.
-1: error

4.12 add_Object()

4.12.1 Definition

*int add_Object(char *name, struct CWMP_LINKNODE **table,*
*struct CWMP_LINKNODE ori_table[], int size, unsigned int *num);*

4.12.2 Description

This function is used to add a new instance in the parameter tree, and is used when handling **eCWMP_tADDOBJ** and **eCWMP_tUPDATEOBJ** actions to manage the instances of a dynamic object.

4.12.3 Arguments

name: the parameter path name
table: the root node of the linking list
ori_table: the instance definition table
size: the size of ori_table
num: the instance number for this new instance. If 0, cwmpClient gives an new instance number.

4.12.4 Return Value

0: successful.
-1: error

4.13 del_Object()

4.13.1 Definition

*int del_Object(char *name, struct CWMP_LINKNODE **table, unsigned int num);*

4.13.2 Description

This function is used to delete an existed instance in the parameter tree, and is used when handling **eCWMP_tDELOBJ** action to manage the instances of a dynamic object.

4.13.3 Arguments

name: the parameter name path
table: the root node of the linking list
num: the instance number that will be deleted.

4.13.4 Return Value

0: successful.
-1: error
ERR_9005: invalid parameter name.

4.14 add_SiblingEntity()

4.14.1 Definition

*int add_SiblingEntity(struct CWMP_LINKNODE **table,*
*struct CWMP_LINKNODE *new_entity);*

4.14.2 Description

This function is used to add an instance into the linking list, and is used when handling **eCWMP_tUPDATEOBJ** action to manage the instances of a dynamic object.

4.14.3 Arguments

table: the root node of the linking list
new_entity: the new instance that will be added

4.14.4 Return Value

0: successful.
-1: error

4.15 remove_SiblingEntity()

4.15.1 Definition

*struct CWMP_LINKNODE *remove_SiblingEntity(struct CWMP_LINKNODE **table, unsigned int num);*

4.15.2 Description

This function is used to remove an existed instance with the specific instance number from the linking list, and is used when handling **eCWMP_tUPDATEOBJ** action to manage the instances of a dynamic object.

4.15.3 Arguments

table: the root node of the linking list

num: the instance number that a instance will be removed with

4.15.4 Return Value

NULL: do not find the specific instance.

Non-NULL: a pointer to the removed data, *struct CWMP_LINKNODE*.

4.16 find_SiblingEntity()

4.16.1 Definition

*struct CWMP_LINKNODE *find_SiblingEntity(struct CWMP_LINKNODE **table, unsigned int num);*

4.16.2 Description

This function is used to find if there is an instance in the linking list with the specific instance number, and is used when handling **eCWMP_tUPDATEOBJ** action to manage the instances of a dynamic object.

4.16.3 Arguments

table: the root node of the linking list

num: the instance number that a instance will be searched for

4.16.4 Return Value

NULL: do not find the specific instance.

Non-NULL: a pointer to the found data, *struct CWMP_LINKNODE*.

5 Add a New Parameter

This session will describe how to add a new parameter in our implementation architecture. Adding a new parameter includes 3 steps. First, define the necessary data structures or tables to contain information about parameters. The second step is to write the get-callback function. The last is to write the set-callback function if need.

The first step of adding new parameters is to define the necessary data structures. For example, the table below shows how to define the root node, "*InternetGatewayDevice*.".

```
struct CWMP_PRMT tROOTObjectInfo[] =
{
/*(name,                    type,              flag,          op)*/
{"InternetGatewayDevice",   eCWMP_tOBJECT,     CWMP_READ,     NULL}
};
enum eROOTObject
{
    eInternetGatewayDevice
};
struct CWMP_NODE tROOT[] =
{
/*info,                                        leaf,          next*/
{&tROOTObjectInfo[eInternetGatewayDevice],     tIGDLeaf,      tIGDObject },
{NULL,                                         NULL,          NULL}
};
```

There is no operation data structure defined for "*InternetGatewayDevice*.". **tROOTObjectInfo[]** table includes only one entry, that is "*InternetGatewayDevice*.". From this table, we can know that "*InternetGatewayDevice*." is object, read-only, and has no operation. **eROOTObject** is used to define an enumeration of parameters listed in the **tROOTObjectInfo** table. The elements in the enumeration must be the same order with those parameters in the **tROOTObjectInfo** table. **tRoot[]** is the root node/directory of all TR-069 parameters. It contains one meaning entry and ends with a NULL entry ({NULL, NULL, NULL}) (for internal use to count the table size). The first entry in the **tRoot**[] shows that it has a *leaf* array, **tIGDLeaf** and a *next* array, **tIGDObject**. In the next, we will show you the table definitions, **tIGDLeaf** and **tIGDObject**.

tIGDLeaf[] is defined below:

```
struct CWMP_OP tIGDLeafOP = { getIGD, NULL };
struct CWMP_PRMT tIGDLeafInfo[] =
{
/*(name,                        type,              flag,          op)*/
{"LANDeviceNumberOfEntries",    eCWMP_tUINT,       CWMP_READ,     &tIGDLeafOP},
{"WANDeviceNumberOfEntries",    eCWMP_tUINT,       CWMP_READ,     &tIGDLeafOP}
};
enum eIGDLeaf
{
    eLANDeviceNumberOfEntries,
    eWANDeviceNumberOfEntries
};
```

```
struct CWMP_LEAF tIGDLeaf[] =
{
{ &tIGDLeafInfo[eLANDeviceNumberOfEntries]    },
{ &tIGDLeafInfo[eWANDeviceNumberOfEntries]    },
{ NULL      }
};
```

tIGDLeafOP defines the set and get operations, and only the get-callback function, getIGD, is defined. **tIGDLeafInfo[]** defines the *leaf* parameters of its root node, "*InternetGatewayDevice.*", and it includes 2 entries, "*LANDeviceNumberOfEntries*" and "*WANDeviceNumberOfEntries*". From this table, we know that "*LANDeviceNumberOfEntries*" is unsigned integer, read-only, and has an operation named **tIGDLeafOP**, and "*WANDeviceNumberOfEntries*" has the same attributes with LANDeviceNumberOfEntries. **eIGDLeaf** is used to define an enumeration of parameters listed in the **tIGDLeafInfo** tables. **tIGDLeaf[]** is a "struct CWMP_LEAF" array, and is used to link with its root node, **tRoot[]**. It includes the parameters defined in the ***tIGDLeafInfo[]*** and ends with a NULL entry.

tIGDObject [] is defined below:

```
struct CWMP_PRMT tIGDObjectInfo[] =
{
/*(name,                    type,                flag,              op)*/
{"DeviceInfo",              eCWMP_tOBJECT,       CWMP_READ,         NULL},
{"Layer3Forwarding",        eCWMP_tOBJECT,       CWMP_READ,         NULL}
};
enum eIGDObject
{
        eDeviceInfo,
        eLayer3Forwarding
};
struct CWMP_NODE tIGDObject[] =
{
/*info,                                  leaf,                      next)*/
{&tIGDObjectInfo[eDeviceInfo],           tDeviceInfoLeaf,           tDeviceInfoObject},
{&tIGDObjectInfo[eLayer3Forwarding],     tLayer3ForwardingLeaf,     tLayer3ForwardingObject},
{NULL,                                   NULL,                      NULL}
};
```

There is no operation defined for **tIGDObjectInfo[]** because all of the objects are static in this example. We will show you how to manage dynamic objects in session 6. **tIGDObjectInfo[]** defines the *object* parameters of its root node, "*InternetGatewayDevice.*", and it includes 2 entries, "*DeviceInfo.*" and "*Layer3Forwarding.*". From this table, we know that "*DeviceInfo.*" is object, read-only, and has no operation, and "*Layer3Forwarding.*" has the same attributes with "DeviceInfo". **eIGDObject** is used to define an enumeration of parameters listed in the **tIGDObjectInfo[]** table. **tIGDObject[]** is a "struct CWMP_NODE" array, and is used to link with its root node, **tRoot[]**. It includes the parameters defined in the **tIGDObjectInfo[]** and ends with a NULL entry.

After we define all of the necessary tables, the next step is to write the get-callback

function. The prototype of the get callback function is defined as:

*int getIGD(char *name, struct CWMP_LEAF *entity, int *type, void **data);*

The arguments of the get-callback function are:

name: the parameter name whose value will be returned.

entity: the information of the parameter that is encapsulated in struct CWMP_LEAF.

type: the output result to represent the data type of this parameter

data: the output result to represent the value of this parameter.

The return values of the get-callback function are:

0: no error

-1: error (caused by internal error)

ERR_9xxx: as described in session 4.8

In the above example, we have to write the callback function, *getIGD()*, to get the values of the parameters, "*InternetGatewayDevice.LANDeviceNumberOfEntries"* and "*InternetGatewayDevice.WANDeviceNumberOfEntries"*. The C codes of *getIGD()* are shown below:

```
int getIGD(char *name, struct CWMP_LEAF *entity, int *type, void **data)
{
	char  *lastname = entity->info->name;

	if( (name==NULL) || (type==NULL) || (data==NULL) || (entity==NULL))
		return -1;

	*type = entity->info->type;
	*data = NULL;
	if( strcmp( lastname, "LANDeviceNumberOfEntries" )==0 )
	{
		*data = uintdup( LANDEVICE_NUM );
	}else if( strcmp( lastname, "WANDeviceNumberOfEntries" )==0 )
	{
		*data = uintdup( WANDEVICE_NUM );
	}else{
		return ERR_9005;
	}

	return 0;
}
```

In this function, we use the *entity->info->name* to get the last-level name of the parameter. Because the parameters with the same prefix parameter path name may share the same callback function, we use *strcmp* function to compare the parameter name that we need to handle now. If the parameter can be handled in this callback function, get its value, and return 0. If there is any problem when handling a parameter, you need to return the fault code defined in session 4.8. The argument, *type*, is used to return the data type of this parameter, and the argument, *data*, is used to return the value of this parameter. Because the values of different parameters may be different data types (string, integer, unsigned integer, boolean,

base64, or dateTime), the declare type of the *data* variable is "void**", and you have to allocate the appropriate memory space to store the value. There are some utilities to duplicate values of different data types, like *strdup()* for string duplication, *intdup()* for integer, *uintdup()* for unsigned integer, *booldup()* for boolean, and *timedup()* for DateTime. After the get-callback function is executed, the cwmpClient will automatically free the memory space allocated by you.

Finally you need to write the set-callback function for those writable parameters. The prototype of the set-callback function is defined as:

*int setDeviceInfo(char *name, struct CWMP_LEAF *entity, int type, void *data);*

The arguments of the set-callback function are:

name: the parameter name whose value will be set.

entity: the information of the parameter that is encapsulated in struct CWMP_LEAF.

type: the data type of the *data* argument.

data: the new value to be set. You can know its data type by the *type* argument.

The return values of the set-callback function are:

1: no error (the new value has NOT been applied)

0: no error (the new value has been applied)

-1: error (caused by internal error)

ERR_9xxx: as described in session 4.8

Give an example to explain how to use the set-callback function. This example is to set the parameter, "*InternetGatewayDevice.DeviceInfo.ProvisioningCode*", and the C codes are shown below:

```
int setDeviceInfo(char *name, struct CWMP_LEAF *entity, int type, void *data)
{
	char	*lastname = entity->info->name;
	char	*buf=data;
	int	len=0;

	if( (name==NULL) || (data==NULL) || (entity==NULL)) return -1;
	if( entity->info->type!=type ) return ERR_9006;

	if( strcmp( lastname, "ProvisioningCode" )==0 )
	{
		if( buf ) len = strlen( buf );
		if( len ==0 )
			mib_set( CWMP_PROVISIONINGCODE, (void *)"");
		else if( len < 64 )
			mib_set( CWMP_PROVISIONINGCODE, (void *)buf);
		else
			return ERR_9007;

		return 0;
	}else
		return ERR_9005;
	return 0;
```

}

From this function, we check the input argument, type, if it is the same data type with the *entity->info->type*. If the check is passed, we will continue this function. If not, we will return ERR_9006(Invalid parameter type). Then, we use *entity->info->name* to find out the parameter we are going to handle. If the parameter can be handled in this callback function, set its value, and return 0 or 1. The return value 1 means the cwmpClient program has been committed this new value but not yet applied. The return value 0 means the cwmpClient program has been committed and applied the new value. If there is any problem when setting the new value, return the appropriate fault code defined in session 4.8. In our system, we use MIB APIs to commit the new values, and you can refer to *RTL867x_sw_api.doc* for these MIB APIs. The take-effect mechanism will be described in session 7.

6 Add a New Instance of a Multi-Instance Object

In this session, we will discuss how to manage the instances of a dynamic object. A dynamic object is object, and can have multiple instances. We use the object parameter, "*InternetGatewayDevice.Layer3Forwarding.Forwarding.*" to discuss this how-to.

Let's see how ACS manages the dynamic objects. ACS can use AddObject RPC method to create an instance of "*InternetGatewayDevice.Layer3Forwarding.Forwarding.*", and CPE will return the new created instance number to ACS if successful. For example, if the new created instance number is 5, the path name of the object instance, "*InternetGatewayDevice.Layer3Forwarding.Forwarding.5.*", must exist in the system. ACS can use SetParameterValues to set the values or GetParameterValues to get the values of these parameter with the prefix name of "*InternetGatewayDevice.Layer3Forwarding.Forwarding.5.*". If ACS wants to delete an existed instance, DeleteObject RPC Method will be used with a path name of the instance including the instance number. For example, if ACS wants to delete the previous created instance of the instance number 5, ACS sends DeleteObject RPC method with an argument, "*InternetGatewayDevice.Layer3Forwarding.Forwarding.5.*" to delete this instance.

The first step of creating a instance is to define the necessary tables. The tables/structures are defined below:

```
struct CWMP_OP tFW_Forwarding_OP = { NULL, objForwading };
struct CWMP_PRMT tLayer3ForwardingObjectInfo[] =
{
/*(name,          type,              flag,                       op)*/
{"Forwarding",  eCWMP_tOBJECT,  CWMP_WRITE|CWMP_READ,  &tFW_Forwarding_OP}
};
enum eLayer3ForwardingObject
{
 eFWForwarding
};
```

```
struct CWMP_NODE tLayer3ForwardingObject[] =
{
/*info,                                          leaf,        next)*/
{&tLayer3ForwardingObjectInfo[eFWForwarding],    NULL,        NULL},
{NULL,                                           NULL,        NULL}
};
```

tFW_Forwarding_OP defines an operation and only has a set-callback function, *objForward*, to be defined. **tLayer3ForwardingObjectInfo[]** defines the object parameter with the prefix path name, "*InternetGatewayDevice.Layer3Forwarding.*", and includes one entry, "Forwarding". From this entry, we can know "Forwarding" is object, read-write, and has an operation, tFW_Forwarding_OP. In session 5, we know a static object has no operation, but a dynamic object has its own operation to manage its object instances. **eLayer3ForwardingObject** is used to define an enumeration of parameters listed in the **tLayer3ForwardingObjectInfo** tables. **tLayer3ForwardingObject[]** is a "struct CWMP_NODE" array, and is used to link with its root node, **tIGDObject []**. It includes the parameters defined in the **tLayer3ForwardingObjectInfo []** and ends with a NULL entry. You will notice the **tLayer3ForwardingObject[0].next** is equal to NULL and is going to be initialized/managed by *objForwarding()*. The data structure definitions of the instance, "*InternetGatewayDevice.Layer3Forwarding.Forwarding.{i}.*", are defined below:

```
struct CWMP_PRMT tForwardingOjbectInfo[] =
{
/*(name,    type,                flag,                                  op)*/
{"0",       eCWMP_tOBJECT,       CWMP_READ|CWMP_WRITE|CWMP_LNKLIST,     NULL}
};
enum eForwardingOjbect
{
 eFW0
};
struct CWMP_LINKNODE tForwardingObject[] =
{
/*info,                              leaf,                      next,    sibling,    instnum)*/
{&tForwardingOjbectInfo[eFW0],  tForwardingEntityLeaf,     NULL,    NULL,       0},
};
```

tForwardingObjectInfo[] defines the *object* parameters with the prefix name, "*InternetGatewayDevice.Layer3Forwarding.Forwarding.*", and it includes one entry, "0". From this table, we know that "0" is object, read-write, linking-list, and has no operation. **eForwardingObject** is used to define an enumeration of parameters listed in the **tForwardingObjectInfo[]** tables. **tForwardingObject[]** is a "*struct CWMP_LINKNODE*" linking list, and is used to link with its root node, **tLayer3ForwardingObject[]**. It includes the parameters defined in the **tForwardingObjectInfo[],** and ends with NO NULL entry. **tForwardingObject** is the type of *struct CWMP_LINKNODE* that is used for dynamic object instances. *struct CWMP_LINKNODE* has 2 more extra fields than *struct CWMP_NODE*, that are *sibling* and *instnum*. This kind of table is connected into a linking list by using of the *sibling* field. Hence, we can easily manage this linking list (add, delete, and update elements

in the linking list). The *instnum* field in the *struct CWMP_LINKNODE* is used to save the instance number of the instance that is not saved in the *tForwardingObjectInfo[0].name*. The leaf parameters, tForwardingEntityLeaf, can use the way described in session 5 to implement.

The next step is to write the callback function to manage the instances after we define all the necessary tables. The prototype of the callback function is defined as:

*int objForwading(char *name, struct CWMP_LEAF *e, int type, void *data);*

The prototype is the same with set-callback function. The arguments for this callback function are:

name: the object path name that will be managed.

entity: the information of the parameter that is encapsulated in struct CWMP_LEAF.

type: the action type for this object as described in session 4.6. The action types include eCWMP_tINITOBJ, eCWMP_tADDOBJ, eCWMP_tDELOBJ, and eCWMP_tUPDATEOBJ.

data: this argument has a different meaning with a different action type.

The return values for this callback function are:

1: no error (the handled instance has NOT been applied)

0: no error (the handled instance has been applied)

-1: error (caused by internal error)

ERR_9xxx: as described in session 4.8

Give an example to explain how to use the callback function. This example is to manage this object, "*InternetGatewayDevice.Layer3Forwarding.Forwarding.*", and the C codes are shown below:

```
int objForwading(char *name, struct CWMP_LEAF *e, int type, void *data)
{
	struct CWMP_NODE *entity=(struct CWMP_NODE *)e;
	//fprintf( stderr, "%s:action:%d: %s\n", __FUNCTION__, type, name);

	switch( type )
	{
	case eCWMP_tINITOBJ:
		{
		int num=0,MaxInstNum=0,i;
		struct CWMP_LINKNODE **c = (struct CWMP_LINKNODE **)data;
		MIB_CE_IP_ROUTE_T *p,route_entity;

		if( (name==NULL) || (entity==NULL) || (data==NULL) ) return -1;

		num = mib_chain_total( MIB_IP_ROUTE_TBL );
		for( i=0; i<num;i++ )
		{
			p = &route_entity;
			if( !mib_chain_get( MIB_IP_ROUTE_TBL, i, (void*)p ) )
				continue;

			if( p->InstanceNum==0 ) //maybe createn by web or cli
```

```
        {
            MaxInstNum++;
            p->InstanceNum = MaxInstNum;
            mib_chain_update( MIB_IP_ROUTE_TBL, (unsigned char*)p, i );
        }else
            MaxInstNum = p->InstanceNum;
        if(create_Object(c,tForwardingObject,sizeof(tForwardingObject),1, MaxInstNum)<0)
            return -1;
        //c = & (*c)->sibling;
    }
    add_objectNum( name, MaxInstNum );
    return 0;
    }
```

We use the *type* argument to handle different actions. The first handled action is **eCWMP_tINITOBJ**. In this case, we need to initialize those instances that have existed in the configuration. We use *mib_chain_total()* to get the total number of route entries in our configuration. Then, use a for-loop to get each of route entries and add them to the parameter tree by *create_Object()*. Because each instance has its own instance number, if there is an entry whose instance number is 0, we must assign a unique instance number for it. Finally, use *add_objectNum()* to record that the current maximum instance number of this object is *MaxInstNum*. The purpose to record the maximum is to use this to get the next instance number when adding a new instance.

```
case eCWMP_tADDOBJ:
    {
    int ret;
    if( (name==NULL) || (entity==NULL) || (data==NULL) ) return -1;
    ret = add_Object( name,  (struct CWMP_LINKNODE **)&entity->next,
                        tForwardingObject,  sizeof(tForwardingObject),  data );
    if( ret >= 0 )
    {
        MIB_CE_IP_ROUTE_T fentry;
        memset( &fentry, 0, sizeof( MIB_CE_IP_ROUTE_T ) );
        fentry.InstanceNum = *(unsigned int*)data;
        fentry.FWMetric = -1;
        fentry.ifIndex = 0xff;
        mib_chain_add( MIB_IP_ROUTE_TBL, (unsigned char*)&fentry );
    }
    return ret;
    }
```

The second action to be handled in the *objForwarding()* is **eCWMP_tADDOBJ**. We use *add_Object()* to add a new instance for this multi-instance object in the parameter tree. If add_Object() returns successfully, the instance number for this new created instance will be returned at the same time (*data is used to save this number). After creating a new instance, we need to create a new route entry in our configuration (*mib_chain_add*() is used to add a new route entry), and we need to give this new entry some default values at the same time.

```
case eCWMP_tDELOBJ:
    {
```

```
        int ret, id;
        if( (name==NULL) || (entity==NULL) || (data==NULL) ) return -1;
        id = getChainID( entity->next, *(int*)data    );
        if(id==-1) return ERR_9005;
        mib_chain_delete( MIB_IP_ROUTE_TBL, id );
        ret = del_Object( name, (struct CWMP_LINKNODE **)&entity->next, *(int*)data );
        if( ret == 0 ) return 1;
    }
```

The third action to be handled in the *objForwarding()* is **eCWMP_tDELOBJ**. The **data* variable is an input argument to tell us which instance will be deleted by ACS. Because the given instance number is not the number of the chain record ID for this route MIB table. We can use the given instance number to search the route MIB table to find the real entry. After we find the chain record ID, call mib_chain_delete() to delete the specific entry, and call del_Object() to remove this entry from the parameter tree.

```
case eCWMP_tUPDATEOBJ:
    {
        int num=0,i;
        struct CWMP_LINKNODE *old_table;

        num = mib_chain_total( MIB_IP_ROUTE_TBL );
        old_table = (struct CWMP_LINKNODE*)entity->next;
        entity->next = NULL;
        for( i=0; i<num;i++ )
        {
            struct CWMP_LINKNODE *remove_entity=NULL;
            MIB_CE_IP_ROUTE_T *p,route_entity;

            p = &route_entity;
            if( !mib_chain_get( MIB_IP_ROUTE_TBL, i, (void*)p ) )
                continue;

            remove_entity = remove_SiblingEntity( &old_table, p->InstanceNum );
            if( remove_entity!=NULL )
            {
                add_SiblingEntity( (struct CWMP_LINKNODE **)&entity->next,
                                                        remove_entity );
            }else{
                unsigned int MaxInstNum=p->InstanceNum;

                add_Object( name, (struct CWMP_LINKNODE **)&entity->next,
                        tForwardingObject, sizeof(tForwardingObject), &MaxInstNum );
                if(MaxInstNum!=p->InstanceNum)
                {
                    p->InstanceNum = MaxInstNum;
                    mib_chain_update( MIB_IP_ROUTE_TBL, (unsigned char*)p, i );
                }
            }
        }

        if( old_table )
            destroy_ParameterTable( (struct CWMP_NODE *)old_table );

        return 0;
    }
}
```

```
        return -1;
}
```

The last to be handled in the *objForwarding()* is **eCWMP_tUPDATEOBJ**. This action type is called only when cwmpClient program starts a new TR-069 session. Because there are other interfaces to be able to change the system configuration (ex. Web or CLI), the parameter tree may not synchronize with the latest configuration. Hence, we can use such an action to update the parameter tree to keep the synchronization. We use *mib_chain_total()* to get the total entry number of the route table, use the *old_table* variable to save the old linking list of the instances, and reset the *entity->next* to NULL. Then, use a for-loop to get each of route entries from the route table, and use *remove_SiblingEntity()* to find if there is such a instance number in the old liking list, *old_table*, that is the same with that of the route entry got by *mib_chain_get()*. If yes, use *add_SiblingEntity()* to add it into the new linking list pointed by *entity->next*. If not found, use *add_Object()* to add the new instance, and update the instance number saved in the configuration for this route entry (*mib_chain_update()* is used to update a chain entry). After the for-loop, if *old_table* is not empty, call destroy_ParamterTable() to remove those instances that had been removed from the configuration by other interfaces (ex. Web or CLI).

7 How to Apply the New Parameter Values

Although TR-069 defines that the new values can take effect immediately or after reboot, to avoid a reboot whenever ACS sets new values, we provide a mechanism to apply the new parameter values. We will describe this mechanism in the following paragraphs.

The prototype of the take-effect function is defined as:

*int apply_Layer3Forwarding(int action_type, int id, void *olddata);*

The arguments for this take-effect function are:

action_type: the action type that is going to be handled. We define 3 action types: **CWMP_START**, **CWMP_STOP**, and **CWMP_RESTART**.

id: be used for the chain record ID, or for other purpose (ex. other index)

olddata: be used for the old data of a specific chain record ID, or for other purpose.

The return values for this take-effect function are:

0: no error

-1: error

Give an example of the take-effect mechanism. This example is the take-effect function for the parameters with the prefix parameter path name, "*InternetGatewayDevice.Layer3Forwarding.Forwarding.{i}*.".

*int apply_Layer3Forwarding(int action_type, int id, void *olddata)*

```
{
    MIB_CE_IP_ROUTE_T *pOldRoute=olddata;
    MIB_CE_IP_ROUTE_T RouteEntry, *pNewRoute=NULL;

    //got the lastest entry
    if( mib_chain_get( MIB_IP_ROUTE_TBL, id,    (void*)&RouteEntry )!=0 ) //0:error
        pNewRoute =    &RouteEntry;

    switch( action_type )
    {
    case CWMP_RESTART:
        if( pOldRoute &&
            pOldRoute->Enable &&
            ( pOldRoute->ifIndex==0xff &&
              pOldRoute->nextHop[0]==0 &&
              pOldRoute->nextHop[0]==0 &&
              pOldRoute->nextHop[0]==0 &&
              pOldRoute->nextHop[0]==0 )
          )
            route_cfg_modify( pOldRoute, 1 );
    case CWMP_START:
        if( pNewRoute &&
            pNewRoute->Enable &&
            ( pNewRoute->ifIndex==0xff &&
              pNewRoute->nextHop[0]==0 &&
              pNewRoute->nextHop[0]==0 &&
              pNewRoute->nextHop[0]==0 &&
              pNewRoute->nextHop[0]==0 )
          )
            route_cfg_modify( pNewRoute, -1 );
        break;
    case CWMP_STOP:
        if( pOldRoute &&
            pOldRoute->Enable &&
            ( pOldRoute->ifIndex==0xff &&
              pOldRoute->nextHop[0]==0 &&
              pOldRoute->nextHop[0]==0 &&
              pOldRoute->nextHop[0]==0 &&
              pOldRoute->nextHop[0]==0 )
          )
            route_cfg_modify( pOldRoute, 1 );
        break;
    default:
        return -1;
    }
    return 0;
}
```

In this function, we can handle different types of actions by using of the *action_type* variable. The argument, *id*, is used for the chain record ID of the MIB route table, and the argument, *olddata*, is used to save the previous route data without modification. Hence, we can get the latest data of the specific record from the MIB route table with *id* variable. *route_cfg_modify()* is the really take-effect function to modify the route table in the system. If the *action_type* is equal to **CWMP_STOP**, we need to delete the old route entry from the system. If the action_type is equal to **CWMP_START**, we need to add the route entry got

from the MIB configuration to the system. If the action_type is equal to **CWMP_RESTART**, we need to delete the old route entry from the system, and add the new route entry to the system.

After we write the take-effect function, there should be 3 conditions that need to call the take-effect function. They are AddObject, DeleteObject, and SetParameterValues RPC Methods that are called by ACS.

First, we talk about the AddObject condition. When ACS calls AddObject RPC method to create a new instance of a multi-instance object, most of the default values for this new instance are disabled. Hence, we could ignore this condition unless the default values for this new instance are enabled.

The second condition is DeleteObject. Because the purpose of DeleteObject RPC method is to delete an existed instance, we can call the take-effect function directly with the action_type, **CWMP_STOP**. Give an example of deleting a "*InternetGatewayDevice.Layer3Forwarding.Forwarding.{i}*." instance below:

```
int objForwading(char *name, struct CWMP_LEAF *e, int type, void *data)
{
	struct CWMP_NODE *entity=(struct CWMP_NODE *)e;
	//fprintf( stderr, "%s:action:%d: %s\n", __FUNCTION__, type, name);

	switch( type )
	{
	...
	case eCWMP_tDELOBJ:
		{
			int ret, id;
			if( (name==NULL) || (entity==NULL) || (data==NULL) ) return -1;
			id = getChainID( entity->next, *(int*)data    );
			if(id==-1) return ERR_9005;
#ifdef _CWMP_APPLY_
			{
				MIB_CE_IP_ROUTE_T route_old;
				if( mib_chain_get( MIB_IP_ROUTE_TBL, id, (void*)&route_old) ) //1:success
				{
				apply_Layer3Forwarding( CWMP_STOP, id, &route_old );
				}
			}
#endif
			mib_chain_delete( MIB_IP_ROUTE_TBL, id );
			ret = del_Object( name, (struct CWMP_LINKNODE **)&entity->next, *(int*)data );
#ifndef _CWMP_APPLY_
			if( ret == 0 ) return 1;
#endif
			return ret;
		}//case eCWMP_tDELOBJ
	...
	} //switch
	return -1;
}
```

In the code, we use the **_CWMP_APPLY_** flag to enable the take-effect mechanism. If

_CWMP_APPLY_ is not defined, cwmpClient will not support the take-effect mechanism. From the above code, when handling the eCWMP_tDELOBJ condition, we will get the old data of the route entry by *mib_chain_get()*, and directly call the take-effect function, *apply_Layer3Foreading()*, with CWMP_STOP action, chain record ID, and the old data of the route entry. After that, if enabling the take-effect mechanism, objForwarding() should return 0 that means the new setting has been committed and applied.

The last condition that needs to take-effect is SetParameterValues. This is more complex case to be discussed. According TR-069, an ACS can set more than one parameter by SetParameterValues RPC method. When one of those parameters that are set by a SetParameterValues transaction is error, the take-effect function for those right parameters cannot be applied. In other words, the CPE must apply the changes to all of the specified parameters atomically. Either all of the value changes are applied together, or none of the changes are applied at all. To meet this requirement, we implement a simple priority queue. The simple priority queue is used to save the take-effect functions and theirs arguments in a priority way. Why needs priority? There are some take-effect functions that may need to execute before other take-effect functions. We can use the function defined below to add the take-effect function to the simple priority queue:

*int apply_add(int priority, void *cb_fun, int action, int id, void *olddata, int size);*

The arguments for this function are:

priority: the priority that includes 5 priorities:

a. **CWMP_PRI_SH**: the highest priority (super high)
b. **CWMP_PRI_H**: the second high priority (high)
c. **CWMP_PRI_N**: the third high priority (normal)
d. **CWMP_PRI_L**: the fourth high priority (low)
e. **CWMP_PRI_SL**: the lowest priority (super low)

cb_fun: the take-effect function that will be executed

action: the action_type argument for the take-effect function

id: the id argument for the take-effect function

olddata: the olddata argument for the take-effect function

size: the size of the olddata.

The return values for this function are:

0: no error

-1: error

Let's show an example for SetParameterValues condition. The C codes are listed below:

```
int setLayer3Fw(char *name, struct CWMP_LEAF *entity, int type, void *data)
{
	char  *lastname = entity->info->name;

	if( (name==NULL) || (data==NULL) || (entity==NULL)) return -1;
	if( entity->info->type!=type ) return ERR_9006;
```

```
        if( strcmp( lastname, "DefaultConnectionService" )==0 )
        {
                char *buf=data;
                if( buf==NULL ) return ERR_9007;
                if( strlen(buf)==0 ) return ERR_9007;
                if( setDefaultRoute( buf ) < 0 ) return ERR_9007;
#ifdef _CWMP_APPLY_
                apply_add( CWMP_PRI_N, apply_DefaultRoute, CWMP_RESTART, 0, NULL, 0 );
                return 0;
#else
                return 1;
#endif //_CWMP_APPLY_
        }else{
                return ERR_9005;
        }

        return 0;
}
```

This example is to set the parameter, "*InternetGatewayDevice.Layer3Forwarding.DefaultConnectionSerivce*". We use the *apply_add()* to add this take-effect function into the sample priority queue. From the argument of *apply_add()*, we know that the priority of the take-effect function is normal (**CWMP_PRI_N**), the callback function is *apply_DefaultRoute()*, and the action_type for the take-effect function is **CWMP_RESTART** that means the old default route will be deleted and the new default route will be added. The rest arguments of *id*, *olddata*, and *size* can be ignored in this case.

8 Web Configuration for TR-069

In the CPE Web pages, login with the administrator account, and click "Admin" ➔ "TR-069 Config". You will see the configuration page of TR-069. In this page, there are 4 sessions to configure TR-069 including ACS configuration, connection-request configuration, debug/flag configuration, and certificate management.

8.1 ACS Configuration

The part of ACS configuration is shown in the figure below.

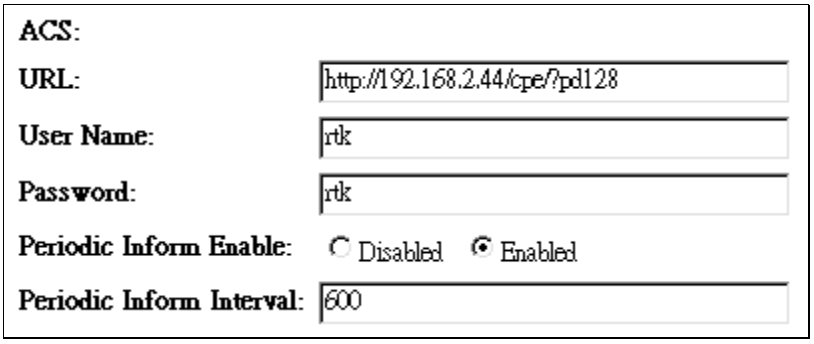
URL: the ACS URL

User Name: the username for HTTP authentication

Password: the password for HTTP authentication

Periodic Inform Enable: enable or disable that the CPE periodically sends CPE information to the ACS using the Inform method call.

Periodic Inform Interval: the duration in seconds of the interval for which the CPE must attempt to connect with the ACS and call the Inform method if Periodic Inform is enabled.

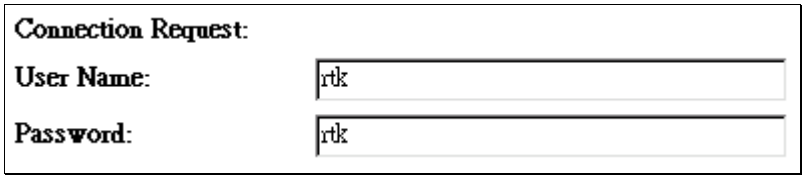


8.2 Connection Request Configuration

The part of Connection Request configuration is shown in the figure below.

User Name: the username used to authenticate an ACS making a Connection Request to the CPE.

Password: the password used to authenticate an ACS making a Connection Request to the CPE.



8.3 Debug/Flag Configuration

The part of debug/flag configuration is shown in the figure below.

ACS Certificates CPE: whether or not the ACS needs to certificate the CPE. If yes, the CPE will also ensure that the Common Name (CN) component of the Subject field in the certificate exactly matches the host portion of the ACS URL known to the CPE.

Show Message: enable or disable to show debug messages on console

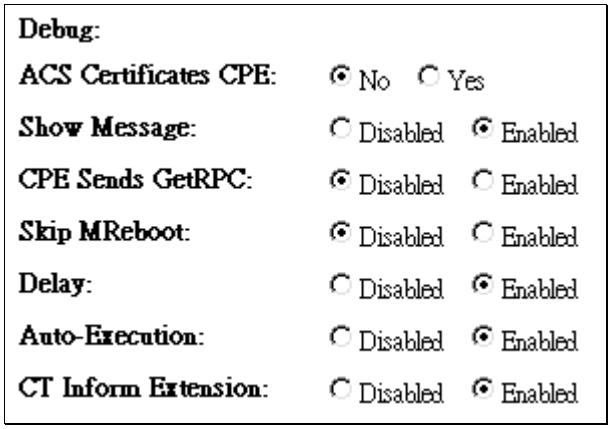
CPE Sends GetPRC: whether or not the CPE sends GetPRCMethod method call to ACS.

Skip MReboot: whether or not the CPE skips the MReboot event code. If enabled, the MReboot event code will not be carried in the Inform message.

Delay: enable or disable the delay time to connect to the ACS. If enabled, the default delay time is 30 seconds to wait the creation of the WAN connections

Auto-Execution: whether or not cwmpClient runs at the boot time.

CT Inform Extension: enable or disable those extra fields that china-telecom extends in the Inform message.



8.4 Certificate Management

The part for certificate management is shown in the figure below.

CPE Certificate Password: the password to decrypt the CPE certificate for the private key.

CPE Certificate: upload the CPE certificate including the private key.

CA Certificate: upload the CA certificates.