



# **Configuration Management System (CMS) Developer's Guide**

BROADCOM CONFIDENTIAL

## Revision History

<b>Revision</b>	<b>Date</b>	<b>Change Description</b>
CPE-SWUM102-R	07/03/14	<b>Updated:</b> <ul style="list-style-type: none"><li>• <a href="#">Figure 2: “API Layering in the CMS Architecture,” on page 16</a></li><li>• <a href="#">“Major Concepts” on page 18</a></li><li>• <a href="#">Table 3: “Possible Object Element Attributes,” on page 29</a></li><li>• <a href="#">Table 4: “Parameter Element Attributes,” on page 31</a></li><li>• <a href="#">“Profiles” on page 34</a></li><li>• <a href="#">“Adding and Deleting Data Model Profiles” on page 47</a></li><li>• <a href="#">Section 6: “Automatic Generation of Source and Header Files from the Data Model,” on page 50</a></li><li>• <a href="#">Figure 7: “Automatic File Generation Components,” on page 50</a></li><li>• <a href="#">Table 6: “Auto-generated Source and Header Files,” on page 50</a></li><li>• <a href="#">“Merge Commands” on page 51</a></li><li>• <a href="#">“Generating a Report of Supported Objects, Parameters, and Profiles” on page 53</a></li><li>• <a href="#">Section 10: “Data Access Layer,” on page 85</a></li><li>• <a href="#">See “Sharing RUT Functions with DAL” on page 99.</a></li><li>• <a href="#">Table 22: “Acronyms and Abbreviations,” on page 159</a></li></ul> <b>Added:</b> <ul style="list-style-type: none"><li>• <a href="#">“Selecting the Data Model Mode” on page 18</a></li><li>• <a href="#">Section 3: “Major Updates in CMS for TR-181,” on page 22</a></li><li>• <a href="#">“Centralized Control of Starting/Stopping Applications” on page 20</a></li></ul>
CPE-SWUM101-R	01/21/14	<b>Updated:</b> <ul style="list-style-type: none"><li>• <a href="#">“Generating a Report of All Objects and Parameters” on page 41</a></li></ul>
CPE-SWUM100-R	03/21/13	Initial release

Broadcom Corporation  
5300 California Avenue  
Irvine, CA 92617

© 2014 by Broadcom Corporation  
All rights reserved  
Printed in the U.S.A.

Broadcom®, the pulse logo, Connecting everything®, and the Connecting everything logo are among the trademarks of Broadcom Corporation and/or its affiliates in the United States, certain other countries and/or the EU. Any other trademarks or trade names mentioned are the property of their respective owners.

# Table of Contents

<b>About This Document</b> .....	11
Purpose and Audience .....	11
How To Use This Document .....	11
Acronyms and Abbreviations.....	11
Document Conventions .....	11
References .....	12
<b>Technical Support</b> .....	12
<b>Section 1: Introduction</b> .....	<b>13</b>
<b>Scope of CMS</b> .....	14
<b>Section 2: Overview</b> .....	<b>15</b>
<b>Architecture</b> .....	15
<b>API Layering</b> .....	16
<b>Major Concepts</b> .....	18
Use of Broadband Forum Data Models.....	18
Selecting the Data Model Mode .....	18
Separation of Management Applications from Runtime Configuration.....	19
Centralized Control of Starting/Stopping Applications.....	20
Centralized Collection and Processing of Asynchronous Events.....	21
Using Messages to Communicate.....	21
<b>Section 3: Major Updates in CMS for TR-181</b> .....	<b>22</b>
<b>Section 4: The Data Model</b> .....	<b>28</b>
<b>Object and Parameter Definition</b> .....	29
<b>Valid Strings Arrays</b> .....	33
<b>Profiles</b> .....	34
<b>Section 5: Data Model Designer</b> .....	<b>36</b>
<b>Data Model Designer Requirements</b> .....	36
<b>Running the Data Model Designer</b> .....	36
<b>Edit Modes</b> .....	37
TR-editor Mode .....	37
BRCM-dev Mode.....	37
Custom-dev Mode.....	37
Read-Only Mode .....	38
<b>Viewing the Data Model</b> .....	39
<b>Adding New Objects to the Data Model</b> .....	40
<b>Adding New Parameters to the Data Model</b> .....	42

Deleting Objects or Parameters .....	45
Editing an Object or Parameter .....	45
Viewing, Editing, Adding, or Deleting a Valid String Array .....	46
Valid String Arrays and the Broadband Forum Data Model .....	46
Adding and Deleting Data Model Profiles .....	47
Saving Changes.....	49
<b>Section 6: Automatic Generation of Source and Header Files from the Data Model..</b>	<b>50</b>
Merge Commands .....	51
Generating a Report of Supported Objects, Parameters, and Profiles .....	53
Supported Profiles Report Considerations.....	53
<b>Section 7: System Bootup .....</b>	<b>54</b>
CFE and Kernel Boot.....	54
init process .....	54
/etc/inittab.....	54
::sysinit:/bin/sh -l -c "bcm_boot_launcher start".....	54
::respawn:/bin/sh -l -c consold.....	55
::shutdown:/bin/sh -l -c "bcm_boot_launcher stop" .....	55
/etc/profile .....	55
Smd.....	56
Stage 1 .....	56
MDM Initialization.....	56
Stage 2.....	56
Inetd Style Dynamic Application Launching.....	57
<b>Section 8: HTTPD .....</b>	<b>58</b>
Introduction.....	58
General Overview of httpd.....	58
Displaying a Page.....	60
Page From HTML Directory .....	60
Page Generated from C Code.....	62
Reading Input from User.....	62
CGI Method .....	62
CMD Method .....	63
Preventing Cross Site Request Forgery Attacks.....	63
Adding Web Pages to the Left Panel Menu.....	64
Customizing the Logo.....	64
<b>Section 9: Accessing the MDM using the Object Layer Interface.....</b>	<b>65</b>
Key MDM Concepts .....	65
MDM Object .....	65

Instance ID Stack .....	65
<b>Initializing the MDM .....</b>	<b>66</b>
<b>Acquiring and Releasing a Lock .....</b>	<b>66</b>
<b>Reading from the MDM .....</b>	<b>68</b>
Using cmsObj_get() .....	68
Using cmsObj_get() to Get Type 1 or Type 2 Objects .....	69
Using cmsObj_getNext() .....	69
Early Exit from cmsObj_getNext() .....	70
Returning an Object or iidStack from cmsObj_getNext() .....	71
Using cmsObj_getNextInSubTree() .....	72
Using cmsObj_getAncestor() .....	73
Using the OGF_NO_VALUE_UPDATE Flag .....	74
<b>Writing to the MDM .....</b>	<b>75</b>
Setting String Parameters .....	77
<b>Adding a New Instance of an Object .....</b>	<b>79</b>
<b>Deleting an Existing Instance of an Object .....</b>	<b>81</b>
Deleting Multiple Instances Inside the While Loop .....	82
<b>Saving Changes to Flash Memory .....</b>	<b>84</b>
<b>Section 10: Data Access Layer .....</b>	<b>85</b>
<b>Section 11: Query Data Model (QDM) Library .....</b>	<b>86</b>
<b>Section 12: Runtime Configuration Layer .....</b>	<b>87</b>
<b>At System Bootup .....</b>	<b>88</b>
<b>When an Object is Added .....</b>	<b>88</b>
<b>When an Object is Modified .....</b>	<b>90</b>
<b>When an Object is Deleted .....</b>	<b>91</b>
<b>RCL Design Patterns .....</b>	<b>91</b>
RCL Design Pattern 1 .....	92
RCL Design Pattern 2 .....	94
<b>MDM State During Add, Set, and Delete Operations .....</b>	<b>96</b>
During an Add Operation .....	96
4.02L.01 and 4.02L.02 Release Version Add Operations .....	96
During a Set Operation .....	97
During a Delete Operation .....	97
<b>Other RCL Design Issues .....</b>	<b>97</b>
RCL Handler Function Required For All Objects .....	97
Modifying Object Inside RCL Handler Function .....	97
Use of mdmLibCtx.allocFlags .....	98
Avoiding Deadlock .....	98

Doing System Action .....	98
Starting an Application .....	98
<b>Section 13: Runtime Utilities .....</b>	<b>99</b>
<b>Naming of RUT Functions .....</b>	<b>99</b>
<b>Sharing RUT Functions with DAL .....</b>	<b>99</b>
<b>Section 14: System Status Layer .....</b>	<b>100</b>
<b>STL Design Patterns .....</b>	<b>100</b>
STL Design Pattern 1: Getting Statistics from Device Driver and Clearing Statistics .....	100
STL Design Pattern 2: Getting One Time Parameters .....	101
STL Design Pattern 3: Getting Status from Other Applications .....	102
STL Design Pattern 4: STL Handler Functions for Configuration Objects .....	102
<b>Section 15: Messaging .....</b>	<b>103</b>
<b>CMS Messaging API .....</b>	<b>103</b>
<b>CMS Messaging in RCL/STL/RUT .....</b>	<b>106</b>
Declaring Message Header on the Stack .....	107
Declaring Message Header Plus Variable Length Data Section on the Stack .....	107
Allocating Message Header Plus Variable Length Data Section from the Heap .....	109
Scenarios to Avoid .....	110
Unexpected Response Messages .....	110
Deadlock in SMD .....	111
Deadlock in SSK .....	111
<b>CMS Messaging in Applications .....</b>	<b>112</b>
Initializing and Destroying the Message Handle .....	112
Using the Message Handle in a Select Loop .....	112
Registering Interest in an Event .....	114
Simple Event Registration .....	115
Conditional Event Registration .....	116
Requesting a Delayed Event Message .....	116
Some Useful Messages .....	117
CMS_MSG_REGISTER_EVENT_INTEREST/CMS_MSG_UNREGISTER_EVENT_INTEREST ....	117
CMS_MSG_WAN_LINK_UP/CMS_MSG_WAN_LINK_DOWN .....	118
CMS_MSG_WAN_CONNECTION_UP/CMS_MSG_WAN_CONNECTION_DOWN .....	118
CMS_MSG_GET_WAN_LINK_STATUS .....	119
CMS_MSG_GET_WAN_CONN_STATUS .....	119
CMS_MSG_START_APP .....	120
CMS_MSG_RESTART_APP .....	120
CMS_MSG_STOP_APP .....	121

CMS_MSG_IS_APP_RUNNING.....	121
CMS_MSG_REBOOT_SYSTEM.....	122
CMS_MSG_SYSTEM_BOOT .....	122
CMS_MSG_INTERNAL_NOOP.....	122
<b>Defining New Messages in a Modular Way .....</b>	<b>123</b>
Step 1: Allocate a Block of Message Type Numbers .....	123
Step 2: Create Your Own Message Header File .....	124
Step 3: Using Feature Specific Message Types .....	124
<b>Message Processing in SMD .....</b>	<b>125</b>
<b>Section 16: Command Line Interface Library .....</b>	<b>126</b>
Menu Driven CLI .....	126
Command Line CLI .....	126
<b>Section 17: TR69C .....</b>	<b>127</b>
Dynamic Launch of TR69C .....	127
Inform Event Code .....	128
Bound Interface Name .....	128
Generating the Connection Request URL.....	129
TR69C Startup Algorithm .....	129
Use of CMS Data Model .....	130
<b>Section 18: Directory Layout and the GPL.....</b>	<b>131</b>
OS Independence .....	132
GPL Issues .....	132
License of Broadcom-developed Libraries in the Public Directory .....	132
Applications in the Public Directory .....	132
Summary of Linking Rules.....	133
The GPL and Kernel Drivers and Modules.....	133
<b>Section 19: Adding a New Application or Command.....</b>	<b>134</b>
Adding the Application/Command to the Build System .....	134
Allocating an Entity ID .....	135
Creating a CmsEntityInfo Entry .....	136
Patterns for Applications.....	138
Applications that Access the MDM.....	138
Applications that Do Not Access the MDM.....	141
Applications Launched by SMD During Bootup .....	141
Starting the Application .....	142
From inittab and bcm_boot_launcher.....	142
Launched by smd in Stage 1.....	142
Launched by smd in Stage 2.....	142

Event Interest or Delayed Message .....	143
Applications with Server Sockets .....	143
Application Started by RCL Handler Functions .....	143
<b>Stopping the Application .....</b>	<b>145</b>
Self Exit .....	145
Applications Stopped by RCL Handler Functions .....	145
<b>Executing the Command .....</b>	<b>146</b>
<b>Section 20: The CMS Utilities Library .....</b>	<b>147</b>
Logging (cms_log.h) .....	147
Memory Allocation (cms_mem.h) .....	148
Timestamps (cms_tms.h) .....	149
Timer (cms_tmr.h) .....	149
Assert (cms_ast.h) .....	149
Doubly-linked List (cms_dlist.h) .....	149
Persistent Scratch Pad (cms_psp.h) .....	149
Base64 (cms_base64.h) .....	149
HexBinary (cms_hexbinary.h) .....	149
XML (cms_xml.h) .....	150
LED (cms_led.h) .....	150
LZW Compression/Decompression (cms_lzw.h) .....	150
<b>Section 21: Board and Device Control Libraries .....</b>	<b>151</b>
<b>Section 22: Doxygen API Documentation .....</b>	<b>152</b>
Writing Doxygen-Compatible Comment Blocks .....	152
Documenting a Function .....	152
Documenting a Structure .....	153
Documenting a macro, #define, or typedef .....	153
Documenting an Enumeration .....	154
Generating Documentation with Doxygen .....	154
<b>Section 23: Debugging .....</b>	<b>155</b>
Using Log Messages .....	155
Startup Debugging .....	155
Dumping the MDM .....	156
Dumping the Persistent Scratch Pad .....	156
Memory Leak Tracing .....	157
Enabling Shell Timeout .....	158
<b>Appendix A: Acronyms and Abbreviations .....</b>	<b>159</b>
<b>Appendix B: Compiling the Data Model Designer .....</b>	<b>162</b>



## List of Figures

Figure 1: Applications View of the CMS Architecture .....	15
Figure 2: API Layering in the CMS Architecture .....	16
Figure 3: Separation of Management Application Protocol and System Configuration .....	20
Figure 4: IPv4 and IPv6 State Machines in IP.Interface .....	24
Figure 5: Setting the Edit Mode to Custom-dev .....	38
Figure 6: CMS Data Model Designer .....	39
Figure 7: Automatic File Generation Components .....	50
Figure 8: Functional Block Overview .....	59
Figure 9: <%ejGet Tag Processing .....	61
Figure 10: InstanceId Stack Structure .....	66
Figure 11: CMS Message Header .....	104
Figure 12: Application, Message Handle, and RCL/STL/RUT Functions .....	106
Figure 13: CMS Directory Layout .....	131

## List of Tables

Table 1: API Library Groupings.....	18
Table 2: Data-Model Directory Files .....	28
Table 3: Possible Object Element Attributes.....	29
Table 4: Parameter Element Attributes .....	31
Table 5: Broadband Forum Parameter Types and Settings .....	44
Table 6: Auto-generated Source and Header Files .....	50
Table 7: CMS_MSG_REGISTER_EVENT_INTEREST/CMS_MSG_UNREGISTER_EVENT_INTEREST ..	117
Table 8: CMS_MSG_WAN_LINK_UP/CMS_MSG_WAN_LINK_DOWN .....	118
Table 9: CMS_MSG_WAN_CONNECTION_UP/CMS_MSG_WAN_CONNECTION_DOWN .....	118
Table 10: CMS_MSG_GET_WAN_LINK_STATUS .....	119
Table 11: CMS_MSG_GET_WAN_CONN_STATUS .....	119
Table 12: CMS_MSG_START_APP .....	120
Table 13: CMS_MSG_RESTART_APP .....	120
Table 14: CMS_MSG_STOP_APP .....	121
Table 15: CMS_MSG_IS_APP_RUNNING .....	121
Table 16: CMS_MSG_REBOOT_SYSTEM .....	122
Table 17: CMS_MSG_SYSTEM_BOOT .....	122
Table 18: CMS_MSG_INTERNAL_NOOP .....	122
Table 19: Inform Event Codes .....	128
Table 20: BoundIfName Values .....	128
Table 21: CmsEntityInfo Entry Keywords .....	136
Table 22: Acronyms and Abbreviations .....	159

---

## About This Document

### Purpose and Audience

The main purpose of this document is to describe to developers how to write software for CMS. A secondary purpose is to provide an overview of the CMS architecture and features in the BCA CPE reference software. This document is intended for software engineers and others working with Broadcom® CPE devices.

### How To Use This Document

This document is intended for CMS developers, and complements the API documentation in the header files of the source code. However, this document is not a substitute for the API documentation. The API documentation in the CMS header files have very precise descriptions of the behavior of all the functions and their expected inputs and outputs. This document does not go into those details; instead it shows examples of typical usage scenarios for the most common API functions.



**Note:** Do not use this document as the only source of information on CMS development. See also [Section 20: “Doxygen API Documentation,” on page 139](#) for how to use Doxygen to browse the documentation and refer also to the API.

### Acronyms and Abbreviations

In most cases, acronyms and abbreviations are defined on first use.

Acronyms and abbreviations in this document are also defined in [“Acronyms and Abbreviations” on page 146](#).

For a comprehensive list of acronyms and other terms used in Broadcom documents, go to:  
<http://www.broadcom.com/press/glossary.php>.

### Document Conventions

The following conventions may be used in this document:

Convention	Description
<b>Bold</b>	User input and actions: for example, type <b>exit</b> , click <b>OK</b> , press <b>Alt+C</b>
Monospace	Code: <code>#include &lt;iostream&gt;</code> HTML: <code>&lt;td rowspan = 3&gt;</code> Command line commands and parameters: <code>w1 [-1] &lt;command&gt;</code>
<code>&lt; &gt;</code>	Placeholders for <i>required</i> elements: enter your <code>&lt;username&gt;</code> or <code>w1 &lt;command&gt;</code>
<code>[]</code>	Indicates <i>optional</i> command-line parameters: <code>w1 [-1]</code> Indicates bit and byte ranges (inclusive): <code>[0:3]</code> or <code>[7:0]</code>

## References

The references in this section may be used in conjunction with this document.



**Note:** Broadcom provides customer access to technical documentation and software through its Customer Support Portal (CSP) and Downloads and Support site (see [Technical Support](#)).

For Broadcom documents, replace the “xx” in the document number with the largest number available in the repository to ensure that you have the most current version of the document.

<b>Document (or Item) Name</b>	<b>Number</b>	<b>Source</b>
<b>Broadcom Items</b>		
[1] <i>BCA Modular Software Download</i>	CPE-AN1xx-R	Broadcom CSP
<b>Other Items</b>		
[2] <i>TR-069 Amendment 2, CPE WAN Management Protocol v1.1, – December 2007</i>	–	Broadband Forum Technical Reports
[3] <i>TR-098 Amendment 1, Internet Gateway Device Data Model for TR-069, December 2006</i>	–	Broadband Forum Technical Reports
[4] <i>TR-104, DSLHome™ Provisioning Parameters for VoIP CPE, September 2005</i>	–	Broadband Forum Technical Reports
[5] <i>TR-111, DSLHome™ Applying TR-069 to Remote Management of Home Networking Devices, December 2005</i>	–	Broadband Forum Technical Reports
[6] <i>TR-181, Device Data Model for TR-069 (Issue 2, Amendment 6) November 2012</i>	–	Broadband Forum Technical Reports
[7] <i>TR-140, TR-069 Data Model for Storage Service Enabled Devices (Amendment 1) April 2010</i>	–	Broadband Forum Technical Reports

## Technical Support

Broadcom provides customer access to a wide range of information, including technical documentation, schematic diagrams, product bill of materials, PCB layout information, and software updates through its customer support portal (<https://support.broadcom.com>). For a CSP account, contact your Sales or Engineering support representative.

In addition, Broadcom provides other product support through its Downloads and Support site (<http://www.broadcom.com/support/>).

# Section 1: Introduction

The userspace management system of the Broadcom® Broadband Carrier Access (BCA) customer premises equipment (CPE) is called the Configuration Management System, or CMS. The purpose of CMS is to coordinate the activities of and to provide functionality beyond:

- The Board Support Package (BSP), which includes the toolchain, uclibc, bootloader, kernel, and kernel device drivers.
- Freely available open source userspace applications, such as busybox, iptables, pppd, dhcpd, etc.

The coordination and functionality provided by CMS include (this is not a complete list):

- Managing the lifecycle of applications, including starting, stopping, and restarting them if they crash.
- Facilitating the configuration of applications through configuration files, user interfaces (e.g. WebUI, command line interface), and remote management protocols (e.g. TR-69).
- Coordinating system actions in response to asynchronous events, such as link up or down.
- Providing development and debug tools.

Starting from release 4.14L.01, CMS began a major enhancement cycle to become more modular. Modularity means the ability to add new applications, libraries, and other functionality to CMS without modifying the core CMS files. Modularity is important to our customers because it allows them to merge new releases from Broadcom with minimal conflicts with their own customization. CMS modularity features will be introduced over several releases. This document describes modular features present in the 4.14L.01 release and will be updated when more modular features are added.

Summary of Modular Enhancements in 4.14L.01:

- Modular system bootstrap. See [Section 7: "System Bootstrap," on page 54](#).
- Modular data model files. See [Section 4: "The Data Model," on page 28](#) and [Section 6: "Automatic Generation of Source and Header Files from the Data Model," on page 50](#).
- Modular CMS entity info entries. See [Section 19: "Adding a New Application or Command," on page 134](#).
- Modular CMS message definitions. See [Section 15: "Messaging," on page 103](#).

Starting in the 4.14L.XX release series and continuing in future release series, support for the Broadband Forum TR-181 [6] data model is being added to CMS. As of release 4.16L.02, support for TR-181 is not complete. However, there is enough TR-181 code that customers who are considering using TR-181 can look at the code and evaluate the SDK in "Pure181" mode. More TR-181 support will be added in future releases. See [Section 3: "Major Updates in CMS for TR-181," on page 22](#) for more details.

---

## Scope of CMS

As stated previously, CMS is a userspace management system. Although it operates in userspace, CMS controls and interacts with the kernel and kernel device drivers, especially the device drivers written by Broadcom. In this way CMS has some influence over the design and implementation of the kernel components.

CMS has a very strong influence over the applications and subsystems in userspace, although the amount of influence varies among the various applications and subsystems. For example, `httpd`, which implements the WebUI, and `tr69c`, which implements the TR-69 remote management protocol, have very strong dependencies on CMS. Other applications, such as `dhcpcd`, `pppd`, and `mcpd`, are configured by CMS but do not use most of the functions of CMS.

Other major subsystems, such as WLAN (`wlmgmr`) and VoIP, were developed outside of the CMS framework, so the bulk of their code is not aware of CMS. However, they have been modified to interface with CMS where necessary. Finally, there are other applications that have no interactions with CMS. These applications include (this is not a complete list) `bpmctl`, the command program to control the Broadcom Kernel Buffer Pool Manager, the "stress" test program, and the Dhrystone performance benchmark program. These applications do not use any CMS functions, and CMS does not call them.

Userspace applications and subsystems are not required to use CMS. They can be started at bootup time or from the command line and once running never talk to CMS or use any CMS facilities.

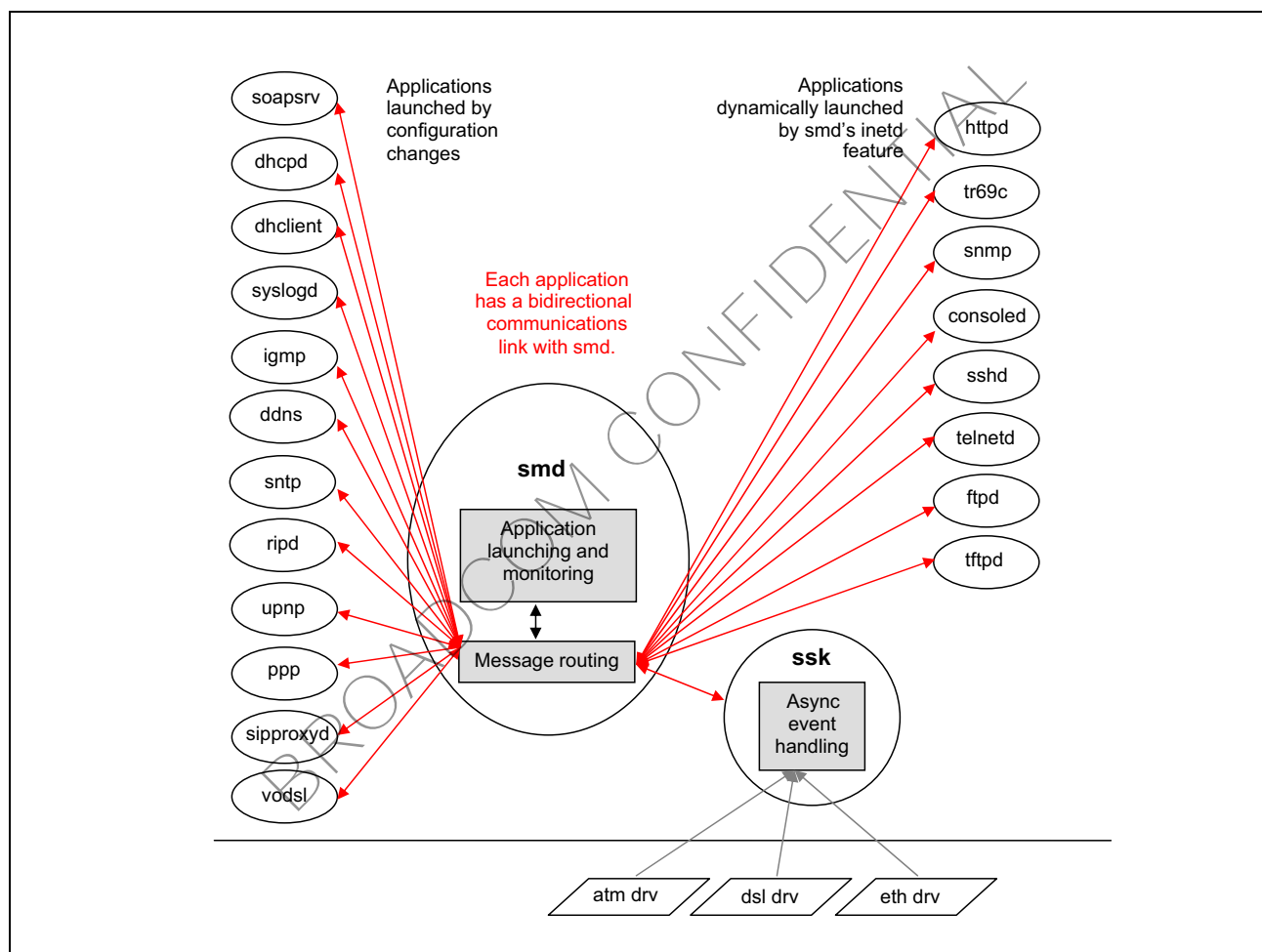
The BCA CPE Build System, which builds the bootloader, kernel, kernel device drivers, everything in userspace (including CMS and non-CMS components), and the final system image are technically not part of CMS. However, the Build System does have knowledge of CMS requirements because it needs to build CMS.

## Section 2: Overview

### Architecture

There are two ways to look at CMS. [Figure 1](#) shows CMS in terms of all the applications in the system and their relationships to each other. [Figure 2](#) shows the CMS shared libraries and APIs and how they are used by various applications.

**Figure 1: Applications View of the CMS Architecture**



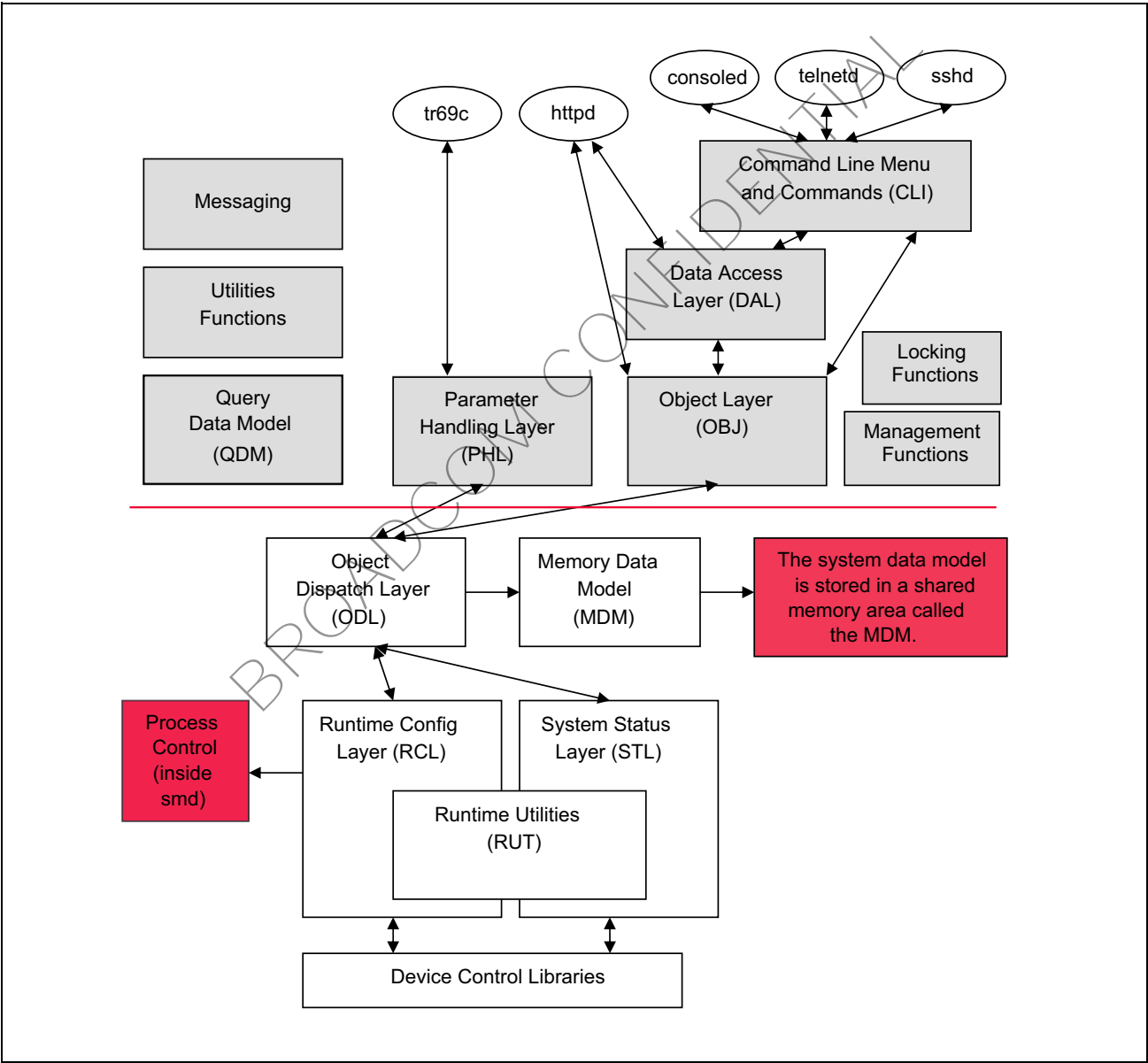
In the CMS architecture shown in [Figure 1](#), the smd and ssk applications are the central applications of the system. Smd is one of the first applications that is started when the system boots, and it manages the life cycle (start, stop, restart) of all the other CMS applications. Once a CMS-aware application launches, it creates a bidirectional communications link back to smd. (The communications link is implemented as a UNIX<sup>®</sup> domain socket.) Since all CMS-aware applications have a communications link to smd, smd serves as the message router of the CMS architecture.

Ssk is the copilot or executive assistant to smd. Ssk is the first application launched by smd on bootup. During bootup, ssk (not smd) initializes the MDM. (The MDM will be described in a later section). Once ssk has initialized the MDM, its main job is to collect asynchronous events from the system and update the MDM with the new information. For example, when the DSL link comes up, ssk finds out about this asynchronous event from the kernel and updates the MDM with the new link status.

# API Layering

Figure 2 shows the APIs and their layering in the CMS architecture.

Figure 2: API Layering in the CMS Architecture





The boxes in gray in [Figure 2](#) (Messaging, Utilities, CLI, DAL, OBJ, ODM, PHL, locking, and management) are APIs that are available for applications to call. The boxes in white (ODL, MDM, RCL, RUT, STL, and device control libraries) are considered internal API components and are not directly accessible by the applications. The red boxes (process control and configuration data) are not part of the API; they are shown to provide some context for the API boxes.

Most of these APIs are discussed in detail in the later sections of this document. This section briefly summarizes their function and provides some high level information about them.

In CMS, the two major configuration interfaces are the PHL and the OBJ. Management applications, such as tr69c, httpd, consoled, telnetd, and sshd access the configuration data and configure the system through one of these two interfaces. As [Figure 2](#) shows, tr69c accesses the PHL directly. Httpd can access the OBJ directly, but it also uses the DAL, which translates between the httpd's internal data structures and the MDM objects. Consoled, telnetd, and sshd use the OBJ interface, however, they do not access the OBJ layer directly. These applications use the CLI library, which implements all of the CLI functionality. The CLI uses the functions in the DAL and OBJ to access the configuration. The PHL and OBJ layers are very generic and should not require any further customization. Implementing a new feature most likely involves modifying the code in the DAL, CLI, and the management applications themselves.

A new API layer, called Query Data Model layer (QDM), is introduced as part of the TR-181 support. The QDM serves two purposes: to hide the differences between TR-98 and TR-181 from the caller and to enable a better way to share code between the upper layer APIs such as DAL and CLI, and the lower layer RCL/STL/RUT functions. QDM is described in [Section 11: "Query Data Model \(QDM\) Library," on page 86](#).

The locking and management functions are also used by the management applications, but they are relatively small APIs. Arrows denoting their relationship to the rest of the APIs are not shown. Typically, these APIs do not need to be modified.

The messaging and utilities APIs are also used by management applications, but arrows to them from the management applications are not shown to keep the diagram clearer. The messaging API probably will not need to be modified. The utilities API provides a wealth of useful functions, but it is quite possible that a CMS developer will want to add additional functionality in this API during development work.

The ODL and MDM contain low-level functions to manipulate the data model. They do not need to be modified by CMS developers so they will not be described further in this document.

The device control libraries contain code that controls the ATM, DSL, Ethernet, and wireless device drivers. They are described in [Section 21: "Board and Device Control Libraries," on page 151](#).

The RCL, RUT, and system status layer (STL) APIs are responsible for system configuration (RCL and RUT) and reporting of system status (STL). A large part of the development work will be done in the RCL, RUT, and STL areas of CMS. See [Section 12: "Runtime Configuration Layer," on page 87](#), [Section 13: "Runtime Utilities," on page 99](#), and [Section 14: "System Status Layer," on page 100](#) for detailed descriptions of these APIs.

Even though the APIs are divided into 15 separate boxes in the diagram, they are actually grouped into a smaller number of shared libraries. [Table 1](#) shows the grouping of APIs into libraries, and summarizes some of the information provided above.

Licensing will be discussed in a later section.

**Table 1: API Library Groupings**

<b>Library Name</b>	<b>APIs</b>	<b>License</b>	<b>Amount of Implementation/Customization Needed</b>
libcms_core.so	PHL, OBJ, Locking, Management, ODL, MDM, RCL, RUT, STL	Proprietary	RCL, RUT, and STL need customization. Others do not need to be modified.
libcms_qdm.so	ODM	Proprietary	Needs customization
libcms_dal.so	DAL	Proprietary	Needs customization.
libcms_cli.so	CLI	Proprietary	Needs customization.
libcms_msg.so	MSG	Public	No modification needed.
libcms_util.so	UTIL	Dual-link	Some customization or enhancements may be needed.
libcms_boardctl.so	Board control	Public	No modification needed.
xdsctl.so	ADSL/VDSL control	Proprietary	No modification needed.
atmctl.so	ATM control	Proprietary	No modification needed.

## Major Concepts

### Use of Broadband Forum Data Models

CMS uses the Broadband Forum-defined data models (TR-98, TR-104, TR-111, TR-140, TR-181, see [“References” on page 13](#)) as its internal data model. This makes the TR-69 client implementation very efficient because the TR-69 client does not have to translate the TR-69 request/responses from the Broadband Forum data model to the system’s internal data model and back again. However, other management applications, such as httpd, telnetd/CLI, snmp, will have to translate their configuration commands into the TR-98 data model. A large portion of that work is done in the DAL.

TR-69 and the Broadband Forum data models have been adopted by carriers around the world, so the optimized support the Broadband Forum data models is an important feature of CMS.

### Selecting the Data Model Mode

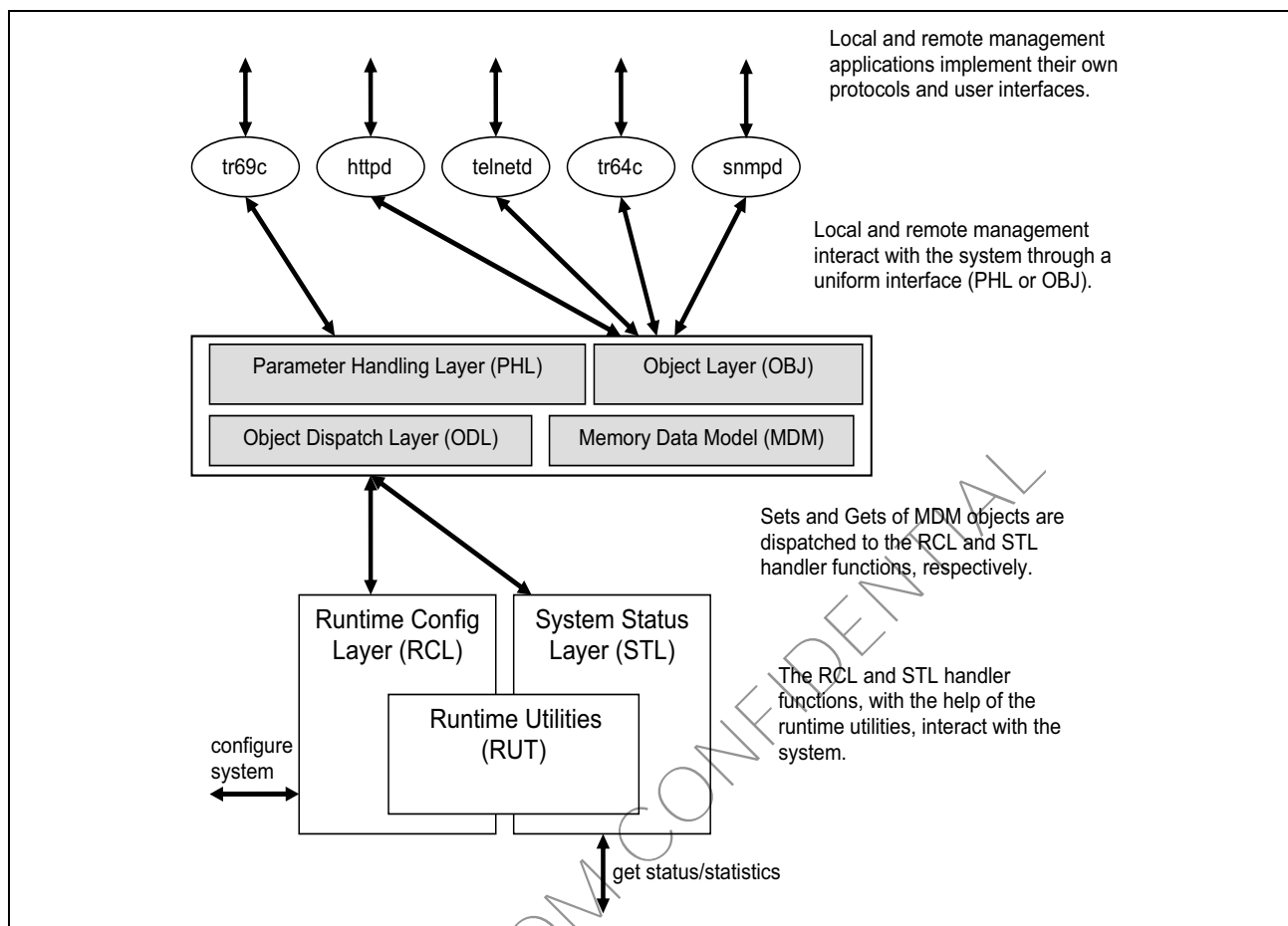
Starting with release 4.16L.02, the following four data model modes are available in CMS. Data Models are selectable from **make menuconfig**, in the Major Features Selection section.

- **Legacy TR-98 (Legacy98):** This was the original data model mode in CMS. The TR-98 data model is used as the main data model. Additional supplemental data models such TR-104 (Voice) and TR-140 (Storage) can be included, but there is no support for any TR-181 objects in the Legacy98 mode.
- **Hybrid TR-98+TR-181 (Hybrid):** This is currently the default data model mode in CMS. In the Hybrid mode, the TR-98 data model is used as the main data model. Additional supplemental data models such TR-104 and TR-140 can be included. In addition, some TR-181 objects are also included in the data model under `InternetGatewayDevice.Device`.

- **Pure TR-181 (Pure181):** If this mode is selected, TR-181 is used as the main data model. Additional supplemental data models such TR-104 and TR-140 can be included. However, there are no TR-98 data model objects in the data model. In accordance to the TR-181 specification, the root of the data model is *Device* not *InternetGatewayDevice*.
- **Data Model Detect (DM Detect):** In this mode, both the Hybrid and Pure181 data models are compiled into the image. At bootup, ssk decides which data model to initialize and use. Once the data model has been selected, a reboot is required to change the data model. Note that in this mode, you cannot select the Legacy TR98 mode. Ssk will only select between Hybrid TR98+181 or Pure181 mode. Also note that both data models cannot exist at the same time at runtime. More information about DM detect mode can be found in [Section 3: “Major Updates in CMS for TR-181,” on page 22](#).

## Separation of Management Applications from Runtime Configuration

In CMS, there is a clear and strong separation of the software code that interacts with configuration requests/responses and the software code that configures the system. Management applications, such as tr69c, httpd, telnetd/CLI, and snmpd, are responsible for implementing the protocol that accepts configuration commands and converting those configuration commands into a manipulation or modification of the data model. Once a change in the data model has been detected, a separate set of code in the RCL configures the system according to the changes to the data model (see [Figure 2 on page 16](#)). In the reverse direction, when a management application receives a request for reading a configuration element or status, it only translates the read request into a read of the data model. When a read of the data model is detected, the code in the STL gathers the information from the system and populates the data model for the management application to read. This is a very important concept of the CMS architecture.

**Figure 3: Separation of Management Application Protocol and System Configuration**

This separation of management application protocol and system configuration has a significant benefit. The code for configuring the system and gathering system information is only written once and resides in one place (in the RCL/RUT and STL, respectively). The management application code only needs to focus on implementing specific protocols (TR-69, HTTP, Telnet, SNMP) and data formats, and translating their specific protocols and data formats to the CMS API and data model. A uniform API is used by all management applications.

## Centralized Control of Starting/Stopping Applications

Smd (one of the main applications of CMS) contains enhanced functionality for managing the life cycle of applications, including starting, stopping, and restarting. Applications that choose to be controlled by smd can be started and stopped via a variety of easy-to-use CMS APIs. If configured to do so, smd also detects abnormal termination of the application and restarts it. Without smd, application developers must start and stop applications using the native Linux<sup>®</sup> and libc functions, which can be complicated and error prone. Of course, applications may choose not to be controlled by smd.

## Centralized Collection and Processing of Asynchronous Events

Ssk is the other main application in CMS. One of its main purposes is to collect asynchronous events, such as link up/down, connection status changes, dhcpd lease updates, and push that information into the MDM. In TR-181 mode, ssk also maintains the Interface Stack and propagates link and connection status up the Interface Stack.

## Using Messages to Communicate

A flexible messaging system is provided to allow applications to communicate with each other and to receive asynchronous event notifications. See [Section 15: "Messaging," on page 103](#) for details.

Applications/features/subsystems that do not want to rely on or use CMS are free to implement a messaging or communications system of their own (or use the existing messaging/communication system built into their applications/features/subsystems).

BROADCOM CONFIDENTIAL

## Section 3: Major Updates in CMS for TR-181

This section describes the major updates in CMS to support TR-181. This is a very technical section intended for readers who are already familiar with CMS. Readers who are not yet familiar with CMS should skip this section and read the other sections to learn about CMS first. This section also assumes the reader has basic familiarity with TR-181. For a complete description of TR-181, see [Reference \[6\] on page 12](#).

TR-181 Updates:

- **Upstream parameter:** In TR-181 each interface object contains a parameter called Upstream. This parameter indicates whether the interface points towards the Internet (upstream == TRUE) or points away from the Internet (towards the home network, upstream == FALSE). The upstream parameter replaces the TR-98 concepts of WAN and LAN.

- **IP.Interface object:** In TR-181, one of the most important objects is the IP.Interface object. Every WAN service is represented by an IP.Interface object. Every LAN side bridge is also represented by an IP.Interface object. The IP.Interface object replaces the TR-98 WANDevice and LANDevice objects.

Note1: In CMS, even a bridged WAN service is represented by an IP.Interface object even though technically, the bridge WAN service is not an IP layer service. The IP.Interface object of a bridged WAN service has its parameters set in the following way:

name == Linux interface name of the WAN interface (e.g., ptm0.1)

X\_BROADCOM\_COM\_Upstream == TRUE,

X\_BROADCOM\_COM\_BridgeService == TRUE,

X\_BROADCOM\_COM\_BridgeName == Linux bridge name (e.g., "br0") that this bridged WAN service is attached to.

Note2: Because TR-181 uses IP.Interface objects to represent LAN side bridge interfaces, the TR-98 interface grouping concept is an inherent part of TR-181. Therefore, in TR-181, interface grouping is always enabled.

- **Flat Hierarchy:** The TR-181 data model is very flat compared to TR-98. Hierarchy is built using the LowerLayers and Interface parameters found in many of the TR-181 objects. This allows TR-181 to easily adapt to future changes and enhancements (whereas in TR-98, interfaces and their relationships are more rigid and fixed by the structure of the data model, making it more difficult to add new interfaces and features).

Note: the extremely flexible nature of the LowerLayer and Interface parameters can also be a problem. Service providers may be tempted to set these parameters in complex, unusual, or illegal ways. Although CMS makes an effort to handle reasonable and common configuration sequences and parameter settings, it may not be able to handle all arbitrary combinations. Broadcom will need to work with service providers to define reasonable test sequences and verify that the CMS code can support them.

- **LowerLayers parameter:** The LowerLayers parameter is used to build hierarchy in the TR-181 data model. Each LowerLayers parameter typically contains a single full path to the next lower layer object. For example, if the WAN service IP.Interface uses PPP, its LowerLayers param would be Device.PPP.Interface.1. However, sometimes a LowerLayers parameter may contain multiple full paths to multiple lower layer objects. For example, if a bridge contains three Ethernet interfaces, then the management port object of the bridge would have a LowerLayers param that contains

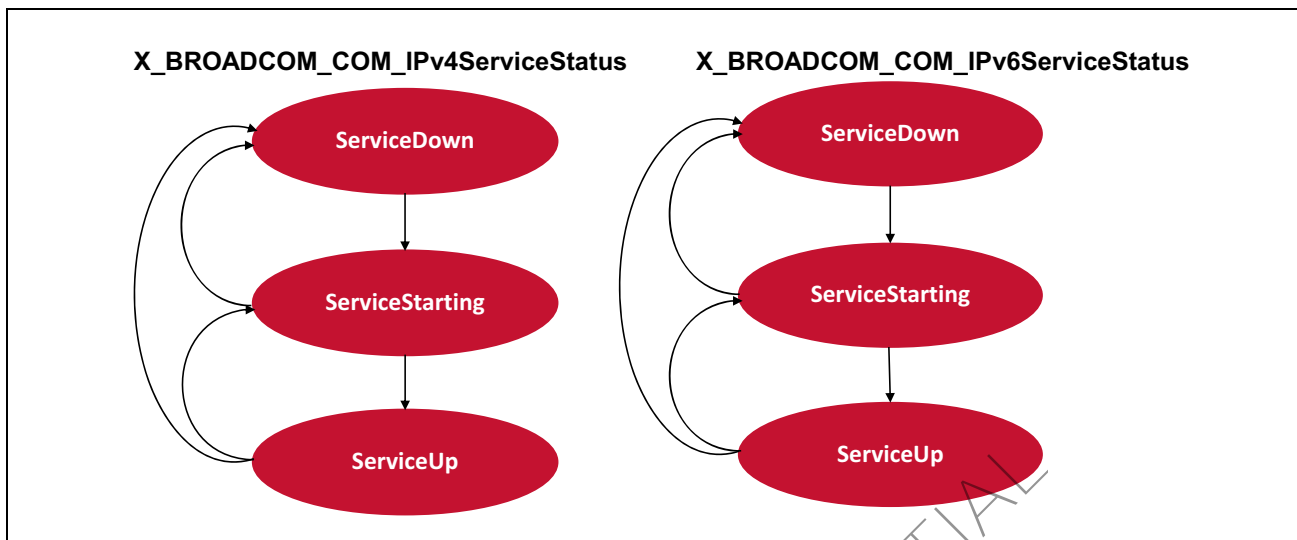
Device.Ethernet.Interface.1, Device.Ethernet.Interface.2, and Device.Ethernet.Interface.3. Refer to the TR-181 specification [Reference \[6\] on page 12](#) for a more detailed example of LowerLayers parameters and Interface Stack.

Note: For the sake of brevity, CMS documentation will sometimes refer to the LowerLayer and Interface parameters as "pointers". Actually, they are not pointers like in the C programming language. These parameters contain a full path which "points" to another object.

- **Interface Stack:** TR-181 defines an InterfaceStack object. There is one InterfaceStack object per full path in the LowerLayers parameters (but not the Interface parameters). Whenever a full path is added, deleted, or changed in any LowerLayers param, TR-181 requires that the InterfaceStack objects be updated. All of these interface stack objects are sometimes referred to as the Interface Stack table. In CMS, the interface stack table is updated in the following way:
  - In the cms-dm-tr181-\*.xml files, each interface object with a LowerLayer param has an attribute called notifySskLowerLayersChanged="true".
  - When the LowerLayer param changes, the code in mdm.c will detect that the LowerLayers parameter has changed. This code is not visible to customers since mdm.c is only released as binary. However, mdm.c will call mdm\_sendLowerLayersChangedMsg() which sends a CMS\_MSG\_INTFSTACK\_LOWERLAYERS\_CHANGED message describing what has changed to ssk. This code is visible to customers because it is in mdm\_binaryhelper.c.
  - When an object which contains a LowerLayers parameter is deleted, mdm.c calls mdm\_sendObjectDeletedMsg() to send a CMS\_MSG\_INTFSTACK\_OBJECT\_DELETED message to ssk. When ssk receives these messages, it updates the Interface Stack table accordingly.
  - When an interface is deleted or is no longer pointed to, a LOWERLAYERDOWN status is propagated up the interface stack.
- **Persistence of Interface Stack:** After ssk modifies the interface stack table, it calls cmsMgm\_saveConfigToFlash() to write the configuration file, which contains the interface stack table, to persistent flash memory. Writing the interface stack to flash memory and restoring the interface stack during reboot keeps the instance numbers in the interface stack table the same across reboots. If the interface stack table is not written to the config file, the interface stack table must be recreated during every bootup, which could lead to slightly different instance numbers in the interface stack table. This may cause service providers to complain. To avoid writing to flash too frequently, ssk will write the config file, at most, once every five seconds. See writeIntfStackToConfigLocked() in ssk2\_intfstack.c
- **Propagating status up the Interface Stack:** TR-181 requires that when an object's status changes, the new status must be propagated up the interface stack. In CMS, ssk is responsible for this task by calling intfStack\_propagateStatusByFullPathLocked(). This function is the heart of the Interface Stack status propagation logic of ssk and CMS.
- **IPv4 and IPv6 state machines in IP.Interface object:** Usually, there is an IP.Interface object at the top of every interface stack. The IP.Interface object contains a status parameter. However, this single status parameter is not able to capture all the possible combinations of states when the system/CPE supports both IPv4 and IPv6. Therefore in CMS, the IP.Interface object contains two Broadcom proprietary parameters:
  - X\_BROADCOM\_COM\_IPv4ServiceStatus
  - X\_BROADCOM\_COM\_IPv6ServiceStatus

These parameters drive independent state machines shown in [Figure 4](#).



**Figure 4: IPv4 and IPv6 State Machines in IP.Interface**

The code which implements these state machines is in:

userspace/private/libs/cms\_core/linux/device2/rut2\_ipservicecfg.c and rut2\_ipservicecfg6.c

If either the IPv4ServiceStatus or IPv6ServiceStatus is UP, then the IP.Interface.Status == UP.

- **Enable versus Status parameter:** In TR-181, most objects have an Enable and a Status parameter. The Enable parameter is a Read/Write parameter which allows the ACS or other management entity to enable or disable an object. The Status parameter is a read-only parameter which is used by the system to report the status of the object. This is an improvement over TR98, where sometimes the enable/disable control functionality and the status reporting functionality were encapsulated in a single parameter.
- **LastChange parameter:** In TR-181, every object which has a Status parameter also contains a LastChange parameter, which is the number of seconds since the object entered its current state. In order to implement the LastChange parameter, CMS also needs a X\_BROADCOM\_COM\_LastChange parameter, which contains the time when the status last changed. So the formula to report the TR181 LastChange is:

$\text{LastChange} = \text{current\_time} - \text{X\_BROADCOM\_COM\_LastChange}$

If the new status is first detected by the STL handler function, X\_BROADCOM\_COM\_LastChange is set in the STL handler function (for an example, see stl\_dev2EthernetInterfaceObject). If the new status is pushed into the MDM, the X\_BROADCOM\_COM\_LastChange can be changed in the code which sets the object or in the RCL handler function (for an example, see the IF\_STATUS\_HAS\_CHANGED\_SET\_LASTCHANGE macro in rcl\_dev2PppInterfaceObject). In all cases, LastChange should be reported using the IF\_OBJ\_NOT\_NULL\_GET\_LASTCHANGE macro in the STL handler function (see stl\_dev2EthernetInterfaceObject).

- **Autogeneration of Alias parameter:** As part of the Alias based addressing feature, TR-181 introduces a new alias parameter. The CPE is required to generate an initial value for the alias parameter when the object is created. After that, the CPE is not allowed to change the alias parameter.

Autogeneration of the Alias value is implemented in CMS using the defaultValue and autoGenerateAlias attributes in the XML data model file. If autoGenerateAlias="true", then when the object is created, its Alias value will be the defaultValue plus '-n', where n is the instance number of the object. TR-181 requires the autogenerated alias value to begin with "cpe". So in CMS, the defaultValue of autogenerated aliases are "cpe-<objname>". For examples, look at the Alias parameters in cms-dm-tr181-networking.xml.





**Note:** CMS does not yet support Alias based addressing. However, it does support the Alias parameter as a first step towards supporting Alias based addressing.

- Sharing data model independent code: When adding support for TR-181, areas in the code, which can and should be data model independent were identified. These data model independent areas can be used by both TR-98 and TR-181 code, thus increasing code sharing and reducing duplication of code. Two major areas were identified:
  - High level code which processes some data before configuring the MDM, typically found in httpd and the DAL.
  - Low level system configuration code, i.e., the RUT layer.

An example of the first type of code sharing can be found by looking at the configuration of the tr69c parameters in httpd. The main (high level) job of httpd is to present a Web based user interface to the user. The user enters the settings and clicks **Apply/Save**. The httpd code which implements all of these actions; handle\_request, do\_cgi, and cgiTr69cConfig() are all data model independent. It is only when httpd wants to push the user configuration into the MDM that data model dependent code must be executed. The data model dependent code is hidden under a Data Model switching macro called cmsDal\_setTr69cCfg(). Data Model switch macros are described in the next bullet.

An example of the second type of code sharing can be found by looking at the rutLan\_addInterfaceToBridge() function. This function is data model independent because it only wants to know three things: what is the interface to add, to which bridge, and is the interface a WAN interface. This function does not care whether the information is stored in a TR-98 data model or TR-181 data model. So if you run grep on the source code for rutLan\_addInterfaceToBridge, you can see that both TR-98 and TR-181 code call this function.

Note1: In many cases, the RUT functions would take a TR-98 object as an argument, which would make it data model dependent. These functions were often refactored to only take specific arguments so they can be data model independent and shared by both TR-98 and TR-181 code.

Note2: In many other cases, the RUT function cannot be made data model independent because they need to access the data model. In these cases, the Data Model switching macro method was used to make the calling code cleaner. For example, see rutLan\_createDhcpdCfg().

- Data Model switching macro: In cases where there must be two versions of the same function, one version for TR-98 and one version for TR-181, a macro was used to make the calling code cleaner. For an example see cmsDal\_setTr69cCfg() in cms\_dal.h.

```
CmsRet cmsDal_setTr69cCfg(const WEB_NTWK_VAR *webVar);
CmsRet cmsDal_setTr69cCfg_igd(const WEB_NTWK_VAR *webVar);
CmsRet cmsDal_setTr69cCfg_dev2(const WEB_NTWK_VAR *webVar);
#if defined(SUPPORT_DM_LEGACY98)
#define cmsDal_setTr69cCfg(w)          cmsDal_setTr69cCfg_igd((w))
#elif defined(SUPPORT_DM_HYBRID)
#define cmsDal_setTr69cCfg(w)          cmsDal_setTr69cCfg_igd((w))
#elif defined(SUPPORT_DM_PURE181)
#define cmsDal_setTr69cCfg(w)          cmsDal_setTr69cCfg_dev2((w))
#elif defined(SUPPORT_DM_DETECT)
#define cmsDal_setTr69cCfg(w)          (cmsMdm_isDataModelDevice2() ? \
                                         cmsDal_setTr69cCfg_dev2((w)) : \
                                         cmsDal_setTr69cCfg_igd((w)))
#endif
```

The calling code just calls `cmsDal_setTr69cCfg()`. Depending on the data model compile mode, the macro in the header file will convert that function to `cmsDal_setTr69cCfg_igd()` or `cmsDal_setTr69cCfg_dev2()` at compile time. However, in the DM Detect mode, we do not know what data model mode will be selected (both Hybrid and Pure181 are compiled into the image), so the macro will call a runtime function `cmsMdm_isDataModelDevice2()` to determine the current data model mode and then call the appropriate `_igd` or `_dev2` function. This macro can take an arbitrary number of arguments (including 0 arguments) and return any type of value.

This macro can be used almost anywhere in the code, however, it cannot be used when the function name is used as a function pointer. For example, in `http/cgi_main.c`, the function `cgiGetDhcpVendorId` is used as a function pointer:

```
{ "vendorid", cgiGetDhcpVendorId },
```

If `cgiGetGhcpVendorId` is defined as a macro as described above, and you are compiling in DM Detect mode, the compiler will flag this line as a syntax error. To avoid this problem, the data model switching macro cannot be used when a function name is used as a function pointer. Instead, implement the function as follows:

```
void cgiGetDhcpVendorId(int argc __attribute__((unused)),
                        char **argv __attribute__((unused)),
                        char *varValue)
{
    #if defined(SUPPORT_DM_LEGACY98)
        cgiGetDhcpVendorId_igd(varValue);
    #elif defined(SUPPORT_DM_HYBRID)
        cgiGetDhcpVendorId_igd(varValue);
    #elif defined(SUPPORT_DM_PURE181)
        cgiGetDhcpVendorId_dev2(varValue);
    #elif defined(SUPPORT_DM_DETECT)
        if (cmsMdm_isDataModelDevice2())
        {
            cgiGetDhcpVendorId_dev2(varValue);
        }
        else
        {
            cgiGetDhcpVendorId_igd(varValue);
        }
    #endif
}
```

And move the original body of `cgiGetDhcpVendorId` into `cgiGetDhcpVendorId_igd`.

- Query Data Model (QDM) API layer: Another way CMS tries to separate data model independent code and data model dependent code is by putting the data model dependent code in a separate library called the QDM. See [Section 11: "Query Data Model \(QDM\) Library," on page 86](#) for more details.
- File naming and directory structure: The TR-181 code is predominantly kept separate so that it can be clearly identified and compiled.

The following convention is used:

- In the `cms_core` library, a separate `device2` directory was added under the Linux directory. All RCL and STL handler functions for TR-181 objects are placed in `device2` and have filenames which begin with `rcl2_*.c` and `stl2_*.c`. RUT functions which are specifically for TR-181 are also placed in the `device2` directory with filename of `rut2*.c`.

- In all other directories, TR-181 specific code is in the same level as the TR-98 code, but the TR-181 code is placed in a separate file with the typical prefix plus the number 2. For example, `ssk2_*.c`, `cgi2_*.c`, `dal2_*.c`, etc.
- All TR-98 specific code and code which can be shared by TR-98 and TR-181 code remain in their current files.
- Data Model Detect (DM Detect) Mode: When this option is selected in make menuconfig (Major Feature Selection section), both the Hybrid and Pure181 data models are compiled into the image. However, when CMS first starts up, ssk follows these steps to determine which data model to use:
  - a. ssk calls `cmsMdm_initWithAcc()`
  - b. `cmsMdm_initWithAcc()` calls `oalShm_init()`, which calls `initDataModelSelection()`

If the code was compiled in a mode other than DM Detect mode, then the correct data model to use is known at compile time, so `initDataModelSelection()` returns a fixed value. However, in DM Detect mode, `initDataModelSelection` looks for and reads the `CMS_DATA_MODEL_PSP_KEY` (`CmsDMSelect`) entry in the Persistent Scratch Pad (PSP). Refer to the comments above `CMS_DATA_MODEL_PSP_KEY` in the `cms_params.h` file for more details.

Once the correct data model is chosen, ssk initializes it and follows the same steps as before.

To change the data model, an application can call `cmsUtl_toggleDataModel()`, `cmsUtl_setDataModelDevice2()`, or `cmsUtl_clearDataModel()`. See `cms_data_model_selector.h` for more details. A reboot is required for the new data model setting to take effect. How a system determines which data model it will use is outside the scope of CMS.

Note1: DM Detect mode will only allow you to select between Hybrid TR98+TR181 and Pure181 mode. You cannot select the Legacy TR-98 data model in DM Detect mode.

Note2: Only one data model can exist in the runtime system. For example, if ssk selects the Pure181 data model, the Hybrid TR98+TR181 data model cannot exist in the runtime system.

Note 3: if the image has been compiled in DM Detect mode, a CMS CLI command called `datamodel` is available for debugging. Type `datamodel -h` for usage.

Below is a list of other TR-181 related topics which are discussed elsewhere in this document:

- See [Section 4: “The Data Model,” on page 28](#) for the following:
  - A complete list of Object and Parameter attributes added (`oid`, `autoOrder`, `countPersistentForConfigFile`, `autoGenerateAlias`, `notifySskAliasChanged`, `notifySskLowerLayersChangd`).
  - The prefix attribute to avoid conflicts in valid string defines.
  - A discussion of TR-181 profile naming.
- See [“Adding and Deleting Data Model Profiles” on page 47](#) for directions on how to add profile defines to `make.common`.
- See [Section 6: “Automatic Generation of Source and Header Files from the Data Model,” on page 50](#) for a discussion of how TR-181 data model files are generated using the commands in `merge-dev2.d`

## Section 4: The Data Model

CMS uses the Broadband Forum-defined data models (TR-98, TR-104, TR-111, TR-140, and TR-181; see [“References” on page 13](#)) as its internal data model. TR-98, TR-104, and TR-111 were initially defined in a human-friendly (non-XML) format. Broadcom developers converted these specification documents into a single XML document, cms-data-model.xml, which could be parsed and converted to C data structures (see [Section 6: “Automatic Generation of Source and Header Files from the Data Model,” on page 50](#)). Broadcom proprietary parameters and objects were also added to the cms-data-model.xml file.

As part of the effort to modularize CMS, starting in release 4.14L.01, the monolithic cms-data-model.xml file has been broken into multiple XML files. This allows the definitions from various specification documents (TR-98, TR-104, TR-111, TR-140, TR-181, etc.) to be put into their own file and also separates Broadcom proprietary parameters/objects from the standard parameters/objects. If you type **ls** in the data-model directory, you will see the following files (note cms-data-model.xml is not present).

**Table 2: Data-Model Directory Files**

<b>Files</b>	<b>Comments</b>
cms-dm-tr98.xml	This file contains the entire TR-98 and TR-111 specification plus some Broadcom proprietary parameters and objects.
cms-dm-tr104-voice.xml	This file contains the entire TR-104 specification plus some Broadcom proprietary parameters and objects.
cms-dm-tr140-storage.xml	This file contains the entire TR-140 specification plus some Broadcom proprietary parameters and objects.
cms-dm-bcm-features.xml	This file contains Broadcom proprietary parameters and objects.
cms-dm-gpon.xml	This file contains the GPON data model (also known as a MIB) from the International Telecommunications Union (ITU) for controlling GPON ONT/ONU. Although the ITU is not the same as Broadband Forum, CMS's data model is flexible enough to also incorporate this data model.
cms-dm-epon.xml	This file contains the EPON data model required by the Chinese market. Although this EPON data model was not defined by the Broadband Forum, CMS's data model is flexible enough to also incorporate this data model.
cms-dm-tr181-*.xml	These files contain a portion of the TR-181 data model. In the "Hybrid" data model mode, some of these TR-181 objects are inserted into the TR-98 data model so that they can be used while preserving the existing TR-98 based code.
Makefile, merge-igd.d, merge-dev2, generate_from_dm.pl, GenObjectNode.pm, GenParamNode.pm, and Utils.pm	Control files used to generate the full/complete data model files, cms-data-model-merged.xml and cms-data-model-merged2.xml. See <a href="#">Section 6: “Automatic Generation of Source and Header Files from the Data Model,” on page 50</a> .

**Note:** All data model files must begin with "cms-dm" and end in ".xml."

Each data model (.xml) file contains three sections:

- Object and Parameter Definitions
- Valid Strings Arrays

- Profiles

The format of each of the three sections is described below.

While it is possible to edit this data model directly with a text editor, Broadcom recommends that the Data Model Designer GUI (described in [Section 5: “Data Model Designer,” on page 36](#)) be used to modify the data model to avoid introducing formatting or syntax mistakes, as these would cause the automatic generation of the header and c files to fail. The file should only be edited directly when you have become very knowledgeable about the format and syntax of the data model, or for certain operations that the GUI does not support.

If you edit the data model file with an editor, it is strongly recommended (required for Broadcom internal developers) to run the DataModelDesigner on the data model file and to save it again. This allows DataModelDesigner to catch any formatting errors and also saves the data model file in a standardized format.

## Object and Parameter Definition

The first, and by far the largest, section of the data model file is the section that defines the objects and parameters. First, an object is defined, then all the description fields of the object are added. Next, all the parameters belonging to that object are defined, with each parameter having its own set of description fields. Then the next object is defined, and so on. The following code example shows an example of an object definition from the data model file. Note that not all possible attributes for an object are shown in this example.

**Example:**

```
<object name="InternetGatewayDevice." shortObjectName="IGDObject" specSource="TR98"
profile="Baseline:1" requirements="P" supportLevel="Present" />

<description source="TRx">The top-level object for an Internet Gateway &#10;Device.
</description>
```

[Table 3](#) shows all the possible attributes of an object element.

**Table 3: Possible Object Element Attributes**

Attribute	Purpose
hideObjectFromAcs	Do not send this object or any of its parameters or any of its child objects/parameters to the ACS. <sup>a</sup>
majorVersion	The major version corresponding to this object, as defined in the Broadband Forum data model specification. <sup>a</sup>
minorVersion	The minor version corresponding to this object, as defined in the Broadband Forum data model specification. <sup>a</sup>
oid	The first non-fake object in every cms-dm-*.xml file must have an OID attribute with a number. The number should be picked from the ranges described in data-model/README-OID.txt. All subsequent objects in the cms-dm-*.xml file will have a sequentially higher OID. Specifying a fixed OID ensures that the object will have the same OID in both the TR-98 and TR-181 data models. <sup>a</sup> <b>Note:</b> Due to the current implementation, try not to use OIDs of large values. This will waste memory space.

**Table 3: Possible Object Element Attributes (Cont.)**

<b>Attribute</b>	<b>Purpose</b>
autoOrder	This attribute can be used on objects which contain an Order parameter. If set to true, the MDM will automatically set the Order parameter value according to TR-181 specifications. If an object is deleted or if the Order parameter is changed, the MDM will automatically compact and reorder the objects according to the TR-181 specification. When cmsObj_getNext is called on the object, the instances are returned in ascending value of the Order parameter instead of the instance number. Currently, this attribute is only used in the TR-181 Qos.Classification object, but it can be used on any object with the Order parameter. <sup>a</sup>
name	Generic path name of the object.
profile	Identifies the profile the object belongs to. All TR-181 profile names should begin with Device2_. e.g., the TR-181 IP.Interface object belongs to the TR-181 IPInterface:1 profile. So its profile attribute is Device2_IPInterface:1.
pruneWriteToConfigFile	Do not write this object or any of its parameters or any of its child objects/parameters to the config file. <sup>a</sup>
requirements	If this object was defined by a Broadband Forum specification, then additional requirements for or behavior of this object. Possible values are: <ul style="list-style-type: none"> <li>• P = Object is present, but management applications cannot add or delete instances of it directly.</li> <li>• PC = Management applications can add or delete instances of this object.</li> <li>• R = The system can instantiate multiple instances of this object, but the management applications cannot add or delete instances of it.</li> </ul> If the object was not defined by the Broadband Forum, then the requirements attribute is not present.
shortObjectName	This name is used when generating structure definitions and RCL/STL handler functions for this object. It must end in the word Object, e.g., IGDDDeviceInfoObject. All TR-181 objects should have a Dev2 prefix in the shortObjectName. e.g., the shortObjectName for the TR181 IP.Interface object is Dev2IpInterfaceObject.
specSource	The specification that defined this object. It can be TR98, TR181, TR111, TR104, TR140, Broadcom, or Custom.
supportLevel	Defines the system's support for this object. All objects allow the NotSupported support level. This causes the object to be compiled out. Besides the NotSupported support level, other support levels are: <ul style="list-style-type: none"> <li>• Present = This is the only possible choice for objects that have the requirement of P or does not end in .{i}.</li> <li>• Dynamic Instances = This is the only possible choice for objects that have requirements setting PC.</li> <li>• Multiple Instances = This is the only possible choice for objects that have requirements setting R.</li> </ul>

- a. This attribute is not supported in the Data Model Designer GUI, so it must be entered using a text editor and can only be seen using a text editor. However, once this attribute has been entered in the data model file, the Data Model Designer GUI will preserve it, i.e., the Data Model Designer will not lose the attribute or overwrite it.

The following code shows an example of a parameter definition from the data model file. Note that not all possible attributes for a parameter are shown in this example.

```
<parameter name="LANDeviceNumberOfEntries" type="unsignedInt" specSource="TR98"
profile="Baseline:1" requirements="R" supportLevel="ReadOnly" alwaysWriteToConfigFile="true" />
<description source="TRx">Number of instances of LANDevice.</description>
```

Table 4 shows all possible attributes of a parameter element.

**Table 4: Parameter Element Attributes**

Attribute	Purpose
name	Name of the parameter
type	The data type of the parameter.
specSource	The specification that defined this parameter.
profile	Identifies the profile the parameter belongs to. All TR-181 profile names should begin with Device2_. e.g., the TR181 IP.Interface.Ipv4Enable parameter belongs to the TR181 IPInterface:1 profile. So its profile attribute is Device2_IPInterface:1.
requirements	If this parameter was defined by a standards body such as the Broadband Forum, this attribute reflects the requirements imposed by the specification. Possible values are: <ul style="list-style-type: none"> <li>• R = This is a read-only parameter.</li> <li>• W = This is a read-write parameter.</li> </ul>
supportLevel	Defines the system's support for this parameter. All parameters allow the NotSupported support level. In addition, a parameter can have a support level of: <ul style="list-style-type: none"> <li>• ReadOnly. This is possible for parameters that have a requirement of R or W.</li> <li>• ReadWrite. This is possible for parameters that have a requirement of W.</li> </ul>
validValuesArray	Some string parameters are only allowed to have certain string values. This attribute references an array, defined in the Valid Strings Arrays section, which contains a set of allowed strings for this parameter.
maxLength	Specifies the maximum length of a string parameter (not including the terminating NULL character). This is optional.
defaultValue	Specifies the default value of this parameter. This is optional. If no default value is specified, then the parameter will have a default value of 0 or NULL, depending on the data type.
minValue	For Integer, Unsigned Integer, Long64, and Unsigned Long64, the minimum value of the parameter may be specified. This is optional.
maxValue	For Integer, Unsigned Integer, Long64, and Unsigned Long64, the maximum value of the parameter may be specified. This is optional.
mayDenyActiveNotification	The various Broadband Forum specifications allow the CPE to deny requests for active notification on this parameter, typically statistics parameters. This attribute simply reflects the specification.
denyActiveNotification	The system will deny requests for active notification on this parameter.
hideParameterFromAcs	Do not send this parameter to the ACS. <sup>a</sup>
alwaysWriteToConfigFile	Override the normal rules for deciding when to write a parameter to the config file and always write this parameter to the config file. <sup>a</sup>

**Table 4: Parameter Element Attributes (Cont.)**

<b>Attribute</b>	<b>Purpose</b>
neverWriteToConfigFile	Override the normal rules for deciding when to write a parameter to the config file and never write this parameter to the config file. <sup>a</sup>
isTr69Password	This attribute causes the parameter value to be sent back to the ACS as an empty string. This behavior is required by the various Broadband Forum specifications for parameter that contain passwords or other confidential information. <sup>a</sup>
isConfigPassword	This attribute causes the parameter value to be encoded/scrambled when it is written out to the config file. It is used for password fields. <sup>a</sup>
majorVersion	The major version corresponding to this object, as defined in the Broadband Forum data model specification. <sup>a</sup>
minorVersion	The minor version corresponding to this object, as defined in the Broadband Forum data model specification. <sup>a</sup>
transferDataBuffer	This is currently used on the DeviceLog parameter only. It tells the various CMS APIs to create only one copy of the parameter's data and pass it to the caller. Do not store any copies of the parameter inside the MDM because the data is large and will need to be refreshed the next time it is read. <sup>a</sup>
countPersistentForConfigFile	Some tables will have a mix of persistent and dynamic objects. For example, Device.IP.Interface.1.IPv4Address.1 may be dynamically acquired while Device.IP.Interface.1.IPv4Address.2 may be statically configured. The parent object will have a parameter called IPv4AddressNumberOfEntries. At runtime, IPv4AddressNumberOfEntries will be 2. However, when CMS writes out the config file, it will only write out the persistent or static IPv4Address. Therefore, in the config file, the IPv4AddressNumberOfEntries is 1. When this attribute is set to true on a parameter which has a name of xyzNumberOfEntries, CMS will count the number of persistent or static child entries in the xyz table and write out that number. <sup>a</sup> <b>Note:</b> All object instances are assumed to be persistent. To mark an object instance as dynamic/non-persistent, call cmsObj_setNonPersistentInstance.
autoGenerateAlias	TR-181 requires the CPE generates an initial value for the Alias parameter. If autoGenerateAlias="true", then when the object is created, its Alias value will be the DefaultValue plus '-n', where n is the instance number of the object. TR-181 requires the autogenerated alias value to begin with "cpe". So in CMS, the DefaultValue of autogenerated aliases are "cpe-<objname>". For examples, look at the Alias parameters in cms-dm-tr181-networking.xml. <sup>a</sup>
notifySskAliasChanged	This attribute should be set to true on all Alias parameters. When this flag is set on an Alias parameter, the MDM will detect changes to the value and send a CMS_MSG_INTFSTACK_ALIAS_CHANGED to ssk. Ssk will then update the alias fields in the Interface Stack table as required by TR-181. <sup>a</sup>
notifySskLowerLayersChanged	This attribute should be set to true on all LowerLayers parameters. When this flag is set on a LowerLayers parameter, the MDM will detect changes to the value and send a CMS_MSG_INTFSTACK_LOWERLAYERS_CHANGED to ssk. If an object containing a LowerLayers parameter is deleted, the MDM will send a CMS_MSG_INTFSTACK_OBJECT_DELETED to ssk. Ssk will then update the Interface Stack table, including propagating any new status up the Interface Stack as required by TR-181. <sup>a</sup>



- a. This attribute is not supported in the Data Model Designer GUI, so it must be entered using a text editor and can only be seen using a text editor. However, once this attribute has been entered in the data model file, the Data Model Designer GUI will preserve it, i.e., the Data Model Designer will not lose the attribute or overwrite it.

Each object and parameter element may be followed by up to three additional description elements. The first type of description element (<description source="TRx"), contains descriptions from the Broadband Forum specification of this object or parameter. The second type of description element (<description source="BROADCOM") contains notes from Broadcom engineers about the CMS implementation of a standard Broadband Forum-defined object or parameter, or, if the object or parameter is Broadcom-defined, a description of that object or parameter. The third type of description (<description source="Custom") allows customers to add their own notes to the object or parameter.

Note that in all the .xml files except for cms-dm-tr98.xml and cms-dm-tr181-beginrealroot.xml, one or more "FakeParentObject" are added at the beginning of the file. These FakeParentObjects are required by the DataModelDesigner (see [Section 5: "Data Model Designer," on page 36](#)) so that it can display the objects properly in a hierarchy. The FakeParentObjects will not be inserted in the real data model.

## Valid Strings Arrays

Some string parameters in the data model can only have a limited set of string values. This section defines the sets of valid strings for those parameters. This section is delimited with the <vsaInfo> and </vsaInfo> tags.

An example of a valid string array is shown below:

```
<validstringarray name="ATMLoopbackDiagnosticsValue" />
  <element>None</element>
  <element>Requested</element>
  <element>Complete</element>
  <element>Error_Internal</element>
  <element>Error_Other</element>
```

The name is referenced by the validValuesAttribute in the string parameters. Note that more than one string parameter can reference or use the same validstringarray. This is the case with the ATMLoopbackDiagnosticsValue array.

If a valid string array is used in a data model file, then it must be present in the <vsaInfo> section of that file. For example, the "enabledDisabledStatusValues" valid string array is used by the InternetGatewayDevice.X\_BROADCOM\_COM\_SnmpCfg.Status parameter in cms-dm-bcm-features.xml. So the "enabledDisabledStatusValues" valid string array is defined in the <vsaInfo> section of that file. If the same valid string array is used in another data model file, it must also be defined in the <vsaInfo> section of that file (refer to enabledDisabledStatusValues in cms-dm-tr98.xml). If the same valid string array name is used in multiple files, their definitions (i.e., the elements of the array) must be identical.

Each element in the valid string array is converted to a #define which begins with MDMVS\_ and the value of the valid string in all capitals. All valid string defines are written to mdm\_validstrings.h. Due to the capitalization of the valid string value, conflicts may arise. For example, there is a validStringArray called "ripSupportedModeValues" which contains three valid values: Send, Receive, and Both. Their valid string defines would be:

```
#define MDMVS_SEND "Send"
```

```
#define MDMVS_RECEIVE "Receive"
```

```
#define MDMVS_BOTH "Both"
```

However, suppose there is another validStringArray called "macFilterDirection" which contains these valid values: LAN\_TO\_WAN, WAN\_TO\_LAN, and BOTH. The valid string define for BOTH would be:

```
#define MDMVS_BOTH "BOTH"
```

However, this second define would conflict with the previous #define.

To avoid this naming conflict, one of the valid string arrays can be declared with an extra prefix attribute. For example:

```
<validstringarray name="ripdSupportedModeValues" prefix="RIPMODE" />
<element>Send</element>
<element>Receive</element>
<element>Both</element>
```

The generated valid string defines would now also contain the prefix, i.e.:

```
#define MDMVS_RIPMODE_SEND "Send"
#define MDMVS_RIPMODE_RECEIVE "Receive"
#define MDMVS_RIPMODE_BOTH "Both"
```

Now there is no conflict between

```
#define MDMVS_RIPMODE_BOTH "Both"
```

And

```
#define MDMVS_BOTH "BOTH"
```

## Profiles

The last section in the data model file lists all the profiles used in the file. This section is delimited by the <profileInfo> and </profileInfo> tags. A profile can be defined by the Broadband Forum, e.g., Baseline:1, QoS:1, or defined by a vendor using the same vendor extension mechanism that was used for object and parameter names, e.g., X\_BROADCOM\_COM\_SNMP:1, X\_BROADCOM\_COM\_ParentalControl:1. The same profile name may appear in multiple data model files.

Profiles serve two purposes. First, for the Broadband Forum-defined profiles, they specify the minimum set of objects and parameters that must be present and supported in the system for the system to claim that it supports that profile. For example, if the system reports to the Auto Configuration Server (ACS) that it supports the QoS:1 profile, then the ACS knows that it supports all of the objects and parameters that are defined for the QoS:1 profile.



**Note:** Not all QoS related objects and parameters are in the QoS profile. The profile only specifies a minimum set of objects and parameters.

The second use of profiles is to control conditional compilation. Using the `make menuconfig` system, the developer can select which features to enable in the system's code image. If a feature is not enabled, then the data model and code for that feature will not be compiled. Conditional compilation of features helps save Flash and SDRAM space. Each feature has a corresponding profile. All profiles have a `#define` associated with them that is based on the profile name. For example, the `QoS:1` profile will have a `#define DMP_QOS_1`, while the `X_BROADCOM_COM_DigitalCertificates:1` profile will have a `#define DMP_X_BROADCOM_COM_DIGITALCERTIFICATES_1`. Code for these features should be put under their appropriate `#define` directives.

Both the TR-98 and TR-181 specifications define a profile called `Baseline:1`. To avoid conflicts, all TR-181 profile names should have a `Device2_` prefix. So objects and parameters belonging to the TR-181 `Baseline:1` profile should have a profile attribute of `profile="Device2_Baseline:1"`. All the TR-181 profile names in the `<profileInfo>` section should also have the same `"Device2_"` prefix, i.e.,:

```
<profile name="Device2_Baseline:1" />
```

The `#define` that is generated for the TR-181 profiles all contain a `DEV2` prefix. So the profile `#define` for the TR-181 `Baseline:1` profile is `DMP_DEV2_BASELINE_1`. See the section [“Adding and Deleting Data Model Profiles” on page 47](#) for more information about how these profile defines are used in `make.common`.

There is one special profile called the `Unspecified` profile. All objects and parameters that do not have a profile are automatically assigned to this profile. Objects and parameters belonging to the `Unspecified` profile will be compiled into the image unless their parent objects have been compiled out.



**Note:** The Data Model Designer does not yet support the adding of new profiles. To add a new profile, use a text editor to add the profile to the data model file directly. See [Section 5: “Data Model Designer,” on page 36](#) for detailed instructions.

## Section 5: Data Model Designer

Broadcom provides a user-friendly GUI, called the Data Model Designer, for viewing and editing the data model files. Broadcom recommends that you use the Data Model Designer to view and edit the data model files as much as possible. After you have become very knowledgeable about the formatting of the data model files, you can use a text editor to edit the data model files. Also, when you need to do something that is not supported by the Data Model Designer, then you must use a text editor to edit the data model files. When using a text editor to edit the data model files, be very careful about syntax and formatting. The data model parsing tools have very limited ability to detect errors and print out user-friendly error messages.

### Data Model Designer Requirements

For customers, the Data Model Designer is distributed as a jar file in `hostTools/DataModelDesigner/DataModelDesigner.jar` (the source code to the Data Model Designer is not released to customers). Make sure you have a Java JRE on your system and that it is on your path by typing **java -version**. Java 1.5 or greater is required.

For internal designers, see [Appendix B: "Compiling the Data Model Designer," on page 162](#).

### Running the Data Model Designer

Because the Data Model Designer is written in Java, it can be run on Linux or Windows® systems.

To run the Data Model Designer on a Linux system, `cd` into the `hostTools/DataModelDesigner` directory and run the `runGUIlinux.sh` script. Starting with 4.14L.01, you must give a data-model filename as the first argument to `runGUIlinux.sh`. Examples are shown below:

- `./runGUIlinux.sh cms-dm-tr98.xml`
- `./runGUIlinux.sh cms-dm-bcm-features.xml`
- `./runGUIlinux.sh cms-data-model-merged.xml`

Note that the last example can be used to view the entire data model after it has been merged by the process described in [Section 6: "Automatic Generation of Source and Header Files from the Data Model," on page 50](#). You should not edit this file; `cms-data-model-merged.xml` is a generated file, so all your changes are lost when this file is regenerated.

To run the Data Model Designer on a Windows system, use the `runGUIWin.bat` script. The arguments are the same as in the Linux script.



**Note:** When the data model is saved on a Windows system, an extra character is placed at the end of every line. The data model generator ignores these extra characters. However, if developers are modifying the data model from a mix of Linux and Windows systems, version control systems may mark these extra characters as changes. This means, if you are making edits on a Windows system, you should run a Windows to Linux conversion utility on the data model file(s) before checking into the version control system.

When the Data Model Designer starts, a command bar is displayed above the main window and the two subwindows. The subwindow on the left displays **objects & parameters** on the title bar. The subwindow on the right displays **Valid String Arrays** on the title bar. Each of these subwindows will be described in detail below.

## Edit Modes

The Data Model Designer has four edit modes, which are selectable from the command bar at the top of the window.

### TR-editor Mode

In this mode, any object or parameter can be edited, including objects and parameters defined by the Broadband Forum. This mode should only be used to add new objects or parameters defined by the Broadband Forum or to correct a mistake in a Broadband Forum-defined object or parameter in the data model. Typically, the TR-editor mode should not be used.



**Note:** When adding objects and parameters in this mode, there is no way to enter the requirements. The requirements attribute must be added in the data model via a text editor.

### BRCM-dev Mode



**Note:** This should be used by Broadcom developers only. Customers should use Custom-dev mode only.

In this mode, Broadcom-proprietary objects and parameters can be added. The Short name and support level fields of Broadband Forum-defined objects and parameters can also be edited. (These fields can be edited in the BRCM-dev mode because they are used by the Broadcom software only; they are not part of the Broadband Forum specification.) The requirements attribute for Broadcom-proprietary objects and parameters should not be entered.

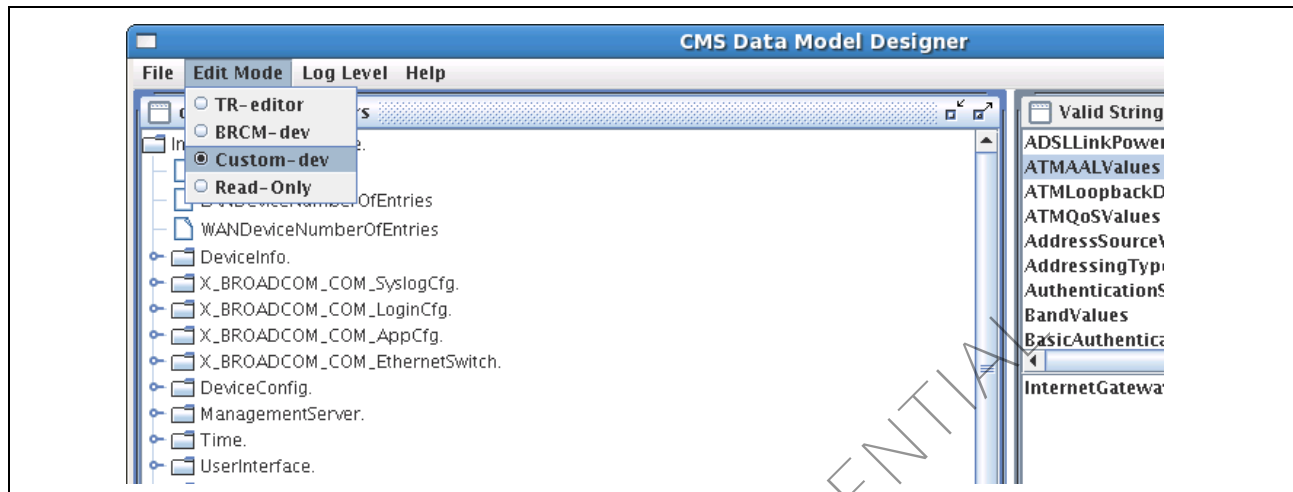
### Custom-dev Mode



**Note:** This is the mode that should be used by customers when working with the Data Model Designer.

In this mode, customer-defined objects and parameters can be added, edited, or deleted. The support level field of any other object or parameter can also be edited. The requirements attribute for customer-proprietary objects and parameters should not be entered.

**Figure 5: Setting the Edit Mode to Custom-dev**

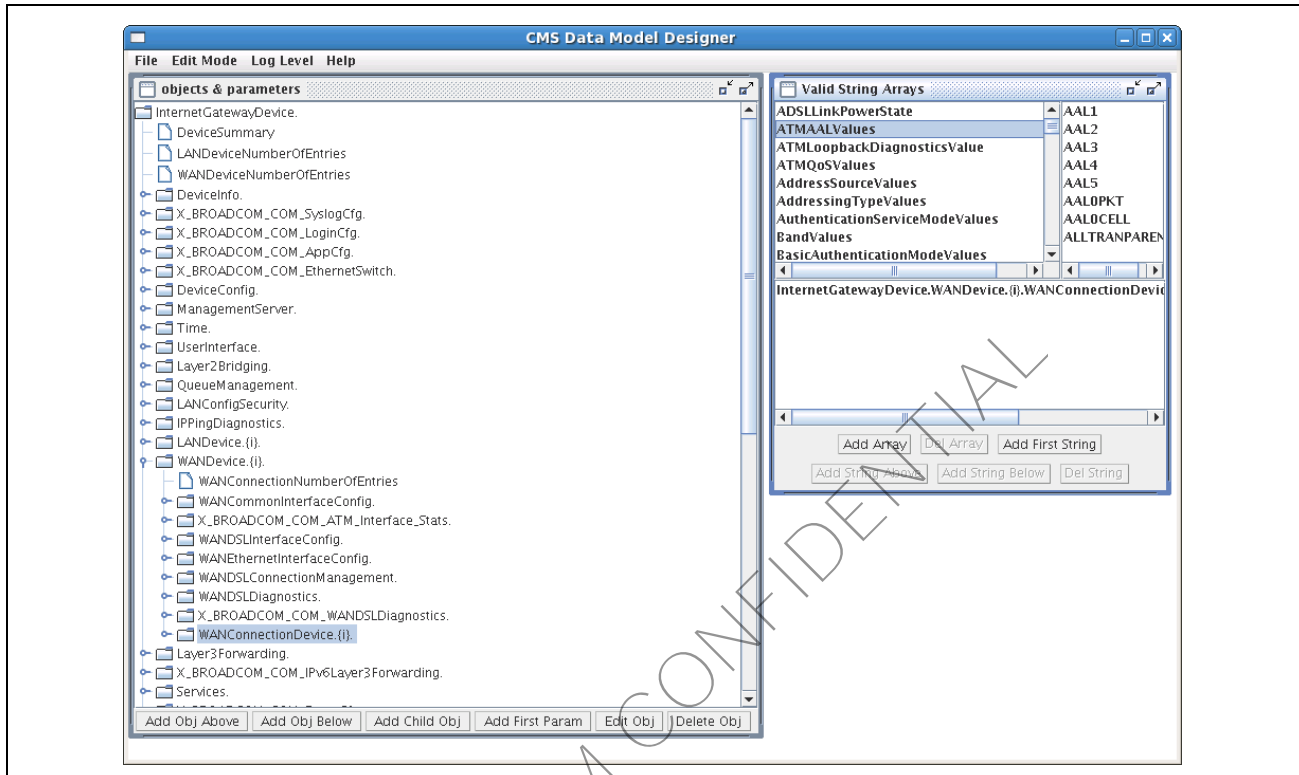


## Read-Only Mode


This mode is used to view but not edit the data model. No modifications are allowed in this mode.

## Viewing the Data Model

Figure 6: CMS Data Model Designer



To view the data model:

- Expand or contract parts of the hierarchy in the **objects & parameters** window (on the left) by clicking on the  icon (the left of the object/parameter name).

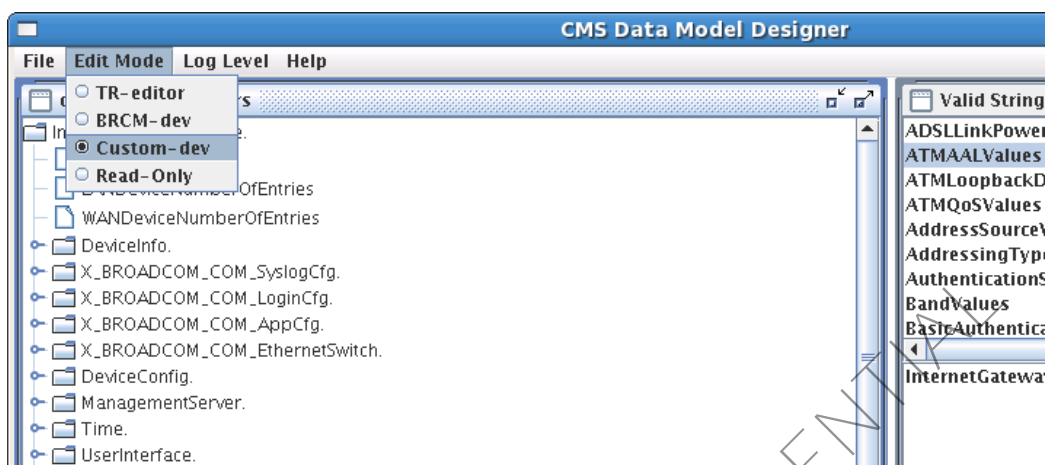
To view the description of the object or parameter:


- Click on the object or parameter name, and then click on **Edit Obj** or **Edit Param** button at the bottom of the window.

## Adding New Objects to the Data Model

To add a new object to the data model:

1. Make sure you are in an edit mode that allows you to add an object. (See “Edit Modes” on page 37.)



2. Click on an object name (objects are identified by the  icon next to them) and then click on the **Add Obj Above**, **Add Obj Below**, or **Add Child Obj** button at the bottom of the CMS Data Model Designer window.
3. For the **name** field:
  - a. If this is an object defined by the Broadband Forum, enter the name as defined by the Broadband Forum.
  - b. If this object is defined by Broadcom, enter **X\_BROADCOM\_COM\_xxx** where xxx is the name of the object.
  - c. If this object is defined by another company, the name should begin with **X\_companyname\_COM\_xxx**, or the other method defined by TR-98.

After a vendor-specific object has been defined, the parameters and objects under that object do not need to start with X\_BROADCOM\_COM, as it is clear that a parameter or object under a vendor-specific object must also be vendor specific.



**Note:** Object names must always end with a period (“.”). Type 2 object names must always end in “.{i}.”.

4. For the **short name** field, enter a short name for the object. This short name will be used as the MDMOID name and also the C data structure name for this object. The RCL and STL handler functions will also use the short name. By convention, the name should have the first letter of each word capitalized (including the first letter) and end in the word “Object”.
5. For the **Specification Source** field, select the source of the object from the list. The values that are available will depend on the Edit Mode that you are currently in. For example, the TR98, TR111, TR104, and TR106 specification source values are only available in TR-editor mode.



6. For the support level, there are up to four choices: Present, Dynamic Instances, Multiple instances, or Not Supported.
- If this object is a type 0 or type 1 object, i.e., the generic path name of the object does not end in `.{i}.`, choose the **Present** option.
  - If this object ends in `.{i}.` and management applications are allowed to add and delete instances of this object, choose the **Dynamic Instances** option.
  - If this object ends in `.{i}.` but management applications are not allowed to add and delete instances of this object, choose **Multiple Instances**. An example of a multiple instance object is:  
`InternetGatewayDevice.LANDevice.{i}.LanHostConfigManagement.Hosts.{i}.`  
The hosts object is used by the system to report the IP and MAC address of any host that it sees on the LAN subnet. It can have zero or more instances, but the instances are not created by the management application. Rather, the instances are created by the system (by `ssk`) as hosts are detected on the LAN. (Dynamic instance and multiple instance objects are both type 2 objects.)
  - If the modem software will not support this object, choose **Not Supported**. When an object is marked as Not Supported, all the object's parameters will also be not supported (regardless of what their support level is set to). However, the support level of all child objects must also be set to **Not Supported**.



**Note:** If the system will not implement code to support an object but the object is defined by the Broadband Forum, it is better to mark the object as Not Supported rather than deleting the object so that the object definition will still be retained in the data model file.

7. For the **TRx Profile** field, choose the profile that this object belongs to from the list.
- If this object was defined by the Broadband Forum, choose the profile that this object belongs to as specified by the Broadband Forum specification. If the object does not belong to a profile, then choose Unspecified.
  - If the object is defined by Broadcom or a customer, the object can belong to a Broadband Forum-defined profile, a vendor-specific profile as defined by Broadcom or a customer, or to **Unspecified**, which means the object does not belong to any profile.

Data Model Profiles serve two purposes. First, they allow the Broadband Forum to specify a minimum group of objects and parameters that belong to a profile. When a modem reports that it supports a profile, the ACS will know that the modem supports a minimum set of objects and parameters. Second, Data Model Profiles allow parts of the data model and corresponding code to be compiled out via `#ifdef DMP_XXX` preprocessor directives.



**Note:** Data Model Profiles are different than the board profiles that are set when the Broadcom modem image is compiled.

8. Next, add any comments.

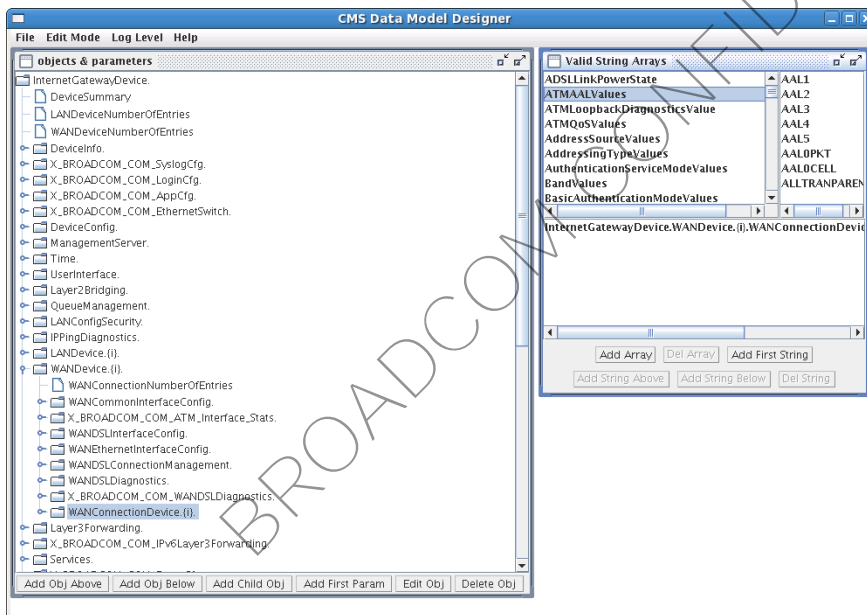
- If the object is Broadband Forum-defined, enter the description of the object from the Broadband Forum specification in the **TRx Description** field. Do not enter the description in this box if the object is defined by Broadcom or another vendor.
- In the **Broadcom Comments** field, enter any Broadcom comments for a TRx object. If the object is a Broadcom-proprietary object, enter its description in this box.
- In the **Custom Comments** field, enter any comments from the customer. These comments may apply to a Broadband Forum-defined object, a Broadcom-defined object, or a customer-defined object.

The purpose of having three separate description fields is to separate comments from different sources so that is clear where the comments came from.

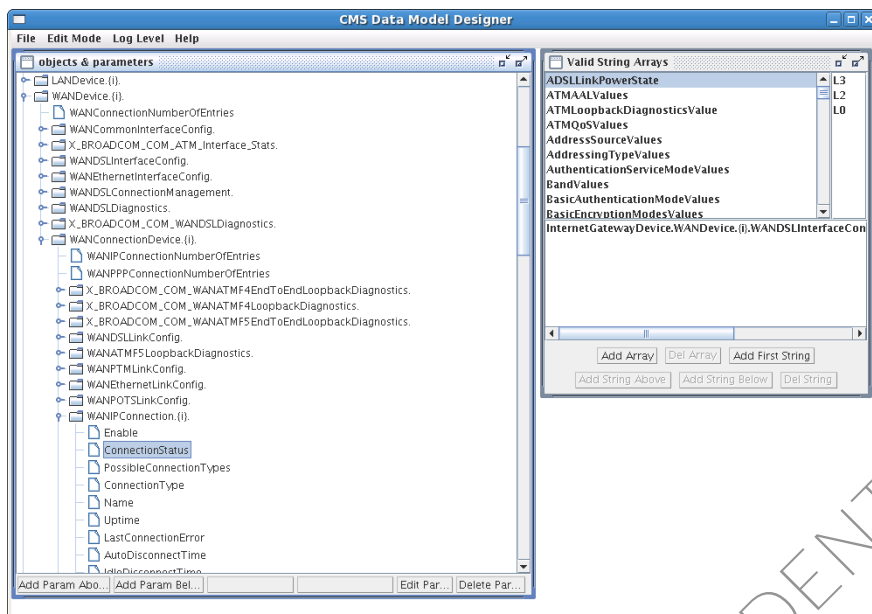
## Adding New Parameters to the Data Model

To add a new parameter:

- If you are adding the first parameter of an object, click on the object name and then click on the **Add First Param** button.



Otherwise, click on an existing parameter name and then click on the **Add Param Above** or **Add Param Below** button at the bottom of the CMS Data Model Designer window.



2. In the **Name** field, enter the name of the parameter.
3. In the **Specification Source** field, enter the source of this parameter. This is the same as the Specification Source field in the **Add new object dialog** box, which was described in the previous section.
4. For **Our Support Level**, select from the available options.



**Note:** If the parameter was defined by the Broadband Forum as read-only, then the ReadWrite support level option will not be available.

5. For the **TRx Profile**, select a profile. This is the same as the TRx Profile field in the **Add new object dialog** box, which was described in the previous section.
6. For the **parameter flags**:
  - a. To set the CPE to deny ACS requests for active notification on this parameter, click **Deny Request for Active Notification**.
  - b. If this parameter was defined by the Broadband Forum, select this option only if the Broadband Forum specification says this parameter may deny requests for active notification.
7. Choose the parameter type from the **Type of Param** list. The Broadband Forum defines six parameter types. Each parameter type specification requires its own unique set of additional information, described in [Table 5](#).

**Table 5: Broadband Forum Parameter Types and Settings**

<b>Type</b>	<b>Description</b>
Base64	This is an ASCII encoding of binary data. The maximum length of this parameter is the length of the data after Base64 encoding. Enter a length limit for the Base64 encoding and a default value. The <b>Default Value</b> should be entered in Base64, not binary. (Internally, the MDM treats Base64 as a string.)
Boolean	This is a very simple parameter type. If the parameter type is Boolean, select a default value of <b>True</b> or <b>False</b> .
DateTime	This is a special encoding of the date and time as specified by the Broadband Forum TR-69 spec. The date/time encoding is based on the Simple Object Access Protocol (SOAP) specification of date and time.
HexBinary	This is another type of ASCII encoding for binary data. Each 8-bit binary data is encoded using two ASCII characters, in the range of 0–9 and A–F. The maximum length of this parameter is the length of the data after HexBinary encoding. <b>Note:</b> This parameter type is available starting with the 4.04L.01 release. The definition differs from TR-106, which specifies the <code>maxLength</code> as the length before encoding.
Integer	This is the same as Unsigned Int parameter, except that the allowed range of values is different.
Long64	This is a signed 64-bit type. Available starting with the 4.04L.01 release.
String	First, decide whether a NULL string pointer is allowed for this parameter. Currently, the MDM does not enforce this setting, but it will in the future. Second, select <b>Restrictions on the String Value</b> and set the limit: <ul style="list-style-type: none"> <li>• If there is no limit on the length of the string value or what strings values the parameter can have, click <b>No Limits</b>.</li> <li>• To set a limit, select <b>Length Limit</b>, and then enter a maximum string length for the parameter or select <b>From predefined enumeration</b> and choose from a set of values from the list to the right of that choice. The predefined string value enumerations can be viewed, edited, and added using the <b>Valid Strings Array</b> pane on the right side of the <b>Data Model Designer</b> GUI. The Valid Strings Array pane will be described in a section below.</li> </ul> Finally, choose the default value for the string. <b>Note:</b> If the restriction on the string value is from a predefined enumeration of values, then only values from the selected enumeration will be allowed as the default value.
UnsignedLong64	This is a signed 64-bit type. Available starting with the 4.04L.01 release.
Unsigned Int	If the parameter is a (32-bit) unsigned integer and the parameter can span the entire 32-bit range of values, select the <b>Yes</b> option in the <b>Does the Unsigned Int Value have a limited range?</b> box. Next, select the <b>minimum</b> and <b>maximum</b> (inclusive) values for the parameter. Finally, enter the default value for the parameter.

The TRx, Broadcom, and Custom description fields are the same as in the **Add object dialog**.

---

## Deleting Objects or Parameters

Do not delete objects or parameters that are defined by the Broadband Forum. Instead, if a Broadband Forum-defined object or parameter is not supported, set the **support level** field to **Not Supported**. If the support level of an object is set to **Not Supported**, all of the parameters will automatically be not supported (even if their support level is still read only or read write). The support level of all the child objects must also be set to **Not Supported**.

Objects and parameters defined by Broadcom can be deleted if they are no longer used or supported.

To delete a parameter:

- Click on the parameter name and then click the **Delete Param** button at the bottom of the window.

To delete an object:

Click on the object name and then click on the **Delete Obj** button at the bottom of the window.



**Note:** When an object is deleted, its parameters and child objects and their parameters are also deleted.

---

## Editing an Object or Parameter

Editing an object or parameter is like adding an object or parameter. To edit an object or parameter:

1. Click on the object or parameter name and then click on the **Edit Obj** or **Edit Param** button.
2. Change any of the editable fields (based on the current edit mode).
3. Save your changes (see [“Saving Changes” on page 49](#)).

---

## Viewing, Editing, Adding, or Deleting a Valid String Array

To view, edit, add, or delete a valid string array, use the **Valid String Arrays** pane (on the right side of the Data Model Designer UI). The Valid String Arrays pane is further divided into three smaller panes. The pane on the upper left side displays all the valid string arrays currently defined in the Data Model. If you click on one of the valid string array names, the pane on the upper right side will display all the string values that belong to the valid string array you selected, and the pane on the bottom displays all the parameters that use the valid string array. For example, if you click on **ATMAALValues** in the pane on the upper left, you will see AAL1, AAL2, AAL3, AAL4, AAL5, AAL0PKT, AAL0CELL, and AALTRANSPARENT in the pane on the upper right. The pane on the bottom will display **InternetGatewayDevice.{i}.WANDevice.{i}.WANConnectionDevice.{i}.WANDSLLinkConfig.ATMAAL**, which is the only parameter that uses

ATMAALValues.

All valid strings are converted to a #define and placed in userspace/public/include/mdm\_validstrings.h. The individual valid strings are not separated by the valid string array they belong to. The advantage of this approach is that the same valid string can be used in multiple valid string arrays. For example, the valid string AAL5 is used in the ATMAALValues valid string array, but it can also be used in some other valid string array. However, the disadvantage of this approach is you cannot define two valid strings that only differ in their capitalization. For example, since there is already a valid string AAL5, you cannot define another aal5 string. The UI does not check for this restriction, so when defining new valid strings use a text editor to search for occurrences of that same string with different capitalizations.

## Valid String Arrays and the Broadband Forum Data Model

Broadband Forum-defined data models have a string parameter type. Some string parameters may have arbitrary values, such as DeviceInfo.ModelName. However, some string parameters may have only a limited set of string values. For example, the WANDSLLinkConfig.ATMAAL parameter may only be one of the following: AAL1, AAL2, AAL3, AAL4, AAL5, AAL0PKT, AAL0CELL, and ALLTRANSPARENT. For these types of string parameters, the Broadcom CMS Data Model allows you to define a valid string array and populate that array with the valid string values in that array. When a string parameter is defined, the name of the valid string array that this parameter must inherit values from may also be specified. The CMS will validate this parameter against the specified valid string array, and attempts to set the parameter to a value other than one that is in the specified valid string array will be rejected.

## Adding and Deleting Data Model Profiles

The Data Model Designer does not support adding or deleting Data Model Profiles. To add or delete a Data Model Profile, use a text editor and manually edit the appropriate data model file. All the Data Model Profiles are listed at the end of the file in the section starting with `<profileInfo>`.

To add a Data Model Profile:

1. Add the profile to the `<profileInfo>` section of appropriate data model file.
2. Open **data-model/generate\_from\_dm.pl** with an editor.
3. Go to the function named **output\_mdm** and create another **#include** line similar to the other **#include** lines at the top of the function, except with the name of the profile that you just added.
4. Add the profile to the CMS\_DMP\_FLAGS in **make.common**. (See subsection called "Profiles" in previous section for the relationship between a profile name and the profile `#define`). Since CMS now supports two main data models (TR-98 and TR-181) and four data model compile modes, determining when and where to add a profile flag to CMS\_DMP\_FLAGS can be complicated. Generally, there are three possible scenarios you need consider:

- a. The profile you are adding is the same in both the TR-98 and TR-181 data models. For example, Broadcom defined data model profiles and auxiliary profiles such as TR-104 and TR-140 are used in both the TR-98 and TR-181 data models. In this case, their profile defines are added to the CMS\_DMP\_FLAGS as long as the feature was enabled:

```
ifneq ($(strip $(BUILD_DDNSD)),)
CMS_DMP_FLAGS += -DDMP_X_BROADCOM_COM_DYNAMICDNS_1
endif
ifneq ($(strip $(BUILD_STORAGEESERVICE)),)
CMS_DMP_FLAGS += -DDMP_STORAGEESERVICE_1
endif
```

- b. The profile you are adding has one version in the TR-98 data model and a different version in the TR-181 data model, and their inclusion is optional (selectable via **make menuconfig**). For example, the DSL PTM feature is defined by one set of objects and profiles in TR-98, and by a different set of objects and profiles in TR-181. In addition, inclusion of the DSL PTM feature is optional. So the profile defines are added to CMS\_DMP\_FLAGS in the following way:

```
ifneq ($(strip $(BUILD_PTMWAN)),)
CMS_DMP_FLAGS += -DSUPPORT_PTM
ifneq ($(strip $(BUILD_TR98_PROFILES)),)
CMS_DMP_FLAGS += -DDMP_PTMWAN_1 -DDMP_X_BROADCOM_COM_PTMWAN_1
endif
ifneq ($(strip $(BUILD_PURE181_PROFILES)),)
CMS_DMP_FLAGS += -DDMP_DEVICE2_PTMLINK_1 -DDMP_DEVICE2_X_BROADCOM_COM_PTMLINK_1
endif
endif
```

In this example, if the DSL PTM feature is enabled (BUILD\_PTMWAN is not empty), then the SUPPORT\_PTM flag is added to CMS\_DMP\_FLAGS regardless of which data model was selected. Any flag that begins with SUPPORT\_ means it surrounds code which is data model independent. In the next if block, if the selected data model compile mode uses the TR-98 data model as the main data model (Legacy98, Hybrid, and DM Detect), then the TR-98 PTM profile flags (DDMP\_PTMWAN\_1 -DDMP\_X\_BROADCOM\_COM\_PTMWAN\_1) are added to CMS\_DMP\_FLAGS. In the next if block, if the selected data model compile mode uses the TR-

181 data model as the main data model (Pure181 and DM Select), then the TR-181 PTM profile flags (DDMP\_DEVICE2\_PTMLINK\_1 -DDMP\_DEVICE2\_X\_BROADCOM\_COM\_PTMLINK\_1) are added to CMS\_DMP\_FLAGS. Note that testing for BUILD\_TR98\_PROFILES and BUILD\_TR181\_PROFILES are done with two separate ifneq statements. If, else, end is not used because these two options are not mutually exclusive. In the DM Detect mode, both BUILD\_TR98\_PROFILES and BUILD\_TR181\_PROFILES are defined.

- c. The profile you are adding has one version in the TR-98 data model and a different version in the TR-181 data model, and their inclusion is mandatory (because they are such a fundamental feature that they cannot be excluded). Examples of these types of profiles are the Baseline profile, Device2\_IPInterface profile, etc. These profiles are defined in one of three places, as the code snippet from make.common (around line 630) shows:

```
#
# ==> These TR98 profiles are always defined when the root
#       of the data model is InternetGatewayDevice, i.e. when we are in
#       Legacy TR98 and Hybrid98+181 modes.
#
ifneq ($(strip $(BUILD_TR98_PROFILES)),)
CMS_DMP_FLAGS += -DDMP_BASELINE_1 -DDMP_X_BROADCOM_COM_BASELINE_1
endif

#
# ==> This is a small subset of the TR181 profiles which are always
#       defined in both Hybrid98+181 and Pure TR181 modes.
#
ifneq ($(strip $(BUILD_HYBRID181_PROFILES)),)
CMS_DMP_FLAGS += -DDMP_DEVICE2_BASELINE_1 -DDMP_DEVICE2_X_BROADCOM_COM_BASELINE_1
CMS_DMP_FLAGS += -DDMP_DEVICE2_IPINTERFACE_1 -DDMP_DEVICE2_IPINTERFACE_2
endif

#
# ==> These TR181 profiles are always defined in Pure TR181 mode (but
#       not in Hybrid98+181 mode)
#
ifneq ($(strip $(BUILD_PURE181_PROFILES)),)
CMS_DMP_FLAGS += -DDMP_DEVICE2_BRIDGE_1
endif
```

If the profile you are adding is mandatory to the TR-98 data model, then it should be added to the first if block.

If the profile you are adding is a TR-181 profile and it is a mandatory component of the Hybrid TR98+TR181 data model, then it should be added to the second if block. For example, in Hybrid TR98+181, we need the IP.Interface object from TR181, so the DMP\_DEVICE2\_IPINTERFACE\_1 flag is added to CMS\_DMP\_FLAGS in the second if block.

If the profile you are adding is a TR-181 profile and it is a mandatory component of the Pure181 data model (but not needed in Hybrid), then it should be added in the third if block. Note that if the Pure181 data model mode is selected, all the TR-181 profiles from the second block will also be included in the CMS\_DMP\_FLAGS. Also note that if the DM Select data model mode is chosen, all the profile flags in all three blocks will be added to the CMS\_DMP\_FLAGS.

To delete a Data Model Profile, follow the steps above for adding a profile, except do the opposite. i.e., delete the profile name from cms-data-model.xml, generate\_from\_dm.pl, and make.common.



**Note:** Before deleting a profile, verify that no objects or parameters belong to this profile.



## **Saving Changes**

After making changes to the data model, it must be saved before exiting the Data Model Designer. To do this:

- Click on **File** at the top left side of the window and then click **Save**.

The new data model will be written out to the appropriate data model file in the data-model directory.

BROADCOM CONFIDENTIAL

## Section 6: Automatic Generation of Source and Header Files from the Data Model

In release 4.14L.01, the monolithic cms-data-model.xml file has been broken into multiple data-model files (cms-dm-\*.xml). These individual data model files are then reassembled using the commands in the merge-igd.d and merge-dev2.d directories. The commands in the merge-igd.d directory create the cms-data-model-merged.xml file, which is used in the Legacy98 and Hybrid data model modes. The commands in the merge-dev2.d directory create the cms-data-model-merged2.xml file, which is used in the Pure181 data model mode. (In DM Detect mode, both cms-data-model-merged.xml and cms-data-model-merged2.xml are used).

Figure 7 on page 50 shows the components of the automatic file generation.

**Figure 7: Automatic File Generation Components**

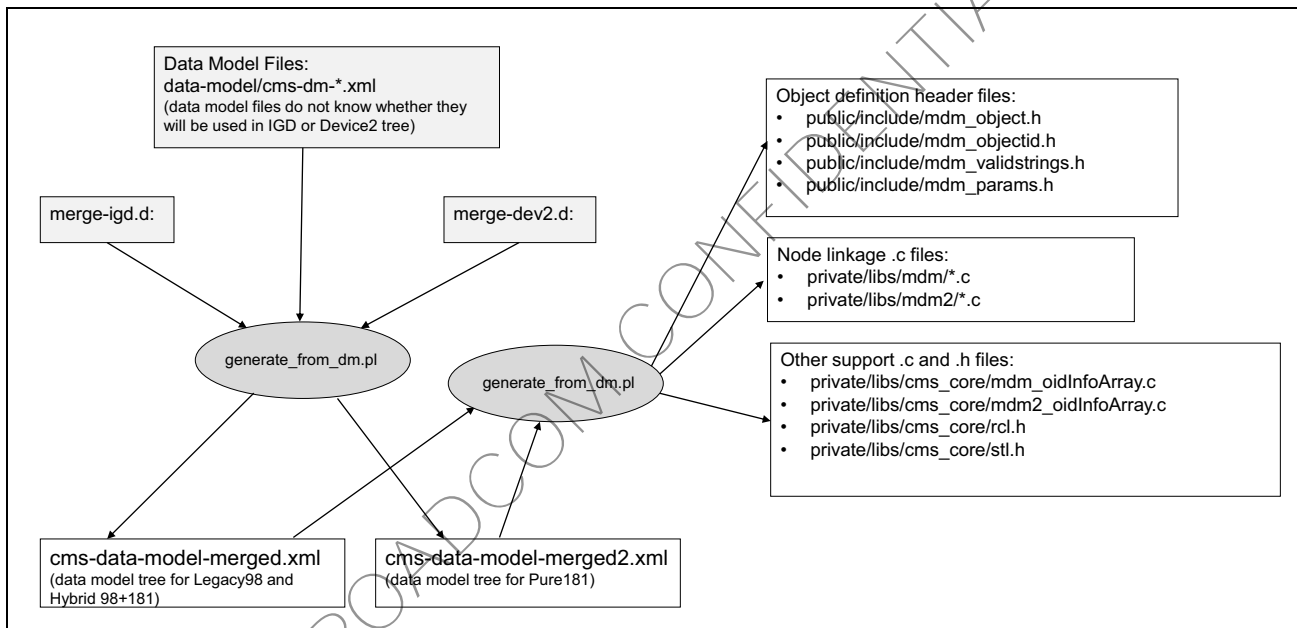


Table 6 lists the files that are generated and the purpose of the file.

**Table 6: Auto-generated Source and Header Files**

File Name	Purpose
data-model/cms-data-model-merged.xml	Combined data model file, used when main data model is TR-98. Replaces previous cms-data-model.xml file.
data-model/cms-data-model-merged2.xml	Combined data model file used when the main data model is TR-181.
userspace/public/include/mdm_objectid.h	Numeric #define values for each object in the data model.
userspace/public/include/mdm_object.h	C data structures defining the parameters that belong to each object.
userspace/public/include/mdm_validstrings.h	#define directives for all possible valid strings.

**Table 6: Auto-generated Source and Header Files (Cont.)**

<b>File Name</b>	<b>Purpose</b>
userspace/private/libs/mdm/*.c	These files combine into a single shared library (libmdm.so) which defines the relationship of the objects and parameters in the data model.
userspace/private/libs/mdm2/*.c	These files combine into a single shared library (libmdm2.so), which defines the relationship of the objects and parameters in the data model when the main data model is TR-181.
userspace/private/libs/cms_core/mdm_oldInfoArray	This file maps an MDM object ID to its generic full path name and its RCL and STL handler functions when the main data model is TR-98.
userspace/private/libs/cms_core/mdm2_oidInfoArray	This file maps a MDM object ID to its generic full path name and its RCL and STL handler functions when the main data model is TR-181.
userspace/private/libs/cms_core/rcl.h & stl.h	Function prototypes for the RCL and STL handler functions.

When a make is run from the top level directory, the build system detects changes to the data-model files and regenerates all the files as needed. If you do not want to do a complete make from the top, you can also change directory into the data-model directory and type:

```
make
```

Data model file generation does not need the Profile name (e.g., 963268GW.)

The generator scripts consists of the following perl scripts:

- generate\_from\_dm.pl, which is the main script.
- GenObjectNode.pm, which contains helper functions for generating object nodes.
- GenParamNode.pm, which contains helper functions for generating parameter nodes.

## Merge Commands

Files in merge-igd.d and merge-dev2.d contain commands which tell the generate\_from\_dm.pl script how to reassemble the individual data model pieces into cms-data-model-merged.xml and cms-data-model-merged2.xml.

Files are executed in lexicographical order. File names should begin with a four digit number starting at 1000. First read merge-igd.d/0000-README.txt and then merge-dev2.d/0000-README.txt for reserved ranges and file name/numbering conventions.

Each file may contain one or more commands, although in most cases, it makes sense to have only one command per file (more modular). The following commands are supported:

- **base:** This must be the first command in the first file in each of the merge directories. The base command specifies the main data model file. Syntax:  
base <data model file>

**Example:**

```
base cms-dm-tr98.xml
```

- **addObjBelowObj:** Add new objects below the target object at the same level (after any children objects of the target object). Syntax:

```
addObjBelowObj <data model file> <target object>
```

**Example:**

```
addObjBelowObj cms-dm-bpaapi.xml InternetGatewayDevice.DeviceInfo.
```

- **addObjAboveObj:** Add new objects above the target object. Syntax:

```
addObjAboveObj <data model file> <target object>
```

**Example:**

```
addObjAboveObj cms-dm-bcm-appcfgstub.xml InternetGatewayDevice.DeviceConfig.
```

- **addFirstChildObjToObj:** Add the new objects as the first child of the target object. Syntax:

```
addFirstChildObjToObj <data model file> <target object>
```

**Example:**

```
addFirstChildObjToObj cms-dm-bcm-baseapps.xml InternetGatewayDevice.X_BROADCOM_COM.AppCfg.
```

- **addLastChildObjToObj:** Add the new objects as the last child of the target object. Syntax:

```
addLastChildObjToObj <data model file> <target object>
```

**Example:**

```
addLastChildObjToObj cms-dm-bcm-features.xml InternetGatewayDevice.
```

- **deleteObj:** Delete the target object and all of its children. Syntax:

```
deleteObj <target object>
```

**Example:**

```
deleteObj InternetGatewayDevice.Device.DeviceInfo.
```

In the merge-igd.d and merge-dev2.d directories, a variety of commands are used. It may be puzzling why sometimes AddObjBelowObj is used instead of AddObjAboveObj, or why addFirstChildObjToObj is used instead of addLastChildObjToObj. This variety of commands is intended to exercise and test the generate\_from\_dm.pl script. Generally speaking, the command that is used is not very important. However, you should select target objects that are unlikely to be deleted by customers, i.e., choose target objects which are commonly used.

The generate\_from\_dm.pl script also supports adding an object to one place in the TR-98 data model (in merge-igd.d) and to a different place and hierarchical level in the TR-181 data model. This is done in the case of cms-dm-bcm-multicastigmpsnoop.xml.

## Generating a Report of Supported Objects, Parameters, and Profiles

To generate a report of all supported objects, parameters, and profiles go into the data-model directory and type:

```
make report
```

The following four files will be generated in the docs/HOWTO directory:

<b>CMS-supported-parameters-report.txt</b>	Report of all objects and profiles supported in cms-data-model-merged.txt.
<b>CMS-supported-parameters-report2.txt</b>	Report of all objects and profiles supported in cms-data-model-merged2.txt.
<b>CMS-supported-profiles-report.txt</b>	Report of which parameters in the TR-98 profiles listed in data-model/report_profiles_list.txt are supported. See <a href="#">Supported Profiles Report Considerations</a> .
<b>CMS-supported-profiles-report2.txt</b>	Report of which parameters in the TR-181 profiles listed in data-model/report_profiles_list2.txt are supported. See <a href="#">Supported Profiles Report Considerations</a> .



**Note:** Even though an object or parameter is listed as supported in these reports, it may not be actually compiled into the software image. If the object or parameter is part of an optional feature, that feature must be selected/enabled via make menuconfig.

### Supported Profiles Report Considerations

Typically, a parameter is not part of any profile, or is included in a single profile. In these simple cases, the profile to which a parameter belongs to can be specified using the profile attribute in the parameter line in the cms-dm-\*.xml file. Sometimes, a parameter can belong to two or more profiles. In this case, the profiles that a parameter belongs to must be specified in the Broadcom description section of the param using the REPORT\_PROFILE: keyword. For example, the InternetGatewayDevice.Time.NTPServer1 parameter belongs to both the Time:1 and Time:2 profiles. The description element for the NTPServer1 parameter looks like this:

```
<description source="BROADCOM">REPORT_PROFILE: Time:1, Time:2</description>
```

The description field may contain other comments, but the REPORT\_PROFILE: portion should be the last part of the field. When generating the supported-profiles-report, if a parameter has a REPORT\_PROFILE: keyword in the Broadcom description element, then the profiles listed after the REPORT\_PROFILE: keyword are used and the profile attribute is ignored. Otherwise, the profile attribute is used. (When generating header files and C code, only the profile attribute is used; the REPORT\_PROFILE: keyword is ignored.) In the supported-profiles-report, an object is listed if one or more of its parameters which belong to a profile is supported. However, the report generation tool does not report the detailed information about objects as required by the TR-181 spec (present, create & delete, create but optional delete, delete but optional create). Typically, CMS will support the appropriate creation and deletion actions for objects.

## Section 7: System Bootup

This section describes what happens when the CPE system boots. The information provided in this section will provide some context for [Section 19: "Adding a New Application or Command," on page 134](#) on how to add a new application.

Note that in 4.14L.01, the system boot sequence has changed to make it more modular. This section describes what happens during bootup on a system running the 4.14L.01 release.

### CFE and Kernel Boot

When the CPE board powers up, the MIPS processor first executes the bootloader, called the CFE. The CFE performs some basic checks on the state of the board, initializes some of the hardware, loads the kernel image, and starts execution of the kernel image. The kernel then fully initializes the board and its devices.

### init process

After the kernel has finished booting, it creates the first userspace process called "init". Init reads from /etc/inittab for instructions on what other processes to launch.

### /etc/inittab

The inittab file can be found in the source tree at targets/fs.src/etc/inittab. There are three important lines to notice in this file, described below.

#### ::sysinit:/bin/sh -l -c "bcm\_boot\_launcher start"

This line tells init to run the bcm\_boot\_launcher app with the "start" argument. Bcm\_boot\_launcher executes all scripts starting with the letter "S" in /etc/rc3.d/ in lexicographical order. For example, in the standard 963268GW build, it executes the following scripts in the order listed below:

- S25mount-fs
- S35system-config
- S45bcm-base-drivers
- S64save-dmesg
- S65smd

Customers are able to add their own scripts into the file system to customize their bootup actions.

These files are actually symlinks to a file without the Snn in /etc/init.d. For example, S25mount-fs is a symlink to mount-fs.sh in /etc/init.d.

The numbers used after the letter "S" should be limited to 10–99. If you really want to use a number less than 10, create the file as S09my-script and not S9my-script.

## **::respawn:-/bin/sh -l -c consoled**

This line tells init to launch the consoled process, which is the application that prompts you for the username and password on the serial console. Once logged in, you are in the CMS command line interface (CLI) as indicated by the ">" prompt.

Many developers want to go immediately to the busybox shell. To save some typing, you can copy the supplied/standard inittab file to inittab.custom and modify the above line to ::respawn:-/bin/sh and rebuild the image. If the build system detects the presence of the inittab.custom file in targets/fs.src/etc, it uses that .custom file as the inittab file. So now when the system boots, the serial console will go to the busybox shell immediately. In this mode, you can still get to the CMS CLI by typing **consoled**.

## **::shutdown:/bin/sh -l -c "bcm\_boot\_launcher stop"**

This line tells init to run the bcm\_boot\_launcher app with the stop argument when the reboot command is executed. Bcm\_boot\_launcher executes all scripts starting with the letter "K" in /etc/rc3.d in lexicographical order. For example, in the standard 963268GW build, it executes the following scripts in the order listed below:

- K25smd
- K95mount-fs

---

## **/etc/profile**

Unlike previous releases, in 4.14L.01, /etc/profile does not perform any bootup actions. It only initializes the environment for /bin/sh. All startup actions have been broken into smaller scripts in /etc/rc3.d (which are symlinks to scripts in /etc/init.d).

---

## Smd

In 4.14L.01, `smd` is launched by `bcm_boot_launcher` during `S65smd`. In CMS, `smd` is responsible for managing the lifecycle of all CMS applications. Once `smd` is started, it launches other applications in two stages: `stage1` and `stage2`. Each of these stages is described below. The code is in `userspace/private/apps/smd/linux/oal_event.c`.

### Stage 1

During stage 1, CMS and the MDM are still in a very primitive (almost nonexistent) state. Most applications should not be launched during stage 1. Currently, only the `ssk` application is launched during stage 1.

`Ssk` is the copilot/executive assistant to `smd`. During the bootup sequence, `ssk` has the job of initializing the MDM. After `ssk` has finished initializing the MDM, it sends a `CMS_MSG_MDM_INITIALIZED` message to `smd`. When `smd` receives this message, it begins stage 2 of application launching. (The `CMS_MSG_MDM_INITIALIZED` message also causes `S65smd` to return a success value, so that `bcm_boot_launcher` can continue executing the other Snn startup scripts.)

### MDM Initialization

MDM initialization is done by `ssk` when it calls `cmsMdm_initWithAcc()`:

- The skeleton data model (`libmdm.so`) is loaded into the MDM.
- The configuration file is loaded from persistent flash memory. The content of the configuration file is used to modify and expand the data model in the MDM. If a configuration file is not found in persistent flash memory, then the default config file at `/etc/default.cfg` is used. If `/etc/default.cfg` is not present, then the MDM remains in its skeleton state.
- `Mdm_adjustForHardware()` is called to make any final adjustments to the MDM so that it matches what is actually on the system.
- The RCL handler function for each object instance in the MDM is called. This process is referred to as object activation (refer to `mdm_activateObjects()` in `userspace/private/libs/cms_core/mdm_init.c`). Object activation configures the system to the state described in the MDM.

During object activation, the RCL handler functions may launch applications. For example, if the configuration specifies that the CPE should run a DHCP server on the LAN side, then the RCL handler function for the DHCP server sends a message to `smd` asking it to launch the DHCP server. See [“Starting the Application” on page 142](#) for the various ways that an application can be launched.

### Stage 2

If a CMS application was not launched as part of the object activation step during MDM initialization, it can set the `EIF_LAUNCH_ON_BOOT` flag in its CMS Entity Info entry (see [Section 19: “Adding a New Application or Command,” on page 134](#)). `Smd` launches the application after MDM initialization is complete.



## Inetd Style Dynamic Application Launching

During the stage 2 application launching, `smd` opens server sockets for all server applications and starts monitoring those sockets for client requests. When a client request is detected on the server socket, the corresponding server application is launched to handle the request. When the server application has launched, `smd` stops monitoring the server socket. All future client requests will be handled by the server application. After a certain amount of idle time, the server application may decide to exit. `Smd` will detect the exit of the server application and start monitoring its server socket again for new client requests. In CMS, this service is called the Dynamic Launch Service (DLS). In the UNIX/Linux world, this service is known as `inetd`. The Dynamic Launch Service reduces system DRAM usage by not launching a server application until it is accessed by a client.

BROADCOM CONFIDENTIAL

## Section 8: HTTPD

---

### Introduction

This section outlines key pieces of code demonstrating how to create custom web pages for the CMS WebUI.

---

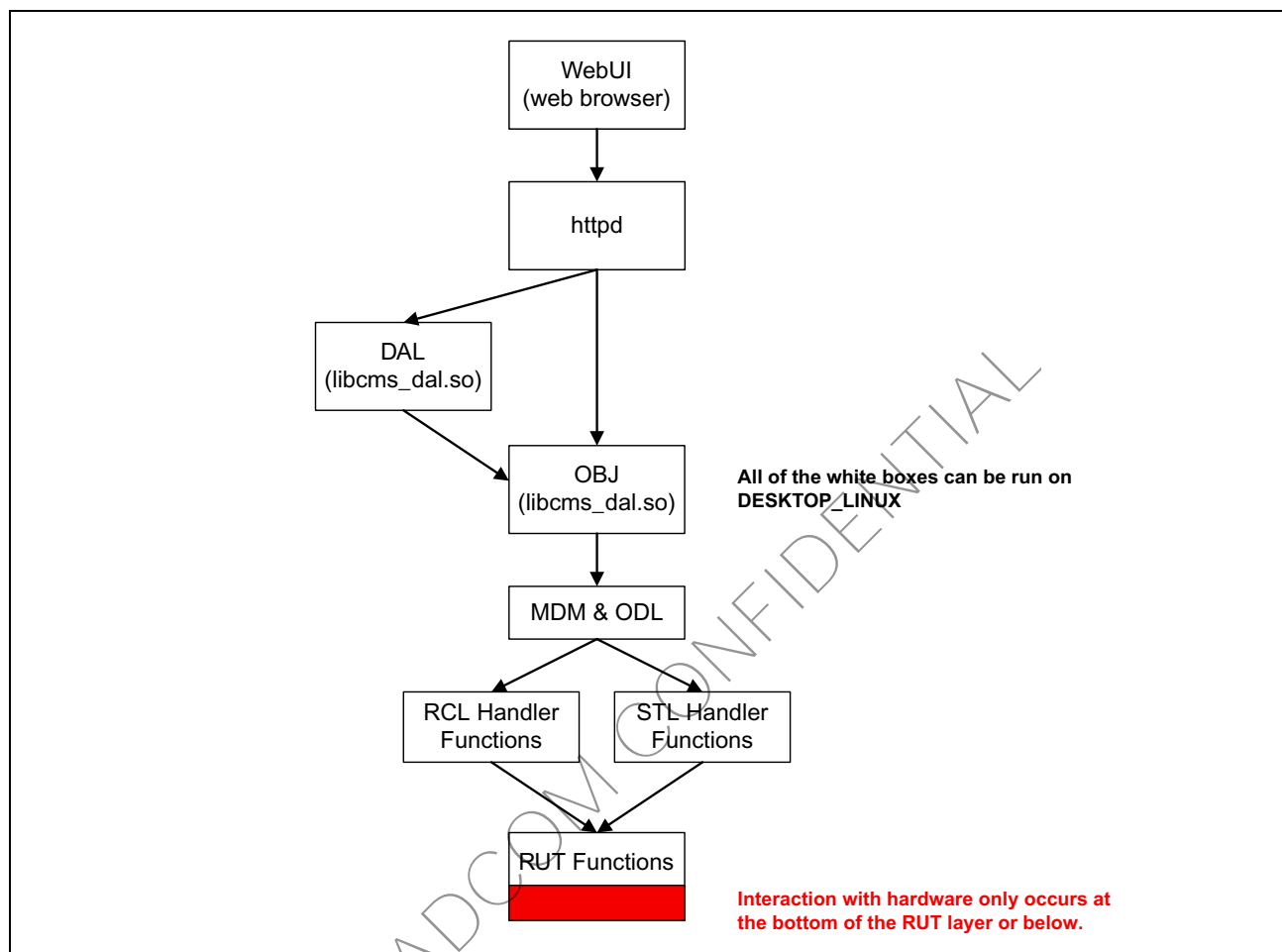
### General Overview of httpd

When developing applications on the WebUI, it is very convenient to use the DESKTOP\_LINUX feature (ask your Broadcom field applications engineer for the latest document describing the Desktop Linux feature). Most of the code in the WebUI does not require the actual CPE hardware. When doing reads from device drivers or configuring device drivers, the STL and RCL handler functions can be stubbed out appropriately with `#ifdef DESKTOP_LINUX` to return some fake values or success code so that the WebUI thinks the read or write occurred. Once all the WebUI code is working, then the software can be run on the actual CPE hardware to finish the debugging process.

BROADCOM CONFIDENTIAL

Figure 8 shows how httpd fits into the CMS architecture and the parts that have interactions with the real hardware.

**Figure 8: Functional Block Overview**



To see the debug messages about what is happening, from the console prompt, type **loglevel set httpd Debug**. All httpd actions begin at the `handle_request()` command, so start looking there to trace the actions.

In `handle_request()`, look for the for loop:

```
for (handler = &mime_handlers[0]; handler->pattern; handler++) {
    if (match(handler->pattern, filename)
```

This finds the appropriate handler function based on the requested filename suffix. Refer to `mime_handlers` in `basic.c`. The most important handlers are `do_ej()`, `do_cmd_cgi()`, and `do_cgi()`.

---

## Displaying a Page

Web pages can be roughly divided up into two categories: those that display information about the system for the user and those that collect information from the user and send it back to httpd. This section will cover the first category. The next section will cover the second category.

Note that even a page designed to gather information from the user may still need to display some information from the system first; the user inputs their configuration data, clicks on **Apply/Save**, the configuration data is sent back to httpd, httpd processes the data and sends another page (containing updated information) back to the user. Therefore, it is important to first understand how web pages are sent out to the user.

## Page From HTML Directory

When the web browser requests a page that ends in .html, the `handle_request()` function will find the `do_ej()` mime handler to process this page.

The `do_ej()` handler first opens the requested file in the html directory.

Then `do_ej()` calls `ejRefreshGlbWebVar()` with the name of the requested file. Based on the name of the file, `ejRefreshGlbWebVar()` will call various other functions that will read the system information and place that information in a global data structure called `glbWebVar` (declared in `cgi_main.c` and defined in `cms_dal.h`).

Finally, `do_ej()` will send the file to the web browser character-by-character, while at the same time, the `do_ej()` function will scan the html file looking for the special tags `<%ejGet` or `<%ejGetOther`. If these special tags are detected, functions are called to get values from the system, and those values are sent out in the web page instead of the `<%ejGet` or `<%ejGetOther` tags. The most commonly used tags are `<%ejGet` and `<%ejGetOther`. Refer to `ej_handlers[]` for a list of all available tag names.

A complete tag looks like this:

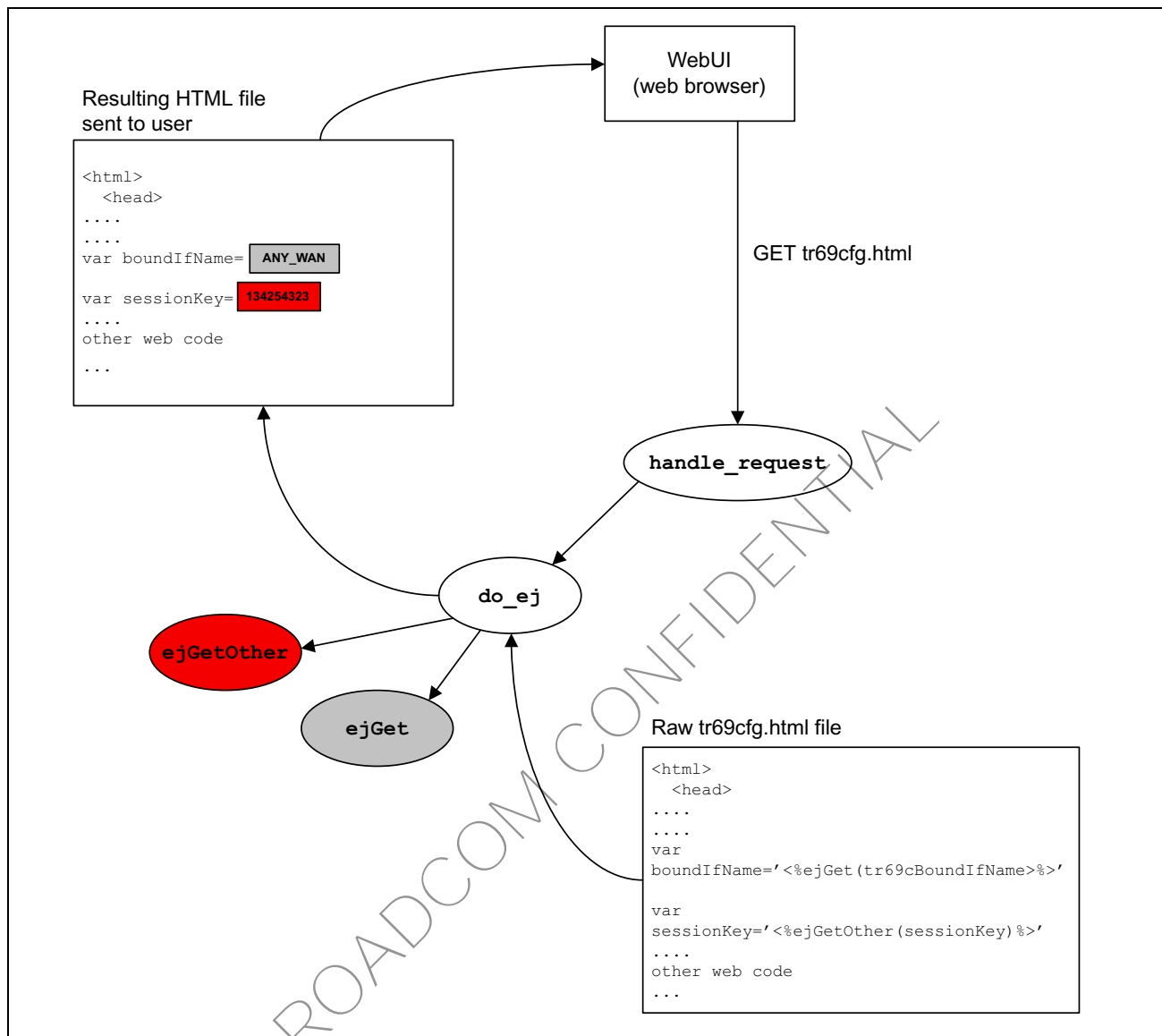
```
<%ejGet(tr69cAcURL)%>
<%ejGetOther(sessionKey)%>
```

The `<%ejGet` tags are handled by the `ejGet()` function, which then calls `cgiGetVar()`. The `cgiGetVar()` command then looks up the variable name inside the `<%ejGet` tag in the `CgiGetTable[]` in `cgi_main.c`. `CgiGetTable[]` then maps the variable name to a field inside the `glbWebVar`. The field is always converted to a string and then sent out to the web browser.

The `<%ejGetOther` tags are handled by the `ejGetOther()` function, which then calls `cgiGetVarOther()`. The `cgiGetVarOther()` command looks up the variable name inside the `<%ejGetOther` tag in the `WebVarTable[]` in `cgi_main.c`. The `WebVarTable[]` then associates a variable name with a function, so that function is called to generate the data for that variable. That data is then sent out to the web browser.

For example, suppose the user clicked on the tr69 configuration page. The web browser first generates a request for `tr69cfg.html`. `ejRefreshGlbWebVar()` calls `cmsDal_getTr69cCfg()` to get the latest tr69c related values from the MDM and put them in `glbWebVar`. `do_ej()` then scans the web page and calls the `ejGet()` and `ejGetOther()` handlers to insert the values. This is partially illustrated in [Figure 9](#).

Figure 9: &lt;%ejGet Tag Processing



## Page Generated from C Code

When the web browser requests a page that ends in .cmd, `handle_request()` will find the `do_cmd_cgi()` `mime_handler` to process this page.

`do_cmd_cgi()` looks up the name of the web page in `WebCmdTable[]` in `cgi_cmd.c` and calls the handler function associated with that file. The handler function then generates the web page using C code and sends the web page to the web browser. Because the web page is being generated using c code, there is no need for the `<%jGet` tags functionality. The C code can directly `sprintf` variable values into a buffer and send the buffer out to the user.

For example, look at `cgiEthWanCfg()` and `cgiEthWanCfgView()`.

---

## Reading Input from User

When the user inputs configuration information in the WebUI via text fields, drop-down lists, option controls, etc., that input must be sent from the web browser to `httpd`. (How to program different controls is out of the scope of this document. It is best to look at existing web pages and copy what they did.)

There are two ways that the web browser sends info back to `httpd`: via (CGI (.cgi) or CMD (.cmd). Both methods are described below.

### CGI Method

If you enable debug logging in `httpd`, when you click **Apply/Save** on the WebUI, `httpd` will log the information that comes back from the web browser. For example, for the `tr69c` config page, you will see a line like:

```
handle_request:619:line=GET /  
tr69cfg.cgi?tr69cInformEnable=0&tr69cInformInterval=300&tr69cAcURL=http://127.0.0.1:4480  
....(stuff omitted)... &sessionKey=2015643031 HTTP/1.1
```

As you can see, the configuration variables start after the first question mark and comes in name/value pairs separated by the "&" symbol. The `sessionKey` is an important variable that will be described in the next section.

For web pages ending in .cgi, `handle_requests()` will find the `do_cgi()` `mime handler` function to process the input.

`do_cgi()` first calls `cgiParseSet()` to extract all the name/value pairs from the string and set the values in the `glbWebVar` according to the mappings in `CgiSetTable[]`.

Then `do_cgi()` will call a specific function based on the filename. For example, if the filename was `tr69cfg`, then the `cgiTr69cConfig()` function will be called. The config functions can assume that whatever data the user entered in the web page has already been updated in the `glbWebVar` structure by `cgiParseSet()`. The data must be converted from simple fields in the `glbWebVar` structure to settings in the CMS Data Model. To do this, the config function can call a DAL function to push the data down via the CMS OBJ API, or the config function can call the CMS OBJ API functions directly.

Finally, `do_cgi()` will send the web page back to the web browser by changing the suffix of the file from `.cgi` to `.html`, and then calling `do_ej()` on the filename. From this point on, the processing is exactly the same as if the user had just requested that web page.

## CMD Method

Some web pages, for example the Ethernet as WAN web page, will send user configuration data back to httpd in the following format:

```
handle_request:619:line=GET /ethwan.cmd?action=add&ifname=eth0&connMode=0&sessionKey=2146027998
HTTP/1.1
```

For web pages ending in `.cmd`, `handle_request()` will find the `do_cmd_cgi()` mime handler function to process the input.

`do_cmd_cgi()` looks up the name of the web page in `WebCmdTable[]` in `cgi_cmd.c` and calls the handler function associated with that file. Because the variable “action” is set to “add”, another function that is designed to process add requests is called. For an example, look at `cgiEthWanCfg()`, which calls `cgiEthWanAdd()`. Note that these functions use `cgiGetValueByName` to extract the value/name pairs from the string sent back by the user. Also note that these `.cmd` pages are used to handle input from the user and to send web pages back to the user.

---

## Preventing Cross Site Request Forgery Attacks

To prevent Cross Site Request Forgery (CSRF) attacks<sup>1</sup>, every web page that configures the CPE has a session key embedded in it. This session key is simply a random number generated by httpd. When a configuration write comes to httpd (via a `.cgi` or `.cmd` page), httpd will verify that the session key in the `.cgi` or `.cmd` string matches the last session key that httpd sent out and that the session key is not too old. (Refer to `HTTPD_SESSION_KEY_VALID` in `httpd.h`.) Note that httpd only remembers the last session key it generated. This means that if browser A reads a web page that contains a session key, and then browser B reads another web page that contains a session key, and then browser A tries to apply some setting, the apply will fail because browser A's session key is no longer valid. Also, a single web page should not have more than one session key embedded in it.

All `*.cgi` pages are verified at the top of `do_cgi()`. The `*.cmd` pages are verified near the bottom of `do_cmd_cgi()`. Look for the `cgiValidateSessionKey()` command.

A session key can be embedded in an `*.html` page using the `<%ejGetOther<sessionKey>%>` tag. Web pages generated by C code can request a new session key by calling `cgiGetCurrSessionKey()` and then embedding the value contained in the `glbCurrSessionKey` parameter inside the web page. Refer to `writeEthCfgScript()` for an example.

---

1. Refer to [http://en.wikipedia.org/wiki/Cross-site\\_request\\_forgery](http://en.wikipedia.org/wiki/Cross-site_request_forgery) for more information on CSRF attacks.

---

## Adding Web Pages to the Left Panel Menu

Users select the feature they want to view and/or configure by clicking on the Left Panel Menu. The content of this menu is controlled by menu.html, menuBcm.js, and menuTitle.js in the html directory. There is also a menutree.js, but you should not have to change anything in this file.

---

## Customizing the Logo

Look at logo.html, logo\_corp.gif, and logobkg.gif in the html directory.

BROADCOM CONFIDENTIAL



## Section 9: Accessing the MDM using the Object Layer Interface

Typically, an application accesses the MDM using the Object Layer API, which is defined in `userspace/public/include/cms_obj.h`.

### Key MDM Concepts

#### MDM Object

In the Object Layer interface, all parameters directly under an object are grouped together in a C data structure, which is called an Object.



**Note:** An object in CMS is not the same as a C++ object. All the functions in the OBJ layer API operate on objects, not on a single parameter.

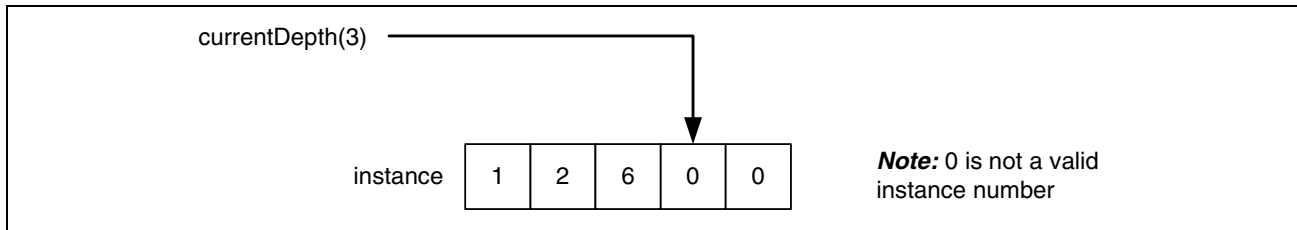
Each object in the Data Model has a constant number assigned to it. The assignment is defined in `userspace/public/include/mdm_objectid.h`. These numbers are called the MDM Object ID, or the Object ID of the object. Most object layer APIs require that the Object ID is passed as one of the parameters. The Object ID is needed by the API to know which object to operate on.

#### Instance ID Stack

Because there can be multiple instances of an object in the MDM, simply specifying the Object ID is often not enough; it is also necessary to specify which instance of the object you want to operate on. The Instance ID Stack (or `iidStack`) is a data structure that is often used in conjunction with the Object ID to specify the exact instance of an object that you want to operate on.

Typically, the `iidStack` can be treated as an opaque cookie from the API. An `iidStack` is created when the `cmsObj_getNext()`, `cmsObj_getNextInSubTree()`, `cmsObj_getAncestor()`, or `cmsObj_addInstance()` API entry points are used. The `iidStack` can be passed back to the `cmsObj_set()`, `cmsObj_getNext()`, `cmsObj_getNextInSubTree()`, and `cmsObj_delete()` API entry points. Detailed examples on how the `iidStack` is used by these API entry points are explained in the later sections.

Sometimes, it is useful to know the exact contents of the `iidStack` for debugging purposes. The remainder of this section describes the internals of the `iidStack`. This is useful if your application is not working as expected.

**Figure 10: InstanceId Stack Structure**

The Instance ID Stack structure is defined in `userspace/public/include/linux/os_defs.h`.

The header file at `userspace/private/include/cms_mdm.h` contains some macros and functions which are useful for dealing with the `iidStack` structure, including:

- `INIT_INSTANCE_ID_STACK`
- `EMPTY_INSTANCE_ID_STACK`
- `cmsMdm_dumpIidStack()`
- `cmsMdm_dumpIidStackToBuf()`

Refer to the comments in `cms_mdm.h` for details on how to use these macros and functions.

## Initializing the MDM

Applications that access the MDM must call `cmsMdm_initWithAcc()`. See `cms_mdm.h` for details about the arguments as well as [“Applications that Access the MDM” on page 138](#) in [Section 19: “Adding a New Application or Command”](#) for an example. Note that `cmsMdm_initWithAcc()` was introduced in 4.14L.01. It is slightly more CPU and memory efficient than the previous initialization function, `cmsMdm_init()`. Both will work, but `cmsMdm_initWithAcc()` is recommended.

## Acquiring and Releasing a Lock

All applications must acquire the MDM lock before using any of the functions in the object layer interface (even operations that only do a get). All applications must release the lock when they are done with calling the object layer interface. There is only one lock that controls access to the MDM, so applications should not hold the lock for too long. (The current goal is a maximum lock hold time of 5 seconds.)

There are only four lock API functions:

- `cmsLck_acquireLock()`
- `cmsLck_acquireLockWithTimeout()`
- `cmsLck_releaseLock()`
- `cmsLck_dumpInfo()`

`cmsLck_dumpInfo()` is a debug function. You can call this function if your code was not able to acquire the lock. `CmsLck_dumpInfo` prints some information about which process is holding the lock, how long it has held it, and other useful information.

As the names suggest, `cmsLck_acquireLock()` will block until it acquires the lock, while `cmsLck_acquireLockWithTimeout()` will take a timeout value. If `cmsLck_acquireLockWithTimeout()` does not get the lock within the timeout specified, then it will return with `CMSRET_TIMED_OUT`. Regardless of how the lock was acquired, it can be released with `cmsLck_releaseLock()`.

A commonly asked question is why does the object layer API not automatically acquire the lock for the application? The reason is that the application may need to do several operations atomically. Therefore, an application must explicitly acquire the lock at the beginning of a sequence of calls into the object layer API and release the lock when it is done. For example, it may need to get all instances of an object and, based on the values of those objects, get another object, modify it, set it, and save the entire data model to the Flash memory. During this sequence of calls, no other application should be allowed to access the MDM and modify an object or even delete an instance of the object, because that would invalidate the information that the first application was reading. Because of these types of scenarios, acquiring and releasing of the lock must be done by the application.

The `httpd` code acquires the lock prior to handling a user request and releases the lock after that request is complete, so the various internal `httpd` handler functions do not need to call any lock code. CLI menu and CLI commands also has a mechanism to acquire and release the lock for the handler functions. This mechanism will be described in the `httpd` and CLI sections. If you are writing a custom application, you will need to implement the appropriate locking calls.

Generally, an application should acquire the lock, make some object layer calls, and release the lock. While holding the lock, applications should not wait for user input since the user may not provide any input for several minutes. While holding the lock, applications also should not block on operations that may take more than a few seconds.

The code below is an example taken from `httpd.c`, showing how the `httpd` code acquires the lock before doing the bulk of the web page processing in the output handler function pointer. Most of the code from `handle_request()` has been omitted so only the relevant lines of code are shown.

```
01 static int handle_request(void)
02 {
03     CmsRet ret;
04     /* most of the code in this function is not shown.
05        the locking code appears near the bottom of the function. */
06
07     ret = cmsLck_acquireLockWithTimeout(HTTPD_LOCK_TIMEOUT);
08     if (ret != CMSRET_SUCCESS)
09     {
10         /* can't get lock, send error page to caller */
11         makePathToWebPage(filename, HTTPD_BUFLLEN_10K, "lockerror.html");
12         do_ej(filename, conn_fp);
13     }
14     else
15     {
16         /* got lock, call handler function */
17         handler->output(filename, conn_fp);
18
19         if (glbSaveConfigNeeded)
20         {
21             ret = cmsMgm_saveConfigToFlash();
22             /* error checking code omitted */
23             glbSaveConfigNeeded = FALSE;
24         }
25     }
26 }
```

```

25
26     cmsLck_releaseLock();
27 }
28
29 }

```

In line 17, we call the function pointed to by the handler->output variable. This handler function does the bulk of the httpd processing, which includes calls the CGI functions, the DAL functions, and ultimately the object layer functions that manipulate the MDM. Because the lock is acquired on line 7, all of those CGI, DAL, OBJ functions are executed with the lock held. When we return from the output handler, we check if we need to save the configuration to Flash memory (line 19). Finally, the lock is released on line 26.

## Reading from the MDM

The Object layer provides several functions for reading/getting an object from the MDM. They are:

- cmsObj\_get()
- cmsObj\_getNext()
- cmsObj\_getNextInSubTree()
- cmsObj\_getAncestor()

Beginning with the 4.02L.03 release, the following additional get variants have been introduced:

- cmsObj\_getNextFlags()
- cmsObj\_getNextInSubTreeFlags()
- cmsObj\_getAncestorFlags()

The function that is used depends mostly the type of the object being retrieved (type 0, 1, or 2) and the situation. However, in all cases the caller is responsible for freeing the object acquired from the get function by calling cmsObj\_free() with the address of the pointer to the object.

The new get APIs that take a “flags” argument were introduced to allow the caller to pass in an OGF\_NO\_VALUE\_UPDATE flag, which is defined in userspace/private/include/cms\_obj.h. The purpose of OGF\_NO\_VALUE\_UPDATE will be explained towards the end of this section. There was no need to introduce a cmsObj\_getFlags() function because cmsObj\_get() already takes a flags argument.

The sections below explain in detail how to use these API functions.

## Using cmsObj\_get()

The simplest object get function is cmsObj\_get(). It is most commonly used to get a type 0 object. As explained in the [Section 2: “Overview,” on page 15](#), there is exactly one instance of a type 0 object in the system. Because the cmsObj\_get() function requires an iidStack argument, you simply pass in an empty/initialized iidStack. In the following example, dalEsw\_getEthernetSwitchInfo() gets the EthernetSwitch object (a type 0 object), and copies the object’s parameter values to the webVar. (webVar is a pointer to the glbWebVar, which is used by the httpd code when it writes out web pages containing current system status.)

```

01 CmsRet dalEsw_getEthernetSwitchInfo(WEB_NTWK_VAR *webVar)
02 {
03     EthernetSwitchObject *switchObj=NULL;

```

```

04 InstanceIdStack iidStack = EMPTY_INSTANCE_ID_STACK;
05 CmsRet ret;
06
07
08 ret = cmsObj_get(MDMOID_ETHERNET_SWITCH, &iidStack, 0, &switchObj);
09 if (ret != CMSRET_SUCCESS)
10 {
11     cmsLog_error("failed to get switch obj, ret=%d", ret);
12     return ret;
13 }
14
15 webVar->virtualPortsEnabled = switchObj->enableVirtualPorts;
16 webVar->numberOfVirtualPorts = switchObj->numberOfVirtualPorts;
17 strncpy(webVar->ethSwitchIfName, switchObj->ifName,
18         sizeof(webVar->ethSwitchIfName)-1);
19
20 /* don't forget to free the object */
21 cmsObj_free((void **) &switchObj);
22 return ret;
23 }

```

In line 4, we initialize the `iidStack` to an empty Instance ID stack. When we call `cmsObj_get()` in line 8, we pass in that empty instance id stack, which tells the API that there is no instance information associated with this object.

Also in line 8, the third argument to `cmsObj_get()` is a flags argument. Normally, no flags are passed into `cmsObj_get()`. However, if you want to get an object with parameters set to their default values instead of their current values in the MDM, you would pass in `OGF_DEFAULT_VALUES`.



**Note:** In line 21, we free the switch object before we return from the function. All objects returned from `cmsObj_get()`, `cmsObj_getNext()`, `cmsObj_getNextFlags()`, `cmsObj_getNextInSubTree()`, `cmsObj_getNextInSubTreeFlags()`, `cmsObj_getAncestor()`, and `cmsObj_getAncestorFlags()` must be freed by the caller using `cmsObj_free()`.

## Using `cmsObj_get()` to Get Type 1 or Type 2 Objects

The `cmsObj_get()` function can also be used to get a type 1 or type 2 object, however, you must have the `iidStack` that points to the instance of the type 1 or type 2 object you want to get (the `iidStack` may have been saved from another function call, or from a `cmsObj_addInstance()` function call). The latter scenario will be demonstrated in [“Adding a New Instance of an Object” on page 79](#).

## Using `cmsObj_getNext()`

The most common way to get type 1 and type 2 objects is by calling `cmsObj_getNext()` in a while loop. There could be zero or more instances of a particular type 1 or type 2 object in the system, so typically, you need to iterate through all the instances of the type 1 or type 2 object to do what you want. The code must also handle the case where there are zero instances of the object. The code below demonstrates the use of `cmsObj_getNext()` in a while loop to count the number of PPP connections that are in the connected state.

```

01 UINT32 getNumberOfConnectedPPP(void)
02 {

```

```

03 WanPPPConnectionObject *pppObj=NULL;
04 InstanceIdStack iidStack = EMPTY_INSTANCE_ID_STACK;
05 CmsRet ret;
06 UINT32 count=0;
07
08 while ((ret = cmsObj_getNext(MDMOID_WAN_PPP_CONNECTION,
09                             &iidStack,
10                             &pppObj)) == CMSRET_SUCCESS)
11 {
12     if (0 == cmsUtl_strcmp(pppObj->connectionStatus, MDMVS_CONNECTED))
13     {
14         count++;
15     }
16     /* don't forget to free the object */
17     cmsObj_free((void **) &pppObj);
18 }
19
20 return count;
21 }

```

On line 4, the Instance ID stack is initialized to empty before the first call to `cmsObj_getNext()`. On return from `cmsObj_getNext()`, the `iidStack` variable is filled out with the instance information of the returned object. On subsequent iterations, the `iidStack` variable from the previous iteration is passed into `cmsObj_getNext()`, causing `cmsObj_getNext()` to return the next instance of the object. Also note that each object returned from `cmsObj_getNext()` must be freed (line 17). The `cmsObj_getNext()` function does not automatically free the object for you if you pass in the previously returned object.

On line 6, `count` must be initialized to 0. If there are no instances of the `WanPPPConnectionObject` in the system, `cmsObj_getNext()` will return an error (`CMSRET_NO_MORE_INSTANCES`) on the first call, so we will immediately go to line 20 and return the count of 0.

## Early Exit from `cmsObj_getNext()`

Often, you only want to know if some condition exists in one of the objects that you are iterating through. When you have found the condition, you can exit early from the while loop. The example below provides you with a design pattern to use for this type of loop. This function is like the previous example, but instead of counting the number of PPP connections in the connected state, it simply returns TRUE if any PPP connection is in the connected state.

```

01 UB00L8 isAnyPPPConnected(void)
02 {
03     WanPPPConnectionObject *pppObj=NULL;
04     InstanceIdStack iidStack = EMPTY_INSTANCE_ID_STACK;
05     CmsRet ret;
06     UINT8 found = FALSE;
07
08     while (!found &&
09           (ret = cmsObj_getNext(MDMOID_WAN_PPP_CONNECTION,
10                               &iidStack,
11                               &pppObj)) == CMSRET_SUCCESS)
12     {
13         if (0 == cmsUtl_strcmp(pppObj->connectionStatus, MDMVS_CONNECTED))
14         {
15             found = TRUE;

```

```

16     }
17     cmsObj_free((void **) &pppObj);
18 }
19
20 return found;
21 }

```

Note the way the while loop tests the found variable and the return value from `cmsObj_getNext()`. Writing the while loop in this way guarantees that the objects returned from `cmsObj_getNext()` will always be freed.

## Returning an Object or iidStack from `cmsObj_getNext()`

Suppose you need to write a utility function to find a specific instance of an object based on a key or some other criteria. When the desired object instance is found, the function will return the object and/or the iidStack. The example below is taken from `rut_pmap.c`. It finds the Layer 2 bridge entry object with the specified key and returns both the object itself and the iidStack of the object.

```

01 CmsRet rutPMap_getBridgeByKey(UINT32 bridgeKey,
02                               InstanceIdStack *iidStack,
03                               L2BridgingEntryObject **bridgeObj)
04 {
05     CmsRet ret;
06
07     INIT_INSTANCE_ID_STACK(iidStack);
08     while ((ret = cmsObj_getNext(MDMOID_L2_BRIDGING_ENTRY,
09                                &iidStack,
10                                &bridgeObj)) == CMSRET_SUCCESS)
11     {
12         if ((*bridgeObj)->bridgeKey == bridgeKey)
13         {
14             break;
15         }
16         /* free the object if there is no match*/
17         cmsObj_free((void **) &bridgeObj);
18     }
19
20     return ret;
21 }

```

In line 7, we initialize the iidStack so that the `cmsObj_getNext()` will return the first instance of the Layer 2 bridging entry object in the system. In lines 12–17, if the object returned from `cmsObj_getNext` does not have the desired bridgeKey, we free the object. However, if the object does have the correct bridgeKey, we break out of the while loop without freeing the object. In line 20, when we return from the function, the ret variable will be still be set to `CMSRET_SUCCESS` if we had found the desired object and broken out of the loop. However, if we did not find the desired object, then we would have broken out of the while loop when `cmsObj_getNext()` returned `CMSRET_NO_MORE_INSTANCES`, and that is what this function would return. If this function returns `CMSRET_SUCCESS` to the caller, that means it also returned the requested L2BridgingEntry object to the caller. The caller is responsible for freeing this object.

## Using cmsObj\_getNextInSubTree()

Sometimes, you want to iterate through a subset of type 1 or type 2 objects. For example, you might use `cmsObj_getNext()` to find a specific instance of a type 2 object. Then, you want to iterate through the (type 1 or type 2) child objects of the object you have. In this case, you should use `cmsObj_getNextInSubTree()`. As the name suggests, `cmsObj_getNextInSubTree()` will only return instances of an object under the specified parent object, while `cmsObj_getNext()` will return all instances of that object in the entire system. The following example is taken from `dal_wan.c`. `getIpConn()` finds an IP connection object with the specified connection ID under a PVC with the specified `portId` and `destAddr`.

```

01 UINT8 getIpConn(UINT32 portId,
02                 char *destAddr,
03                 SINT32 connectionId,
04                 InstanceIdStack *iidStack,
05                 WanIpConnObject **ipConnObj)
06 {
07     InstanceIdStack parentIidStack = EMPTY_INSTANCE_ID_STACK;
08     WanIpConnObject *obj=NULL;
09     UB00L8 found=FALSE;
10     CmsRet ret;
11
12     if (!getDslLinkCfg(portId, destAddr, &parentIidStack, NULL))
13     {
14         return FALSE;
15     }
16
17     INIT_INSTANCE_ID_STACK(iidStack);
18     while ((ret = cmsObj_getNextInSubTree(MDMOID_WAN_IP_CONN,
19                                         &parentIidStack,
20                                         &iidStack,
21                                         &obj)) == CMSRET_SUCCESS)
22     {
23         if (obj->X_BROADCOM_COM_ConnectionId == connectionId)
24         {
25             found = TRUE;
26             break;
27         }
28
29         cmsObj_free((void **) &obj);
30     }
31
32     /* the remainder of the function is not shown */

```

In the TR-98 data model, PVC information is stored in the DSL Link Config object. The call to `getDslLinkCfg()` on line 12 first retrieves the DSL Link Config object with the specified `portId` and `destAddr`. Because the `parentIidStack` variable was only passed in as the third argument the fourth argument was left as `NULL`, we will only get the `iidStack` of the DSL Link Config object and not a pointer to the DSL Link Config object.



The astute reader may have noticed that the DSL Link Config object is not the parent of the Wan IP Connection object. The Wan Connection Device object is the parent of the Wan IP Connection object. However, the iidStack of the DSL Link Config object is the same as the parent WAN Connection Device because the DSL Link Config object is a type 1 child object of the Wan Connection Device type 2 object. Since the DSL Link Config object is a type 1 object, its full path name does not end in `.{i}`. so it does not have any more instance stack information than its parent Wan Connection Device object. So in this example, it would be more accurate to use `ancestorIidStack` rather than `parentIidStack` as the parameter name. However, it is often easier to think of the parent/child relationship rather than the ancestor/descendent relationship.

Now on line 19, we use the `iidStack` of the DSL Link Config object (the `parentIidStack` variable) to iterate through all the child IP Connection objects of that specific DSL Link Config object. Once we find the requested connection ID, we break out of the loop without freeing the object.

The rest of the function is omitted from this example, but it basically checks if we found the requested WAN IP Connection object and returns it to the caller, if requested to do so by the caller.

## Using `cmsObj_getAncestor()`

In another scenario, you may have iterated through a set of (type 1 or type 2) objects and found a specific one. But what you really want is the parent object of the object you just found. In this case, you should use `cmsObj_getAncestorObject()`. The following example is taken from `rut_lan.c`.

```
01 CmsRet getLanDevIidStackOfBridge(const char *bridgeIfName,
02                               InstanceIdStack *iidStack)
03 {
04     UB00L8 found=FALSE;
05     _LanDevObject *lanDevObj=NULL;
06     _LanIpIntfObject *lanIpIntfObj=NULL;
07     CmsRet ret;
08
09     while (!found &&
10           (ret = cmsObj_getNext(MDMOID_LAN_IP_INTF,
11                               iidStack,
12                               (void **) &ipIntfObj)) == CMSRET_SUCCESS))
13     {
14         found = (0 == cmsUtl_strcmp(ipIntfObj->X_BROADCOM_COM_IfName, bridgeIfName));
15         cmsObj_free((void **) &ipIntfObj);
16     }
17
18     if (!found)
19     {
20         return CMSRET_NO_MORE_INSTANCES;
21     }
22
23     if ((ret = cmsObj_getAncestor(MDMOID_LAN_DEV,
24                                 MDMOID_LAN_IP_INTF,
25                                 iidStack,
26                                 (void **) &lanDevObj)) != CMSRET_SUCCESS)
27     {
28         cmsLog_error("could not get ancestor obj, ret=%d", ret);
29         return ret;
30     }
31
32     cmsObj_free((void **) &lanDevObj);
```

```
33     return ret;
34 }
```

This function takes a LAN bridge name and returns the `iidStack` of the LAN Device object that contains the specified LAN bridge.

Lines 9 through 15 iterates through all the LAN bridges in the system to find the desired bridge.

If the desired bridge is found, then by the time we reach line 22, the `iidStack` variable will point to the desired bridge object. Now on line 23, we call `cmsObj_getAncestorObject()` with the `iidStack`. On return, the `iidStack` will now point to the requested ancestor object, which is the LAN Device object. Since we only want the `iidStack` of the LAN Device object and not the object itself, we free the LAN device object on line 32.

## Using the OGF\_NO\_VALUE\_UPDATE Flag

Based on our experiences with using CMS, we realized that when the STL handler function for an object executes non-trivial code, iterating through those objects using the `cmsObj_getNext()` and `cmsObj_getNextInSubTree` caused inefficiencies and undesirable behavior in the system. STL handler functions will be described in [Section 14: "System Status Layer," on page 100](#); briefly, whenever an object is retrieved through any of the `cmsObj_getXXX` interfaces, the STL handler function is called to update the object before it is returned to the caller.

For example, the `getHostEntryByMacAddr()` function in `userspace/apps/ssk/lanhosts.c` iterates through all the `LANDevice.{i}.Hosts.Host.{i}.` entries under the specified `LANDevice` to find the entry the specified MAC address. However, every time the `cmsObj_getNextInSubTree(MDMOID_LAN_HOST_ENTRY, ...)` is called by `ssk`, the STL handler function for the `LANDevice.{i}.Hosts.Host.{i}.` is called. This STL handler function sends a message to `dhcpcd` to update the `LeaseTimeRemaining` field in the host object. If there are 50 host entries, then every call to `getHostEntryByMacAddr()` could generate up to 50 messages to `DHCPD`.

The key thing to notice in this scenario is that `getHostEntryByMacAddr()` only wanted to read the `MACAddress` field of the host object, and the `MACAddress` field of the host object will never change. However, when the STL handler function for the host object is called, it does not know what field the caller wanted, so it is forced to update all the fields. The solution is to use the `cmsObj_getNextInSubTreeFlags()` function with the `OGF_NO_VALUE_UPDATE` flag. This flag will tell the lower levels of the API to return the object from the MDM without calling the STL handler to update the object.

We have noticed that in many cases, code which iterates through a set of objects really just want the value of some parameter that is not updated by the STL handler function, so we strongly recommend that you use the `cmsObj_getXXXFlags()` variant of the API with the `OGF_NO_VALUE_UPDATE` whenever possible.

---

## Writing to the MDM

To write data into the MDM, you should first do a get of the object instance you want to modify, then modify the object, and then do a set. You should not declare the object on the stack, modify the object, and then do the set. If you declare the object on the stack, you are responsible for filling in all the parameters of the object, and it is easy to unintentionally change a parameter or not fill in some special internal fields inside the object. By getting the object from `cmsObj_get()` or one of its variants, and then only modifying the parameters you want to modify, you are guaranteed that only the parameters you want to change will be changed and all the special internal fields the object will be filled in correctly because the object came from the MDM.

The only function for writing the data to the MDM is `cmsObj_set()`. `cmsObj_set()` requires the pointer to the object and the Instance ID stack information associated with the object. (You should have a properly filled out the Instance ID stack information variable when you did a get of the object.) `cmsObj_set()` does not free the object for you, so you must free the object using `cmsObj_free()`.

The first example of a set enables the virtual ports feature on the Ethernet switch. Since the Ethernet switch object is a type 0 object (there is exactly one instance of this object in the system), doing a get and a set is easy.

```
01 void dalEsw_enableVirtualPorts(UB00L8 enable)
02 {
03     EthernetSwitchObject *switchObj=NULL;
04     InstanceIdStack iidStack = EMPTY_INSTANCE_ID_STACK;
05     CmsRet ret;
06
07     ret = cmsObj_get(MDMOID_ETHERNET_SWITCH, &iidStack, 0, &switchObj);
08     if (ret != CMSRET_SUCCESS)
09     {
10         cmsLog_error("failed to read ETH_SWITCH obj, ret=%d", ret);
11         return;
12     }
13
14     switchObj->enableVirtualPorts = enable;
15
16     if ((ret = cmsObj_set(switchObj, &iidStack)) != CMSRET_SUCCESS)
17     {
18         cmsLog_error("cmsObj_set failed, ret=%d", ret);
19     }
20
21     cmsObj_free((void **) &switchObj);
22     return;
23 }
```

The second example modifies a Layer2Bridging Filter object. Since the filter object is a type 2 object, we must call another function `dalPMap_getFilter()` (implementation not shown) to get the desired instance of the filter object before setting it. Even though this example is taken from actual code in `dal_pmap.c`, it has been heavily modified so that only the key lines of code are shown.

```
01 CmsRet dalPMap_assocFilterIntfToBridge(const char *ifName, const char *grpName)
02 {
03     L2BridgingFilterObject *filterObj=NULL;
04     InstanceIdStack filterIdStack = EMPTY_INSTANCE_ID_STACK;
05     Char filterInterface[BUFLen_32];
06     CmsRet ret;
07
08     /* the code which converts ifName to a id string which is put into
09        filterInterface is not shown. The next line finds the
10        filter object that corresponds to ifName. */
11     ret = dalPMap_getFilter(filterInterface, &filterIdStack, &filterObj);
12     if (ret != CMSRET_SUCCESS)
13     {
14         cmsLog_error("could not find filter intf %s", ifName);
15         return ret;
16     }
17
18     /* the code which converts grpName to bridgeRef is not shown.
19        The important thing to see is the filterObj's filterBridgeReference
20        param is modified in the next line. */
21     filterObj->filterBridgeReference = bridgeRef;
22
23     if ((ret = cmsObj_set(filterObj, &filterIdStack)) != CMSRET_SUCCESS)
24     {
25         cmsLog_error("cmsObj_set failed, ret=%d", ret);
26     }
27
28     cmsObj_free((void **) &filterObj);
29     return;
30 }
```

The most important difference between the first example and the second example is that in the second example, we are modifying an object that can have multiple instances in the system. Therefore, we relied on another function, `dalPMap_getFilter()` on line 11, to find the specific instance of the filter we want to modify based on the `ifName`. `dalPMap_getFilter()` returns a copy of the specified filter object and a `filterIdStack` variable that contains the instance information for the specific instance of the filter object that was returned. Both the `filterObj` and the `filterIdStack` are needed when `cmsObj_set()` is called. This more complex form of Get before Set is required when setting type 1 and type 2 objects.

## Setting String Parameters

In both of the above examples demonstrating `cmsObj_set()`, an integer parameter was changed. Assigning a new value to a signed integer, unsigned integer, or boolean is very simple. However, when setting a string (`char *`) parameter, special care must be taken.

First, when you get a object back from `cmsObj_get()` or one of its variants, the string parameter fields (fields with type `char *`) may already be pointing to a string buffer. So before setting the string parameter to a new value, the current string buffer must be freed, otherwise, the original string buffer will be “lost”, resulting in a memory leak.

Second, when assigning a value to a string buffer, use a buffer allocated using `cmsMem_alloc` or one of its variants. Do not use plain `malloc`, and do not point the string parameter to a string allocated from the stack or in the global data section. This restriction comes from the fact that after you call `cmsObj_set()`, you will most likely call `cmsObj_free()` on that object. `cmsObj_free()` will first call `cmsMem_free()` on all the `char *` fields in the object before freeing the object buffer itself. So if you assigned a buffer that was allocated using plain `malloc`, from the stack, or from the global data section, then the `cmsMem_free()` will fail and either leak memory or cause the application to terminate.

Since `cmsObj_free()` calls `cmsMem_free()` on all string parameters, the buffer that the string parameter points to must not be referenced by any other variables. If another variable also points to the buffer and that buffer gets freed in `cmsObj_free()`, then the other variable will be pointing to a buffer that has been freed and could potentially be given to another function. This scenario will typically result in a double-free error or memory corruption.

The following examples uses code from the `dalSec_addIpFilterOut()` function. The first example shows the incorrect way to set a string parameter. The second example shows the correct way.

```
01 /* a string buffer in global data section */
02 char *loopbackAddr = "127.0.0.1";
03
04 dalSec_addIpFilterOut(/* args omitted */)
05 {
06     IpFilterCfgObject *ipFilterCfg = NULL;
07     InstanceIdStack iidStack = EMPTY_INSTANCE_ID_STACK;
08     char buf[128];
09     CmsRet ret;
10
11     /* some parts of this function is omitted */
12
13     cmsObj_get(MDMOID_IP_FILTER_CFG, &iidStack, 0, (void **) &ipFilterCfg);
14
15     /* WRONG! memory leak, previous filterName buffer not freed first */
16     ipFilterCfg->filterName = cmsMem_strdup("new name");
17
18     /* WRONG! do not use plain malloc */
19     ipFilterCfg->filterName = malloc(16);
20     sprintf(ipFilterCfg->filterName, "new name");
21
22     /* WRONG! do not point to global data */
23     ipFilterCfg->sourceIPAddress = loopbackAddr;
24
25     /* WRONG! do not point to stack data */
26     sprintf(buf, "125.7.1.1");
27     ipFilterCfg->sourceIPAddress = buf;
```

```
28
29  /* WRONG! do not share pointers */
30  ipFilterCfg->sourceIPAddress = lanIpIntf->IPInterfaceIPAddress;
31
32  cmsObj_set(ipFilterCfg, &iidStack);
33
34  cmsObj_free((void **) &ipFilterCfg);
```

This is the correct way:

```
01  /* a string buffer in global data section */
02  char *loopbackAddr = "127.0.0.1";
03
04  dalSec_addIpFilterOut(/* args omitted */)
05  {
06      IpFilterCfgObject *ipFilterCfg = NULL;
07      InstanceIdStack iidStack = EMPTY_INSTANCE_ID_STACK;
08      CmsRet ret;
09
10      /* some parts of this function is omitted */
11
12      cmsObj_get(MDMOID_IP_FILTER_CFG, &iidStack, 0, (void **) &ipFilterCfg);
13
14      /* Free previous buffer first */
15      cmsMem_free(ipFilterCfg->filterName);
16      ipFilterCfg->filterName = cmsMem_strdup("new name");
17
18      /* You can do the above in one line with this macro */
19      CMSMEM_REPLACE_STRING(ipFilterCfg->filterName, "new name");
20
21      /* If you need to point to global or stack data, make a copy first */
22      ipFilterCfg->sourceIPAddress = cmsMem_strdup(loopbackAddr);
23
24
25      /* If you need to share data pointer, make a copy first */
26      ipFilterCfg->sourceIPAddress = cmsMem_strdup(
27          lanIpIntf->IPInterfaceIPAddress);
28
29      cmsObj_set(ipFilterCfg, &iidStack);
30
31      cmsObj_free((void **) &ipFilterCfg);
```

## Adding a New Instance of an Object

New instances of objects are added to the MDM using the `cmsObj_addInstance()` function.

`cmsObj_addInstance()` can only be called on type 2 objects. If a type 2 object has type 1 child objects, then the type 1 child object instances are automatically added when the type 2 object instance is added. However, it does not make sense to automatically add type 2 child objects.

The procedure for adding another instance of an object varies slightly depending on the parent of the object. If the parent of the object to be added is a type 0 object, then you can simply call `cmsObj_addInstance()` with an empty (initialized) Instance ID stack. Otherwise, you must first use one of the `cmsObj_getXXX` functions to find the correct instance of the parent object that you want to add the new object under, and then call `cmsObj_addInstance()` with the Instance ID stack information of the parent object. (This procedure ensures that the new object instance is added under the desired parent object.)

This first example shows how a static route entry is added to the MDM. Because the forwarding object's parent, `Layer3Forwarding`, is a type 0 object, only an empty (initialized) Instance ID stack is passed to `cmsObj_addInstance()`.

```
01 CmsRet dalRoute_addEntry(const char *addr, /* other params omitted */)
02 {
03     InstanceIdStack filterIdStack = EMPTY_INSTANCE_ID_STACK;
04     Layer3ForwardingEntryObject *forwardObj=NULL;
05     CmsRet ret;
06
07     ret = cmsObj_addInstance(MDMOID_LAYER3_FORWARDING_ENTRY, &iidStack);
08     if (ret != CMSRET_SUCCESS)
09     {
10         cmsLog_error("could not add Forwarding obj, ret=%d", ret);
11         return ret;
12     }
13
14     /* on return, iidStack is filled with instance information for
15      * the newly added instance. This iidStack can be used to get
16      * the newly added object instance. */
17     ret = cmsObj_get(MDMOID_L3_FORWARDING_ENTRY,
18                    &iidStack, 0, (void **)&forwardObj);
19     /* typically, after adding an object instance, you want to get it,
20      * modify some parameters, and do a set again. This is not shown here */
21
22     /* since this example did a get of the forward obj, we need to free it */
23     cmsObj_free((void **) &forwardObj);
24     return;
25 }
```

Note that on line 17, we are using `cmsObj_get()` to get the Forward object. Typically, we have to use `cmsObj_getNext` or `cmsObj_getNextInSubTree` to get a type 2 object. However, because `cmsObj_addInstance` filled the `iidStack` with the instance information of the newly added Forward object instance, we can use that `iidStack` in `cmsObj_get` to get the instance we want.

When you add a `cmsObj_addInstance` of an object, the parameters of the object will all be in their default state. Typically, immediately after the `addObject`, you will want to do a `get` of the instance you just created, modify some of the parameters, and then do a `set` again. This entire sequence is not shown in the above example. Refer to this function in `dal_route.c` to see the entire sequence.

This second example shows a new WANPPPConnectionObject being added under a WANConnectionDeviceObject. The second argument of addWanPPPConnection, iidStack, contains the instance information of the parent WANConnectionDeviceObject. This iidStack was filled in by getDs1LinkCfg(), which was called by dalWan\_addInterface(). These two functions are not shown in this example. Refer to dal\_wan.c for the full source code.

```
01 CmsRet addWanPPPConnection(const WEB_NTWK_VAR *webVar, InstanceIdStack *iidStack)
02 {
03     CmsRet ret;
04
05     /* this function is quite complex, only the relevant parts are shown here.
06      * The key thing to note here is unlike the previous example, iidStack
07      * is not initialized/empty. It was filled out with the correct parent
08      * object information and passed into this function. */
09
10     ret = cmsObj_addInstance(MDMOID_WAN_PPP_CONN, &iidStack);
11     if (ret != CMSRET_SUCCESS)
12     {
13         cmsLog_error("failed to add wanPPPConnInstance, ret=%d", ret);
14         return ret;
15     }
16
17     /* the rest of this function is not relevant to this example and has
18      * been deleted. */
19 }
```

cmsObj\_addInstance() can only be called on type 2 objects. However, there are two different kinds of type 2 objects: Dynamic Instance and Multiple Instance.

A Dynamic Instance type 2 object allows management applications to call cmsObj\_addInstance() on it. For example, Layer3Forwarding.Forwarding.{i}. and WanDevice.{i}.WanConnectionDevice.{i}. WANPPPConnection.{i}. are both examples of dynamic instance type 2 objects. If the object is defined by the Broadband Forum, its object element entry in the data model file will have an attribute setting = PC. If the object is not defined by the Broadband Forum, there will be no requirements attribute. In both cases, the support level will be Dynamic Instances.

A Multiple Instance type 2 object can have zero or more instances in the system, and their numbers can change depending on system state, however, management applications are not allowed to call cmsObj\_addInstance() on it. Only internal system applications, such as ssk, can add or delete multiple instance type 2 objects.

Examples of multiple instance type 2 objects are LANDevice.Hosts.Host.{i}. and ManagementServer.ManageableDevice.{i}. These object instances reflect system states that the management applications have no control over, therefore, it does not make sense for management applications to add instances of these objects using the cmsObj\_addInstance() function. If the object is defined by the Broadband Forum, its object element entry in the data model file will have an attribute setting = R. If the object is not defined by the Broadband Forum, there will be no requirements attribute. In both cases, the support level will be set to Multiple Instances.



## Deleting an Existing Instance of an Object

Object instances are deleted from the MDM using the `cmsObj_deleteInstance()` function. Only type 2 object instances can be deleted. If a type 2 object has type 1 or type 2 child objects, then all type 1 and type 2 child object instances will be deleted when the type 2 object instance is deleted.

To delete an object instance, first use one of the `cmsObj_getXXX` functions to find the desired object instance to delete. Then call `cmsObj_deleteInstance()`.

The example below shows how a specific `WANIPConnectionObject` is deleted. The first argument of `deleteIpOrPppConnection`, `connectionId`, is an identifier used to find the specific `WANIPConnectionObject`. The second argument, `iidStack`, contains the instance information of the parent `WANConnectionDeviceObject`. This `iidStack` was filled in by `getDslLinkCfg()`, which was called by `dalWan_deleteInterface()`. These two functions are not shown in this example. Refer to `dal_wan.c` for the full source code.

```
01 void deleteIpOrPppConnection(SINT32 connectionId, const InstanceIdStack *iidStack)
02 {
03     InstanceIdStack searchIidStack = EMPTY_INSTANCE_ID_STACK;
04     WanIpConnObj *ipConnObj=NULL;
05     UB00L8 found=FALSE;
06     CmsRet ret;
07
08
09     while ((!found) &&
10           ((ret = cmsObj_getNextInSubTree(MDMOID_WAN_IP_CONN, iidStack,
11                                           &searchIidStack, (void **) &ipConnObj)) == CMSRET_SUCCESS)
12     {
13         if (ipConnObj->X_BROADCOM_COM_ConnectionId == connectionId)
14         {
15             cmsObj_deleteInstance(MDMOID_WAN_IP_CONN, &searchIidStack);
16             found = TRUE;
17         }
18         cmsObj_free((void **) &ipConnObj);
19     }
20
21     /* the rest of this function is not relevant to this example and has
22        * been deleted. */
23 }
```

Note that on line 10, we are using `cmsObj_getNextInSubTree()` instead of `cmsObj_getNext()`. This is because the `connectionId` is only unique within a particular subtree of the `WANConnectionDevice`. The `iidStack` in the second argument of the function identifies the specific subtree to search in, and the `connectionId` identifies the desired `WANIPConnection`.

When the desired `WANIPConnection` object has been found, the `searchIidStack` contains the instance information for that instance of the `WANIPConnection` object. So on line 15, `cmsObj_deleteInstance()` is called with the `MdmObjectId` and the `searchIidStack`.

Note that on line 18, we still delete the `ipConnObj` that we got back from `cmsObj_getNextInSubTree()` even though we had called `cmsObj_deleteInstance()` on line 15. This is because the object returned by `cmsObj_getNextInSubTree()` (and in fact, from all the `cmsObj_getXXX` variants) is a completely independent copy of the object than the one in the MDM. So even though we had deleted that object instance in the MDM, we still need to delete our copy of it.

The same distinction between Dynamic Instances and Multiple Instances that was explained in the previous section on `cmsObj_addInstance()` apply to `cmsObj_deleteInstance()`.

## Deleting Multiple Instances Inside the While Loop

In the previous example, we used a while loop to look for a particular instance of an object. Once the desired object instance was found, we deleted it and exited from the while loop. However, there could be a scenario where you need to traverse through all the instances of an object and delete one or more instances of that object. Ideally, you should be able to do this in a single pass through the while loop.

A straightforward (but incorrect) solution might look like the example below.

```
01     INIT_INSTANCE_ID_STACK(&iidStack);
02
03     while ((ret = cmsObj_getNext(MDMOID_WAN_IP_CONN, &iidStack,
04                                 (void **) &ipConnObj)) == CMSRET_SUCCESS)
05     {
06         if (search_criteria_met)
07         {
08             cmsObj_deleteInstance(MDMOID_WAN_IP_CONN, &iidStack);
09         }
10         cmsObj_free((void **) &ipConnObj);
11     }
12
```

The above code will not work because of the way `cmsObj_getNext()` uses the `iidStack`. When `cmsObj_getNext()` is called, it first finds the object instance pointed to by `iidStack` and then returns the next instance. As a special case, if the `iidStack` has been freshly initialized, which is the case when we call `cmsObj_getNext()` for the first time, then the first object instance is returned.

In the above example, when the search criteria is met, the object instance pointed to by `iidStack` is deleted by calling `cmsObj_deleteInstance()` on line 8. Now when the code goes back up to the while loop and calls `cmsObj_getNext()` again, `cmsObj_getNext()` will not be able to find the object instance pointed to by `iidStack` because that instance has been deleted.

The correct solution is to employ two `iidStack` variables. One `iidStack` variable is used as above. The second `iidStack` variable keeps track of the last nondeleted instance of the object. When the search criteria is met and the object instance pointed to by `iidStack` is deleted, the `iidStack` is reset to the last nondeleted instance, which is stored in the second `iidStack` variable. This ensures that `cmsObj_getNext()` will always be able to find the previous object instance and return the next one. The algorithm is illustrated below:

```
01  INIT_INSTANCE_ID_STACK(&iidStack);
02  INIT_INSTANCE_ID_STACK(&iidStack2);
03
04  while ((ret = cmsObj_getNext(MDMOID_WAN_IP_CONN, &iidStack,
05                               (void **) &ipConnObj)) == CMSRET_SUCCESS)
06  {
07      if (search_criteria_met)
08      {
09          cmsObj_deleteInstance(MDMOID_WAN_IP_CONN, &iidStack);
10
11          /* restore iidStack to last non-deleted instance, since
12           * iidStack now points to a non-existent object instance. */
13          iidStack = iidStack2;
14      }
15      else
16      {
17          /* remember the last non-deleted instance */
18          iidStack2 = iidStack;
19      }
20
21      cmsObj_free((void **) &ipConnObj);
22  }
```

The same concept applies to `cmsObj_getNextInSubTree()`.

---

## Saving Changes to Flash Memory

The Object layer and the MDM API do not automatically save the MDM to the Flash. It is up to the application to flush the MDM to Flash by calling `cmsMgm_saveConfigToFlash()`. However, note that in the `httpd`, `tr69c`, and `CLI` code, there is a global variable that can be set to cause common code to call `cmsMgm_saveConfigToFlash()`. So if you are implementing code in those applications, set the global variable instead of calling `cmsMgm_saveConfigToFlash()` in your own code. However, if you are implementing a custom application (or porting an existing application to CMS), you will need to decide when to call this function in your own application.

When `cmsMgm_saveConfigToFlash()` is called, only some parts of the MDM are written out based on the following rules:

- Any parameter that has been marked with `alwaysWriteToConfigFile="true"` in the data-model file is written to the config file.
- Any parameter that has been marked with `neverWriteToConfigFile="true"` in the data-model file is not written to the config file.
- Any object and its subobjects and parameters that has been marked with `pruneWriteToConfigFile="true"` in the data-model file is not written to the config file.
- All parameters that are read/write and have changed from their default values and are not affected by the previous rules are written to the config file.

If your system uses a NOR flash, the config file is written to a predefined sector in the Flash. The config file is not located anywhere in the file system. However, if you are using a NAND Flash part and are running on top of JFFS2, then the config file will be in a directory location.

By default, the reference software will write the config file to only one location (sector) in the Flash memory. However, starting with the 4.04L.01 release, the reference software can be configured to write the config file to two different sectors in the Flash memory every time `cmsMgm_saveConfigToFlash()` is called. This dual config file feature protects against power loss or system crash in the middle of a write to the config file. When the dual config file feature is enabled, each config file is written out with a CRC. When the system boots, `mdm_loadconfig()` in `mdm_configin.c` will first read in the primary copy config file and verify the CRC. If the CRC is not valid, then the secondary copy of the config file is read in.

To enable the dual config file feature, you have to do two things:

- The bootloader (CFE) must be configured to allocate an extra sector for the secondary config file:
  - At the bootloader prompt, type **e n** (for erase nvram).
  - Answer the questions that it prompts you for.
  - When the question is **Enable Backup PSI [0]1**, type **1**.
- In **make menuconfig**, in the **Other Features** section, select **Support Backup Config File**. Once this option is enabled, you have to do a complete clean make of the software. All CMS code associated with this feature is under `#ifdef SUPPORT_BACKUP_PSI`.

## Section 10: Data Access Layer

The DAL is used by the httpd and CLI code to manipulate the MDM. Because httpd and CLI perform many common operations, e.g., adding a WAN connection, the code in the DAL can be shared/reused by the httpd and CLI code.

The existing httpd and CLI code often stores configuration data to and reads configuration data from a large data structure called `WEB_NTWK_VAR` (`g1bwebVar`). The DAL has many functions that convert configuration data from the `WEB_NTWK_VAR` to the MDM objects and back again.

This section is currently under development.

BROADCOM CONFIDENTIAL

## Section 11: Query Data Model (QDM) Library

Starting in the 4.14L.XX release series, a new library and API layer called Query Data Model (QDM) was introduced. The QDM layer has two main purposes:

- Hide differences between TR-98 and TR-181 from the calling code, therefore making the calling code cleaner.
- Allow sharing of functions between high level application code (DAL and above) with low level code in the RCL/STL/RUT functions. During our many years of experience with CMS, we noticed that sometimes application level code and RUT level code were doing the same kinds of lookups/queries in the MDM. Instead of writing two copies of the same function, we would define a function in the DAL which just calls the same function in the RUT layer. See [“Sharing RUT Functions with DAL” on page 99](#). However, this is a violation of the CMS layering rules. By creating a separate library, both application layer and low level code can share functions without violating CMS layering rules.

The main design rules for QDM functions are as follows (these are also found in `cms_qdm.h`):

1. Functions must not modify the MDM. It should only query, search, and return results.
2. Function prototypes must be data model independent. This means functions must not accept MDM objects or instance id stacks as arguments since these imply the data model. They must also not return MDM objects or iidStacks. Note this does not mean the functions themselves are data model independent. Many functions in the QDM will have two versions: one to handle the query in the TR98 data model, and one to handle the query in the TR181 data model.
3. Callers must have the MDM lock before calling QDM.

See `userspace/private/libs/cms_qdm` for more details and examples.

## Section 12: Runtime Configuration Layer

Each object in the data model must have a runtime configuration layer (RCL) handler function. The RCL handler function of an MDM object is responsible for configuring the system according to the parameter settings in the object. Everything above the RCL handler function is simply doing software manipulations of MDM objects. The RCL handler function translates the settings in the MDM object into real actions: system calls, ioctls, executing commands, starting other applications, etc. All RCL handler functions are located in `userspace/private/libs/cms_core/linux`.

The RCL handler function for an object is called under four scenarios:

- When the system boots and loads the configuration file.
- When an instance of the object is added via the `cmsObj_addInstance()` or `cmsPh1_addObjInstance()` function.
- If the object is modified via `cmsObj_set()` or `cmsPh1_setParameter()` functions.
- When the object is deleted via the `cmsObj_deleteInstance()` or `cmsPh1_delObjInstance()` functions.

Each of these scenarios is described in the sections below.

All RCL handler functions have the same function prototype:

```
CmsRet rcl_XXXObject(_XXXObject *newObj, _XXXObject *currObj, InstanceIdStack *iidStack, char **errorParam, CmsRet *errorCode);
```

where `xxx` is the short object name of the object.

The `newObj` and `currObj` parameters will be described in the sections below. `iidStack` is the instance information for the object that this handler is being called for. `errorParam` is a parameter that the RCL handler function can use to specify a particular parameter that failed. The RCL handler function is responsible for allocating a buffer large enough to hold the string name of the parameter. The ODL, which calls the RCL handler function will free the `errorParam`. The `errorCode` parameter is a specific error code for the `errorParam` that failed. The `errorParam` and `errorCode` parameters are only used by `tr69c`. If `errorParam` and `errorCode` are not set, `tr69c` uses the return value from the function. Generally, just returning a `CmsRet` value from the RCL handler function is sufficient.

The next four sections describe how the RCL handler function can determine which of the four scenarios it is being called in. Knowing how to determine these four scenarios will enable you to write an RCL handler function. However, you should also [“RCL Design Patterns” on page 91](#)<sup>2</sup>.

In the examples that follow, you will see many RCL handler functions calling an RUT function. RUT functions are helper functions to the RCL handler functions. RUT functions help reduce the amount of code in the RCL handler functions so that the true intentions of the RCL handler functions are easier to understand. RUT functions are described in further detail in [Section 13: “Runtime Utilities,” on page 99](#).

---

2. A design pattern is a proven successful approach to solving a common problem, which in this context is how to write a RCL handler function.

---

## At System Bootup

When the system boots, the MDM is initialized from the configuration file or if there is no valid configuration file, it is given a default structure and settings by `mdm_adjustForHardware()`. In either case, after the MDM has been initialized, the RCL handler functions for all objects are called with `newObj` set to the object in the MDM and `currObj` set to NULL.

The RCL handler function should then configure the system according to the settings in the `newObj`. This causes the system to be initialized according to the configuration file.

Note that both during system bootup and when an object is added via `cmsObj_addInstance()` and `cmsPh1_addObjInstance()`, the `newObj` variable is set to the new object and `currObj` variable is set to NULL. So this naturally leads to the question, “how can the RCL handler function tell if it is being called at system bootup time or because of a call to `cmsObj_addInstance()` or `cmsPh1_addObjInstance()`?” There are two methods: first, the RCL handler function can look at a parameter that is typically set to a non-default or non-null value in a configured object. If that parameter has a non-default or non-null value, then the RCL handler function is being called during system bootup (see [“RCL Design Pattern 2” on page 94](#)). Second, the RCL handler function can test the variable `mdmShmCtx.inMdmInit`, which will be TRUE if the RCL is being called during system bootup. However, this mechanism is only available starting with the 4.02L.03 release. Note that in many cases, the RCL handler does not need to distinguish between a call during system startup time or because of a call to `cmsObj_addInstance()/cmsPh1_addObjInstance()`. See [“RCL Design Pattern 1” on page 92](#) as an example.

---

## When an Object is Added

When a management application calls `cmsObj_addInstance()` or `cmsPh1_addObjInstance()`, the RCL handler function is called with the `newObj` variable set to the new object and the `currObj` set to NULL. You can test for this condition with the `ADD_NEW()` macro defined in `rut_util.h`.

In this scenario, the RCL handler function should configure the system according to the settings in `newObj`. Note that many objects have a parameter called `Enable`. The RCL handler function should not configure the system according to the settings in `newObj` if the value of the `enable` parameter is FALSE. If the object has an `enable` parameter, you can use the `ENABLE_NEW_OR_ENABLE_EXISTING()` or the `ENABLE_NEW()` macros to test for this condition. See also [“RCL Design Pattern 1” on page 92](#).

Some objects contain a read-only parameter that must be initialized by the RCL handler function when it is added. For example, the `LANDevice.LANHostConfigManagement.IPInterface.{i}` object has a read-only parameter called `X_BROADCASTOM_COM_IfName`. This parameter contains the name of the bridge that is used to implement a particular subnet. The management application does not set this parameter. Instead, when the object is added, the RCL handler function finds the first available bridge device (e.g., `br1`) and sets this parameter to the name of the bridge device. Once the interface name has been assigned to a bridge device, it is written to the config file so that the same bridge device will be used across reboots of the system. Since this parameter is written to the config file, when the system boots again and the RCL handler function is called on bootup, the RCL handler function should just use the previously assigned bridge device name and not get another bridge device name.



The following snippet of code from `rcl_ipInterfaceObject()` shows how this is done:

```

01 CmsRet rcl_lanIpIntfObject(_LanIpIntfObject *newObj,
02                             const _LanIpIntfObject *currObj,
03                             const InstanceIdStack *iidStack,
04                             char **errorParam __attribute__((unused)),
05                             CmsRet **errorCode __attribute__((unused)))
06 {
07     CmsRet ret;
08
09     /* if this is a new bridge creation, then get an available bridge number */
10     if (newObj != NULL && currObj == NULL && newObj->X_BROADCOM_COM_IfName == NULL)
11     {
12         SINT32 bridgeNum;
13         char bridgeName[BUFLen_32];
14
15         bridgeNum = rutLan_getNextAvailableBridgeNumber();
16         sprintf(bridgeName, "br%d", bridgeNum);
17         CMSMEM_REPLACE_STRING_FLAGS(newObj->X_BROADCOM_COM_IfName,
18                                     bridgeName, mdmLibCtx.allocFlags);
19     }
20
21     /* the rest of this function is not relevant to this example so is omitted. */
22 }

```

Line 10 is the test for the new object add case. The third comparison in that line (`newObj->X_BROADCOM_COM_IfName == NULL`) will be TRUE only when the object is added by a `cmsObj_addObject()` or `cmsPh1_addObjInstance()`. This is because by default, the `X_BROADCOM_COM_IfName` is NULL, so when the object is added, the variable will be NULL. However, if this RCL handler function was being called during bootup, the `X_BROADCOM_COM_IfName` would be set to the value that was in the config file.

There are a couple of other things worth noting in this example.

On lines 1 and 2, the `LanIpIntfObject` structure type has an underscore in front of it. For now, both `_LanIpIntfObject` and `LanIpIntfObject` define the same structure. However, it is possible that in the future, they may define different objects to maintain binary compatibility with existing applications. When writing a RCL, RUT, or STL function, use the version of the structure with the underscore in front. When writing application code or DAL functions, use the version without the underscore.

On lines 4 and 5, `__attribute__((unused))` suppresses compiler warnings about unused variables.

On line 17, the `CMSMEM_REPLACE_STRING_FLAGS()` macro first frees the existing string in the first argument if it is not NULL, then does a `cmsMem_strdupFlags()` of the second argument with the flags in the third argument. In RCL, RUT, and STL functions, all string allocations for parameters in `MdmObjects` must use `cmsMem_strdupFlags()` with `mdmLibCtx.allocFlags` as the flags argument. (In application and DAL code, you can use `cmsMem_strdup()` or `CMSMEM_REPLACE_STRING()`).

## When an Object is Modified

Once the system has booted, a management application may modify an existing object by calling `cmsObj_set()` or `cmsPh1_setParameterValues()`. In this scenario, the RCL handler function will be called with `newObj` pointing to an object containing the new values and `currObj` pointing to an object with the current values.

In theory, the RCL handler function should compare all the parameters of `newObj` and `currObj`, find out which parameters have changed, and configure the system with the new values. In practice, most RCL handler functions just need to detect if a change has occurred in any parameter, and if so, delete or disable the previous configuration and add or reenable the system with the new configuration.

The example below shows how the `X_BROADCOM_COM_LoginCfg` object detects modifications to the admin, support, or user passwords.

```
01 CmsRet rcl_loginCfgObject(_LoginCfgObject *newObj,
02                          const _LoginCfgObject *currObj,
03                          const InstanceIdStack *iidStack __attribute__((unused)),
04                          char **errorParam __attribute__((unused)),
05                          CmsRet **errorCode __attribute__((unused)))
06 {
07     CmsRet ret;
08
09     if ((currObj == NULL) ||
10         (strcmp(newObj->adminPassword, currObj->adminPassword)) ||
11         (strcmp(newObj->supportPassword, currObj->supportPassword)) ||
12         (strcmp(newObj->userPassword, currObj->userPassword)))
13     {
14         ret = rut_createLoginCfg(newObj->adminPassword,
15                                newObj->supportPassword,
16                                newObj->userPassword);
17
18
19     /* the rest of this function is not relevant to this example so is omitted. */
20 }
```

Line 9 detects the bootup case, when `newObj` is not NULL and `currObj` is NULL. Since `LoginCfg` is a type 0 object, it will never be called via `cmsObj_addObject()/cmsPh1_addObjInstance()`.

Lines 10, 11, and 12 detect modification of the existing `LoginCfg` object by comparing the three writable parameters of the `loginCfg` object. If any of the passwords have changed, the previous `/etc/passwd` is simply overwritten with the new version of `/etc/passwd`.

The function `rut_createLoginCfg` on line 14 writes out `/etc/passwd` based on the input arguments.

Note that this function does not need to check for `newObj` equals NULL because `LoginCfg` is a type 0 object, so it can never be deleted, hence the RCL handler function will never be called with `newObj` equals to NULL. If you are writing a RCL handler function for a type 1 or a type 2 object, then you must take care to check for a NULL `newObj` before trying to dereference it.

## When an Object is Deleted

When a management application calls `cmsObj_deleteInstance()` or `cmsPh1_delObjInstance()`, the RCL handler function is called with the `newObj` variable set to `NULL` and the `currObj` set to the current object in the MDM. You can test for this condition with the `DELETE_EXISTING()` macro defined in `rut_util.h`. If the object has an `enable` parameter, you can also use the `DELETE_OR_DISABLE_EXISTING()` macro.

The following code snippet shows how the `rcl_lanIpIntfObject()` detects a delete of the object. (Note that disabling the object and deleting the object is handled the same way, by disabling the bridge.)

```
01 CmsRet rcl_lanIpIntfObject(_LanIpIntfObject *newObj,
02                          const _LanIpIntfObject *currObj,
03                          const InstanceIdStack *iidStack,
04                          char **errorParam __attribute__((unused)),
05                          CmsRet **errorCode __attribute__((unused)))
06 {
07
08     /* the first part of this function is not relevant to this example
09     so it is omitted. */
10     if (DELETE_OR_DISABLE_EXISTING(newObj, currObj))
11     {
12         rutLan_disableBridge(currObj->X_BROADCAST_COM_IfName);
13     }
14 }
```

## RCL Design Patterns

Design patterns are proven successful solutions to a common problem. In the context of this document the problem is how to write a RCL handler function. The following sections describe two design patterns that appear again and again in the RCL handler functions for the MDM objects. Simple RCL handler functions have been chosen as examples so that the basic structure will be easier to see and understand. RCL handler functions for more complex objects will be more complex, but they still follow the same basic patterns described below. Developers should invest the time to understanding these design patterns as they help you understand the vast majority of the RCL handler functions in the CMS. They will also help you write your own RCL handler functions. Of course, you will probably not be able to use the exact pattern that is presented here. Depending on the requirements of your object, you will have to modify the pattern to fit your needs, but the basic structure should be helpful. Ultimately, if you understand these patterns and feel they are not suitable for your object, you can write your RCL handler function in a way that is most suitable for your object. You are not required to use these design patterns.

## RCL Design Pattern 1

This design pattern is probably the most common RCL handler function design pattern. It can only be used if the object has an enable parameter. Most objects defined by the Broadband Forum have an enable parameter.

The code below is the RCL handler function for the Dynamic DNS object. Dynamic DNS is a service provided by external companies on the Internet. The Dynamic DNS service allows devices, e.g., DSL modems, to bind their public (WAN) IP addresses to a domain name. If the public IP address on the device changes, an application called ddnsd running on the modem will change the IP address to domain name mapping in the Dynamic DNS service without user intervention. If Dynamic DNS is enabled and configured on the modem, a user can access the modem using a domain name, e.g., myhomesystem.homelinux.net, instead of an IP address that may occasionally change.

```

01 CmsRet rcl_dDnsCfgObject(_DDnsCfgObject *newObj,
02                          const _DDnsCfgObject *currObj,
03                          const InstanceIdStack *iidStack,
04                          char **errorParam __attribute__((unused)),
05                          CmsRet **errorCode __attribute__((unused)))
06 {
07     CmsRet ret = CMSRET_SUCCESS;
08
09     /* add and enable dynamic dns, or enable existing dynamic dns */
10     if (ENABLE_NEW_OR_ENABLE_EXISTING(newObj, currObj))
11     {
12         cmsLog_debug("Add or enable dynamic dns entry");
13         if (!rutDDns_isAllRequiredValuesPresent(newObj))
14         {
15             return CMSRET_INVALID_ARGUMENTS;
16         }
17         if (rutDDns_isDuplicateFQDN(newObj->fullyQualifiedDomainName, iidStack))
18         {
19             return CMSRET_INVALID_PARAM_VALUE;
20         }
21         rutDDns_stop();
22         rutDDns_start(NULL);
23     }
24
25     /* edit existing dynamic dns */
26     else if (POTENTIAL_CHANGE_OF_EXISTING(newObj, currObj) &&
27             rutDDns_isValuesChanged(newObj, currObj))
28     {
29         cmsLog_debug("Edit dynamic dns entry");
30         if (!rutDDns_isAllRequiredValuesPresent(newObj))
31         {
32             return CMSRET_INVALID_ARGUMENTS;
33         }
34         if (rutDDns_isDuplicateFQDN(newObj->fullyQualifiedDomainName, iidStack))
35         {
36             return CMSRET_INVALID_PARAM_VALUE;
37         }
38         rutDDns_stop();
39         rutDDns_start(NULL);
40     }
41
42     /* remove dynamic dns, or disable existing dynamic dns */
43     else if (DELETE_OR_DISABLE_EXISTING(newObj, currObj))

```

```
44     {
45         cmsLog_debug("delete or disable dynamic dns entry");
46         rutDDns_stop();
47         if (rutDDns_getNumberOfEntries() > 1)
48         {
49             /* there are still other entries, start DDns again, but skip this
50              * entry, which is going to get deleted. */
51             rutDDns_start(iidStack);
52         }
53     }
54
55     return ret;
56 }
```

Design Pattern 1 splits the RCL handler function into three major blocks. The first major block, lines 10–23, handles the case where an object is added and enabled, and where an existing object that was previously not enabled is now enabled. The second major block, lines 26–40, handles the case where an existing object that was enabled and still is enabled has one or more of its values (not including the enable parameter) changed. The third major block, lines 43–53, handles the case where an object is deleted or an existing object is disabled. Note that these three major blocks do not map directly to the four scenarios where the RCL handler function is called.

In the first major block, notice that the RCL handler function only validates parameters and start ddnsd if the macro `ENABLE_NEW_OR_ENABLE_EXISTING()` is true. So if an object is added, but the enable parameter is set to false, this first block will not be executed. Later, if the enable parameter is set to true, then the `ENABLE_NEW_OR_ENABLE_EXISTING` macro would test true, and this block would be executed. Also note that this major block does not distinguish between being called during bootup or called as a result of `cmsObj_add()` or `cmsPh1_addObjInstance()`.

In the second major block, the function calls the `POTENTIAL_CHANGE_OF_EXISTING()` macro. This macro verifies that `newObj` and `currObj` are both not NULL. More importantly, this macro also verifies that the enable parameter in both objects are set to TRUE. If the enable parameter in either of the objects are not true, then the first major block or the third major block should be executed. On line 26, the call to `rutDDns_isValuesChanged()` detects if any of the DDNS configuration values have changed. If so, line 37 simply deletes the existing ddns configuration file and stops the ddns application, and line 38 writes out the entire ddns config file and starts the ddns application. Note this function does not care which parameter has changed. If any parameter changed, then the ddns service is restarted with the new configuration.

The third major block uses the `DELETE_OR_DISABLE_EXISTING()` macro. Note that delete and disable are treated the same, i.e., ddnsd is stopped. The code then checks if there are other DDNS entries besides the current one that is being deleted or disabled (line 44), and if so, starts ddnsd again but without the current DDNS entry, which is about to be deleted or disabled.

## RCL Design Pattern 2

This design pattern follows the four scenarios in which the RCL handler function is called. This design pattern can be used for objects that do not have an enable parameter. (The object in this example, L2BridgingEntry actually does have an enable parameter, but it is not used to determine which of the major blocks to execute.)

The code below is the RCL handler function for the L2BridgingEntry object. This object allows a management application to create an independent LAN subnet that can be used to group a set of physical LAN ports. Packets arriving on those ports can be marked for QoS purposes and/or redirected to a specific WAN interface.

```

01 CmsRet rcl_l2BridgingEntryObject(_L2BridgingEntryObject *newObj,
02                                 const _L2BridgingEntryObject *currObj,
03                                 const InstanceIdStack *iidStack __attribute__((unused)),
04                                 char **errorParam __attribute__((unused)),
05                                 CmsRet **errorCode __attribute__((unused)))
06 {
07     char bridgeIfName[BUFLen_32];
08     CmsRet ret = CMSRET_SUCCESS;
09
10     /* add new bridging entry */
11     if (newObj != NULL && currObj == NULL)
12     {
13         if (newObj->bridgeName)
14         {
15             /* don't need to do anything in this if block */
16             cmsLog_debug("bootup condition: bridgeName=%s", newObj->bridgeName);
17         }
18         else
19         {
20             /* app called cmsObj_addObject or cmsPhl_addObjInstance */
21             if ((ret = rutLan_addBridge(&(newObj->bridgeKey))) != CMSRET_SUCCESS)
22             {
23                 cmsLog_error("rutLan_addBridge failed, ret=%d", ret);
24                 return ret;
25             }
26         }
27
28         /* edit existing bridging entry */
29         else if (newObj != NULL && currObj != NULL)
30         {
31             /* the only field we care about is the enable, check for change */
32             if (newObj->bridgeEnable != currObj->bridgeEnable)
33             {
34                 sprintf(bridgeIfName, "br%d", newObj->bridgeKey);
35                 if ((ret = rutPMap_setLanBridgeEnable(bridgeIfName,
36                                                       newObj->bridgeEnable)) != CMSRET_SUCCESS)
37                 {
38                     cmsLog_error("could not set LAN bridge to enable %d, ret=%d",
39                                   newObj->bridgeEnable, ret);
40                     return ret;
41                 }
42             }
43
44             /* delete bridging entry */
45             else

```

```
46     {  
47         sprintf(bridgeIfName, "br%d", newObj->bridgeKey);  
48         rutLan_deleteBridge(bridgeIfName);  
49     }  
50  
51     return ret;  
52 }
```

This design pattern also has three major blocks, but the first major block is divided into two subblocks.

Major block 1a, lines 13–17, is executed when the RCL handler function is called on bootup. The default value of the `bridgeName` parameter is `NULL`. So if this RCL handler function was called after the MDM initialization code had loaded a config file, then `bridgeName` would have a valid value from the config file. But if the RCL handler function was called because of `cmsObj_addInstance()` or `cmsPh1_addObjInstance()`, then the `bridgeName` would still have the default value of `NULL`.

Major block 1b, lines 20–25, is executed when the RCL handler function is called due to `cmsObj_addInstance()` or `cmsPh1_addObjInstance()`. In this scenario, the RCL handler function creates a new LAN subnet by calling `rutLan_addBridge()` (line 21). Note that the RCL handler function did not call `rutLan_addBridge()` in block 1a because the LAN subnet was present in the config file and would have been enabled/created by the RCL handler function corresponding to that object.

Major block 2 handles any changes to the existing object, including changes to the `enable` parameter. (Note that in design pattern 1, the second major block does not handle changes to the `enable` parameter.) For this object, the only parameter that we need to check for changes is the `enable` parameter. If a change is detected, then this function calls `rutPMap_setLanBridgeEnable()` on line 35.

Major block 3 handles the delete case. Because of the way the if conditions are written in this design pattern, major block 3 can be a single “else” with no conditional test. This is not the case in design pattern 1, where the `DELETE_OR_DISABLE_EXISTING()` macro must test true before block 3 is executed. Also note that in this design pattern, when an object is deleted, then the corresponding bridge is deleted; but if the object is disabled, then the corresponding bridge is not deleted, only disabled. In other words, in this design pattern, the disable and delete case is handled differently. In design pattern 1, the delete case and the disable case are handled in the same way.

The astute reader may have noticed that in major block 1b, the RCL handler function called `rutLan_addBridge()` on line 21 without first checking `newObj->enable` parameter. Technically, it should pass the `enable` parameter into `rutLan_addBridge()` so that a bridge can be created in a disabled state if `newObj->enable` is false. In practice, when a `L2BridgingEntry` is created, it is immediately followed by an `enable`, so this function takes a shortcut and anticipates user behavior. If this shortcut causes problems, it can be fixed.

---

## MDM State During Add, Set, and Delete Operations

In many cases, when an object is added, modified, or deleted, the RCL handler function can execute the system actions completely within the RCL handler function itself or by calling a few helper `rut_xxx` functions. However, in some cases, an add, set, or delete of an object will trigger a complex set of actions which requires calls through many `rut_xxx` functions. Some of these `rut_xxx` functions may actually call `cmsObj_get()`, `cmsObj_getNext()`, `cmsObj_getNextInSubTree()`, etc to read the state of the MDM. Since this is happening during the middle of an add, set, or delete operation, what will the `cmsObj_xxx` functions get when they query the MDM? The next few sections describe what happens in these situations.

### During an Add Operation

During a `cmsObj_addInstance()` or `cmsPh1_addObjInstance()`, the new object instance is first inserted into the MDM. Then the RCL handler function is called. This means that if the RCL handler function or an `rut_xxx` function then calls `cmsObj_get()`, `cmsObj_getNext()`, or `cmsObj_getNextInSubTree()` for that same object, they will get the new object instance that is currently being added.

#### 4.02L.01 and 4.02L.02 Release Version Add Operations

Adding an object can sometimes cause a subtree of objects to be created. For example, if an instance of the `LANDevice.{i}` object is added, then a `LANHostConfigManagement` and `Hosts` objects are also created under that `LANDevice.{i}` object. In this situation, the top most object is added into the MDM first, and its RCL handler function is called. Then all the children objects are added (top-down) into the MDM, and their RCL handler function is called. This means that when the RCL handler function for the `LANDevice` is being called, `cmsObj_get()`, `cmsObj_getNext()`, and `cmsObj_getNextInSubTree()` will get the new `LANDevice` object, but they will not be able to get the new `LANHostConfigManagement` or `Hosts` objects because they have not been added into the MDM yet. When the RCL handler function for the `Hosts` object is called, `cmsObj_get()`, `cmsObj_getNext()`, `cmsObj_getNextInSubTree()`, and `cmsObj_getAncestor()` will be able to get the new `LANDevice` and `LANHostConfigManagement` objects because they were created prior to the `Host` object.

Starting with the 4.02L.03 release, when an object instance is added, that object instance and all of its sub-objects will be created first. Then the RCL handler function for each of those objects will be called, starting from the top. So this means that when a `LANDevice` object instance is created, and the RCL handler function for the `LANDevice` object is called, that RCL handler function will be able to get the new `LANHostConfigManagement` object under it. (This is implemented via a two-pass add algorithm in the MDM.)



**Note:** If any of the RCL handler functions returns an error, the entire created subtree will be removed from the MDM, and the RCL handler functions will get called as if there was a delete operation.



## During a Set Operation

During a `cmsObj_set()` or `cmsPh1_setParameter()`, the modified object is placed into a set-queue, which is managed by the ODL. If the RCL handler function or a `rut_xxx` function then calls `cmsObj_get()`, `cmsObj_getNext()`, `cmsObj_getNextInSubTree()`, or `cmsObj_getAncestor()` for that same object, the newly modified object will be returned.

If the RCL handler function returns a success code, then the modified object will be inserted into the MDM.

## During a Delete Operation

During `cmsObj_deleteInstance()` and `cmsPh1_deleteInstance()`, the object or objects to be deleted are not removed from the MDM, but they are marked as “delete pending”. Then the RCL handler functions for the deleted objects are called in depth first order. When the RCL handler function for an object returns, the object instance corresponding to that RCL handler function is removed from the MDM. (This is implemented via a two-pass delete algorithm in the MDM.)

If the RCL handler function or a `rut_xxx` function calls `cmsObj_getNext()` or `cmsObj_getNextInSubTree()` and the next object instance has not been deleted from the MDM but has been marked as delete pending, that object instance will be skipped over by `cmsObj_getNext()` and `cmsObj_getNextInSubTree()`. However, if the RCL handler function or a `rut_xxx` function calls `cmsObj_get()` or `cmsObj_getAncestor()`, those functions will return the object even if has been marked as delete pending. If you are in this situation and you would like to know if the object you got from `cmsObj_get()` or `cmsObj_getAncestor()` is about to be deleted, you can call `mdm_isObjectPendingDelete()`.

If you want `cmsObj_getNext()` and `cmsObj_getNextInSubTree()` to return objects that have been marked for deletion, you can set the `mdmLibCtx.hideObjectsPendingDelete` to `FALSE` before you call `cmsObj_getNext()` and `cmsObj_getNextInSubTree()`. After you are done with those calls, you should set `mdmLibCtx.hideObjectsPendingDelete` back to its original value.

---

## Other RCL Design Issues

### RCL Handler Function Required For All Objects

The Data Model Designer does not automatically generate a stub RCL handler function for you when you add a new object. So whenever you add a new object, even if it is a statistics object, you will have to write a RCL handler function for it.

In the case of a statistics object, the RCL handler function can simply return `CMSRET_SUCCESS`. (It probably will never get called anyway.)

### Modifying Object Inside RCL Handler Function

- Modifying the `newObj` is allowed inside the RCL handler function. The modified object is put into the MDM.
- Modifying the `currObj` is not allowed.

## Use of mdmLibCtx.allocFlags

When setting a string parameter in a MDM object, use the `cmsMem_xxxFlags()` functions. Refer to `userspace/public/include/cms_mem.h`. For example:

```
cmsMem_allocFlags(bufLen, mdmLibCtx.allocFlags);  
cmsMem_strdupFlags(MDMVS_CONNECTED, mdmLibCtx.allocFlags);  
CMSMEM_REPLACE_STRING_FLAGS(obj->status, MDMVS_UP, mdmLibCtx.allocFlags);  
REPLACE_STRING_IF_NOT_EQUALS_FLAGS(obj->status, MDMVS_UP, mdmLibCtx.allocFlags);
```

Using the `cmsMem_xxxFlags()` functions instead of the `cmsMem_xxx` functions is required for RCL, STL, and RUT functions.

## Avoiding Deadlock

When inside a RCL handler function or calling an `rut_xxx` function, the calling application has acquired the global lock. So the RCL handler function and the `rut_xxx` function should be careful to not block for more than a few seconds. Also, the RCL handler function and the `rut_xxx` functions should not call any other functions or send messages to other applications (e.g., `ssk`) which would cause them to try to acquire the lock before they can send a response because those other applications cannot acquire the lock since it is already being held by the application that called the RCL handler function.

## Doing System Action

If the RCL or RUT function needs to execute a command, use `rut_doSystemAction()` instead of calling `system()` directly (see [“Executing the Command” on page 146](#)).

## Starting an Application

If the RCL handler function needs to start an application, follow the instructions in [Section 19: “Adding a New Application or Command,” on page 134](#), and also see [“Starting the Application” on page 142](#). Avoid using `fork/exec` directly in the RCL handler function.

## Section 13: Runtime Utilities

RUT functions are helper functions to the RCL and STL handler functions. RUT functions help reduce the amount of code in the RCL and STL handler functions so that the true intentions of the RCL and STL handler functions is easier to understand.

### Naming of RUT Functions

By convention, rut function names have the following format: rutXXX\_verbNoun, where,

- *XXX* is a three-letter code denoting which group of functions this RUT function belongs to. In some cases, *XXX* can be more than three letters, but in general, it should be kept short. *XXX* is also the name of the file that this function is in.
- *verb* is an action word, such as add, delete, enable, disable, get, set, move, etc. If the function just checks for a true or false condition and returns a Boolean, verb should be “is”.
- *Noun* is the subject of the action/verb. In cases where the subject of the action is completely obvious, it may be omitted.

Below are some good examples of RUT function names:

- rutLan\_addSubnet()—this function is in rut\_lan.c
- rutLan\_deleteSubnet()
- rutRoute\_addEntry()
- rutRoute\_deleteEntry()
- rutPMap\_getAvailableInterfaceByRef()
- rutPMap\_isLanInterfaceFilter()—this function returns a Boolean value.

Try to avoid these function names:

- rut\_getCurrentDns()—no hint about which file this function is in.
- rutLan\_staticIPLease()—not obvious what is being done to the staticIPLease.

### Sharing RUT Functions with DAL

In some cases, both the DAL and the RUT applications need to use the same function. The recommended new approach is to create a function in the QDM library (see [Section 11: “Query Data Model \(QDM\) Library,” on page 86](#)). If the function cannot be implemented in QDM, the old approach is to implement the function in the RUT then a wrapper function, which calls the RUT function, should be defined in the DAL. Refer to dalPMap\_getBridgeByKey() in userspace/private/libs/cms\_dal/dal\_pmap.c for an example.

## Section 14: System Status Layer

Each object in the data model must have a system status layer (STL) handler function associated with it. The STL handler function is responsible for gathering data, statistics, status, and other information from the system and putting that information into the MDM object. STL handler functions are located in `userspace/private/libs/cms_core/linux`.

The STL handler function for an object is called when an application calls `cmsObj_get`, `cmsObj_getNext`, `cmsObj_getNextInSubTree`, `cmsObj_getAncestorObject`, and `cmsPh1_getObjectParam`. However, the STL handler function is not called if the `ODF_NO_VALUE_UPDATE` flag is used with the `cmsObj_getxxx` functions.



**Note:** For some set operations, the ODL may do a get before doing a set.

The STL handler function may also be called when an application calls `cmsObj_clearStatistics()`; this scenario is only applicable for statistics objects.

The prototype for all STL handler functions look like this:

```
CmsRet stl_xxxObject(_xxxObject *obj, InstanceIdStack *iidStack);
```

where `xxx` is the short object name of the object.

### STL Design Patterns

The following sections describe common design patterns for STL handler functions. Even though there are four different patterns for the STL handler functions, they are generally much simpler than the RCL handler functions.

#### STL Design Pattern 1: Getting Statistics from Device Driver and Clearing Statistics

This pattern should work for all objects containing statistics and status information that can be obtained from a kernel driver. The basic structure is: if the `obj` variable is not `NULL`, do an `ioctl` or call a `devctl` API to get the latest statistics from the kernel. These statistics will be in a structure/format that is specific to that kernel driver. The STL handler function then copies the appropriate fields from the kernel statistics data structure to the MDM object. If the `obj` variable is `NULL`, then that means the application has requested a reset of the statistics via `cmsObj_clearStatistics()`. The STL handler function should make the appropriate `ioctl` or `devctl` API call to reset the statistics in the device driver, if one exists.

Refer to `stl_wanPppConnStatsObject()` in `userspace/private/libs/cms_core/linux/stl_wan.c` as an example.

## STL Design Pattern 2: Getting One Time Parameters

Some objects in the MDM contain status parameters that do not change once the system has booted. The STL handler function only needs to fill in these parameters once. If the STL handler function for that object is called again, it should have logic to detect if the parameter has been filled in already, and avoid doing the same work a second or third time.

The following example code is from the STL handler function for the DeviceInfo object. DeviceInfo contains a string parameter called `additionalHardwareVersion`, which is NULL when the system boots up. The STL handler function puts the `boardId` into this parameter. Since the `boardId` does not change, once the STL handler function has filled in the parameter, it will not have to do it again (until the next reboot, since the value of the parameter is not saved in the config file.)

```
01 CmsRet stl_igdDeviceInfoObject(_IGDDeviceInfoObject *obj,
02                               const InstanceIdStack *iidStack __attribute__((unused)))
03 {
04     CmsRet ret=CMSRET_SUCCESS_OBJECT_UNCHANGED;
05
06     if (obj->additionalHardwareVersion == NULL)
07     {
08         char boardId[BUFLen_64]={0};
09
10         if ((ret = devCtl_boardIoctl(BOARD_IOCTL_GET_ID, 0, boardId, sizeof(boardId),
11                                     0, NULL)) != CMSRET_SUCCESS)
12         {
13             cmsLog_error("could not get board id");
14             snprintf(finalString, sizeof(finalString), "BoardId=unknown");
15             ret = CMSRET_SUCCESS;
16         }
17         else
18         {
19             snprintf(finalString, sizeof(finalString), "BoardId=%s", boardId);
20         }
21         obj->additionalHardwareVersion = cmsMem_strdupFlags
22         (finalString, mdmLibCtx.allocFlags);
23     }
24
25     /* the rest of this function basically does the same thing as
26        additionalHardwareVersion for the other parameters in this object.
27        That code is omitted from this example. */
28
29     return ret;
30 }
```

Note that on line 4, the return value is initialized to `CMSRET_SUCCESS_OBJECT_UNCHANGED`. On line 6, if `additionalHardware` is NULL, then lines 8–21 sets the parameter. The return variable is also set `CMSRET_SUCCESS`. On subsequent calls to this function, `additionalHardwareVersion` will not be NULL, so most of the code will be skipped and we return `CMSRET_OBJECT_UNCHANGED` in line 29. A return value of `CMSRET_OBJECT_UNCHANGED` allows the ODL to do less work if the object has not changed. The ODL will convert this return value to `CMSRET_SUCCESS` and return that to the caller of `cmsObj_get()` and their variants.

Note on line 21 that `cmsMem_strdupFlags()` was used to set a string in the `obj->additionalHardwareVersion` parameter. Just like in the RCL handler function, whenever a STL handler function sets a string variable inside an object, it must use a function or macro which takes a flags argument and pass in the `mdmLibCtx.allocFlags`.

## STL Design Pattern 3: Getting Status from Other Applications

Some objects have status parameters that reflect the status of something that is only known by another application (and not by a kernel device driver). For example, in the `LANDevice.{i}.Hosts.Host.{i}`, there is a `LeaseTimeRemaining` parameter. Only `dhcpd` knows how much time is remaining for each of its leases. To handle this situation, the STL handler function needs to send a message to `dhcpd` to ask it for the lease time remaining, and when `dhcpd` responds to the request, the STL handler function can fill in the object.

Refer to `stl_lanHostEntryObject()` in `userspace/private/libs/cms_core/linux/stl_lan.c` for an example.

## STL Design Pattern 4: STL Handler Functions for Configuration Objects

Some objects have only configuration parameters and no status or statistics parameters. Even these objects are required to have a STL handler function because every object in the MDM must have a STL handler function. So when an application does a `cmsObj_get()` or one of its variants on an object, and the STL handler function for that object is called, the STL handler function can simply return `CMSRET_SUCCESS_OBJECT_UNCHANGED`. (The current settings of the configuration parameters are already supplied by the MDM, so the STL handler function does not need to get it from the MDM and update the object with those values.)

The example code below shows the STL handler function for the `X_BROADCOM_COM_LoginCfg` variable, which contains only configuration parameters and no statistics or status parameters.

```
01 CmsRet stl_loginCfgObject(_loginCfgObject *obj __attribute__((unused)),
02                          const InstanceIdStack *iidStack __attribute__((unused)))
03 {
04     /* DONE. This handler function does not need to do anything. */
05     return CMSRET_SUCCESS_OBJECT_UNCHANGED;
06 }
```

Note that on line 3, there is a comment that says `DONE`. This indicates a developer has looked at this function and decided that nothing needs to be done in the function. When CMS development began, all STL handler functions contains no code and simply returned `CMSRET_SUCCESS_OBJECT_UNCHANGED`. As development progresses, they should either be filled in with code, or be marked as `DONE`. So if a STL handler function does not have a `DONE` comment and does not contain any code, it means that function has not been reviewed by a developer and may need to be implemented.

When a STL handler function returns `CMSRET_SUCCESS_OBJECT_UNCHANGED`, that return value will get converted to `CMSRET_SUCCESS` by the ODL. So applications which are calling `cmsObj_get()` or one of its variants will only need to check for `CMSRET_SUCCESS` as the successful return value, and not both `CMSRET_SUCCESS` and `CMSRET_SUCCESS_OBJECT_UNCHANGED`.

## Section 15: Messaging

CMS provides a simple and easy-to-use messaging service to userspace applications. For applications that do not access the MDM, the messaging service is optional but is recommended. For applications that access the MDM, the messaging service is mandatory. In CMS, the messaging service is implemented with Unix Domain sockets.

Messaging code can be found in three major areas of CMS:

- In the RCL/STL/RUT functions
- In CMS applications
- In `smd`, which acts as the message router and event distributor for the CMS system

Many CMS developers may only use the messaging subsystem in the context of an RCL/STL/RUT function. In this context, knowledge of only a subset of the full CMS messaging capabilities is needed. Some CMS developers who are porting an application or writing a new application will need to know how to initialize the messaging subsystem and process incoming messages. For this work, knowledge of the full CMS messaging capabilities is needed. Very few CMS developers will need to modify the message routing and event distribution code in `smd` because that code is generic and should not need to be modified.

The rest of this section will be divided into six topics:

- A general introduction to the CMS Messaging API
- Using the CMS Messaging API in the context of the RCL/STL/RUT function
- Using the CMS Messaging API in the context of an application or command
- Useful messages
- Defining new messages in a modular way
- CMS message processing in `smd`

### CMS Messaging API

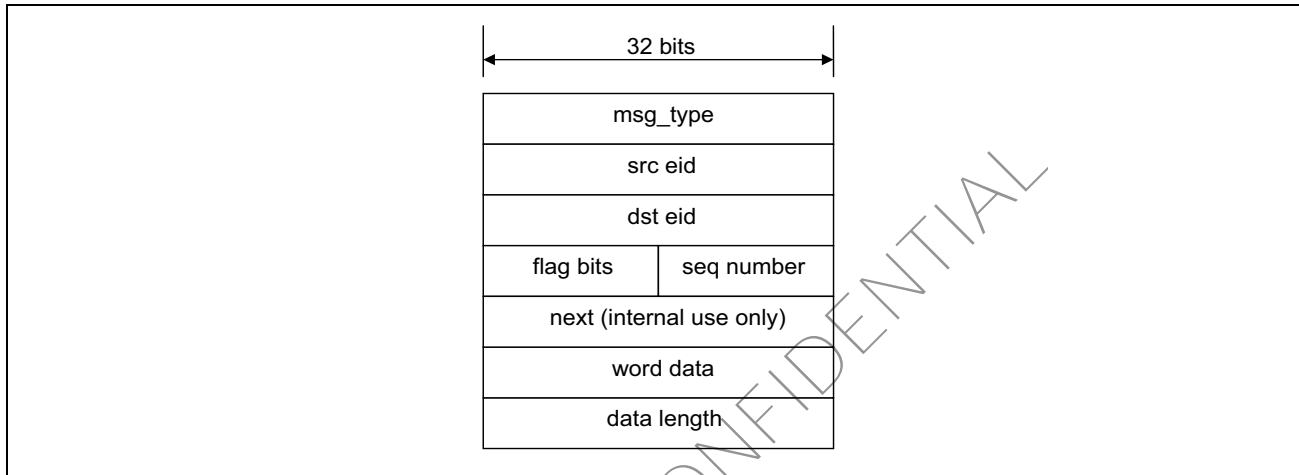
The CMS Messaging API is defined in `userspace/public/include/cms_msg.h`. Most applications should use this API and link with the messaging shared library (`libcms_msg.so`).

A CMS message can be a request message, a response message, or an event message. When an application sends out a request message, it expects a response message. When an application sends out an event message, it does not expect a response. We need to clearly distinguish between an “event” and an “event message”. An “event” is something that happens; an event is not a message. An event causes one or more event messages to be sent. (We decided not to use the word “notifications” because “notifications” have special meaning in TR69). Applications can register their interest in an event, so that when the event occurs, that application will receive an event message. Typically, an application which detects an event sends an event message to `smd`. The message handler in `smd` then sends a copy of that initial event message to all applications which have previously registered their interest in that event.

For networking-related events, such as link up/down, acquired IP address, and dhcp assigned address, the event message is first sent to ssk. Ssk then sends the event message to smd, which distributes that event message to interested applications.

Each message has a fixed sized header and an optional variable length data section. The message header contains message type, source and destination, flag bits, and some other fields. The content of the optional variable length data field depends on the message type; many messages do not have a variable length data field, just the fixed length header. The message header is shown in [Figure 11 on page 104](#).

**Figure 11: CMS Message Header**



The various message types are defined in `userspace/public/include/cms_msg.h`. Some examples of message types are `CMS_MSG_WAN_LINK_UP` (an event message) and `CMS_MSG_START_APP` (a request/response message).

The CMS Entity IDs (EIDs), which go into the source and destination fields, are defined in `userspace/public/include/cms_eid.h`. Some examples of EIDs are `EID_SMD`, `EID_SSK`, and `EID_PPP`.

In most cases, the EID is sufficient to uniquely identify the application that you want to send the message to. However, some applications may have multiple instances running on the system at the same time. For example, there will be one PPP daemon running for each PPPoE connection on the system. When sending a message to these Multiple Instance applications, you must include the PID of the application in the top 16 bits of the `dst eid` field and the EID in the lower 16 bits. A macro called `MAKE_SPECIFIC_EID` is defined in `cms_eid.h` that helps you do this. Note that you must include the PID in the `dst eid` field when sending to a multiple instance application even if you know there is currently only one instance of that application running. If you try to send a message to an application that can have multiple instances and you do not include the PID in the top 16 bits of the `dst eid` field, `smd` (which is responsible for routing the message) will return an error.

If an application that can have multiple instances is sending a message or is responding to a message, it must also include its PID in the top 16 bits of the `src eid` field. Again, the `MAKE_SPECIFIC_EID` macro can be used to concatenate the PID and EID.

The `flag bits` are explained in the `cms_msg.h` file. The event, request, response, and `bounceIfNotRunning` bits can be set by any application. The `requeue` bit is for internal use only.

The sequence number field is not currently used, and is reserved for internal use.



The next field is also reserved for internal use.

The word `data` field allows senders to include 4 bytes of arbitrary data with their message. This allows a small amount of data to be passed with each message without having to define the variable length data area.

If the message includes a variable length data section, the `data length` field indicates its length. If the `data length` field is 0, that means there is no variable length data associated with this message.

Applications that use the CMS Messaging API must first initialize a message handle with `cmsMsg_initWithFlags()`. Refer to the comment block above `cmsMsg_initWithFlags` in `userspace/public/include/cms_msg.h` for details on the arguments. `cmsMsg_initWithFlags()` was introduced in 4.14L.01. The previous function, `cmsMsg_init()` still works but has higher processing and memory overhead. The code snippet below demonstrates declaring and initializing a message handle:

```
01 #include "cms_msg.h"
02
03 void *msgHandle=NULL;
04 const CmsEntityId myEid=EID_SSK;
05
06 int main(int argc, char **argv)
07 {
08 cmsMsg_initWithFlags(myEid, 0, &msgHandle);
```

The `msgHandle` is typically a global variable that can be used by all code in the application.

Applications that access the MDM must also pass its message handle into the MDM using the `cmsMdm_initWithAcc()` API call. Refer to `userspace/private/include/cms_mdm.h`.

Here are some of the most commonly used functions from the CMS Messaging API:

```
cmsMsg_sendAndGetReply(void *msgHandle, const CmsMsgHeader *buf);
cmsMsg_sendAndGetReplyWithTimeout(void *msgHandle, const CmsMsgHeader *buf, UINT32
timeoutInMilliseconds);
cmsMsg_sendAndGetReplyBuf(void *msgHandle, const CmsMsgHeader *buf, CmsMsgHeader **replyBuf);
cmsMsg_sendAndGetReplyBufWithTimeout(void *msgHandle, const CmsMsgHeader *buf, CmsMsgHeader
**replyBuf, UINT32 timeoutInMilliseconds);
cmsMsg_send(void *msgHandle, const CmsMsgHeader *buf);
cmsMsg_receive(void *msgHandle, CmsMsgHeader **buf);
cmsMsg_receiveWithTimeout(void *msgHandle, CmsMsgHeader **buf, UINT32 timeoutInMilliseconds);
```

Refer to the `cms_msg.h` for detailed documentation for these API functions. Here are some general comments about the functions:

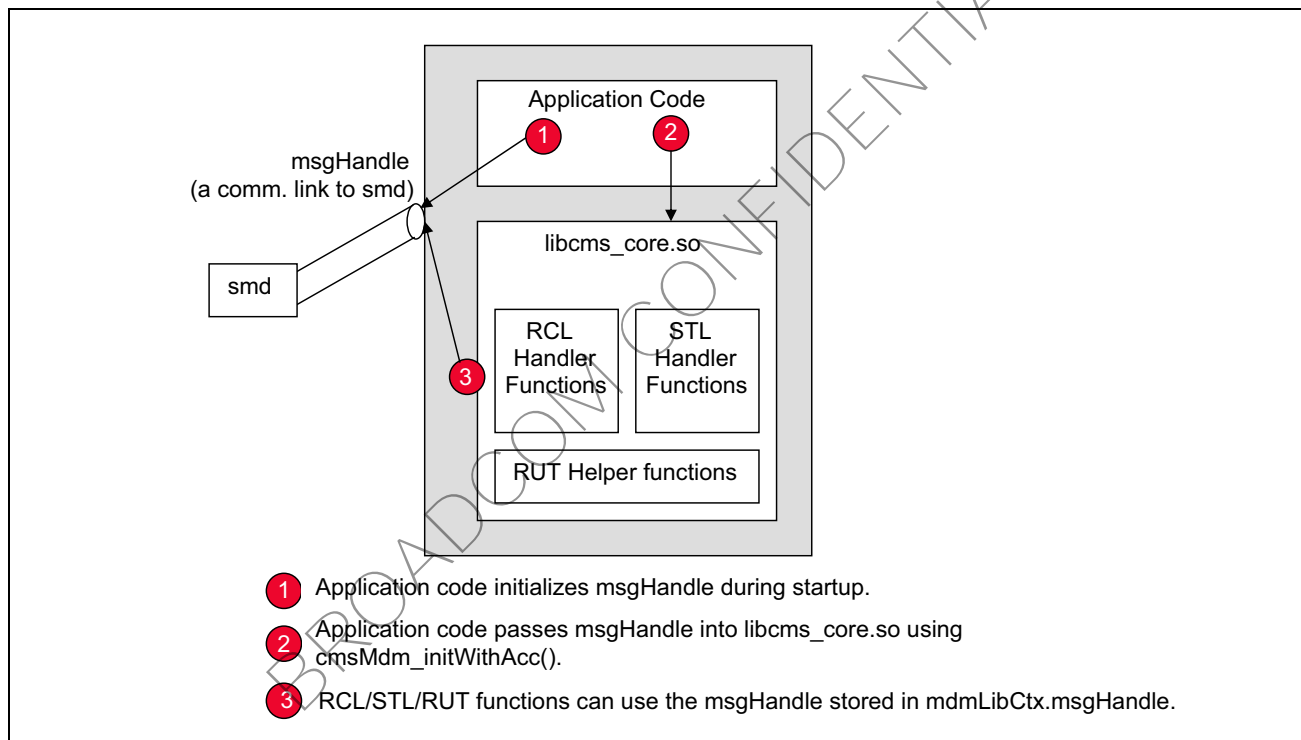
- In all the send message variants, even though the second argument is declared as a `CmsMsgHeader`, the buffer can point to a CMS Message Header followed by variable length data.
- In all the send message variants, the caller is responsible for freeing the message buffer that was sent (if it was allocated from the heap.)
- In `cmsMsg_sendAndGetReply()` and `cmsMsg_sendAndGetReplyWithTimeout()`, the caller is allowed to send a request message with just a header or a header plus variable length data. However, the response message can only contain the fixed length header. The word `data` field in the response message is the return value of `cmsMsg_sendAndGetReply()` and `cmsMsg_sendAndGetReplyWithTimeout()`.
- In `cmsMsg_receive()`, `cmsMsg_sendAndGetReplyBuf()`, and `cmsMsg_sendAndGetReplyBufWithTimeout()`, the messaging API will allocate memory from the heap to hold the received message. The caller is responsible for freeing the received message.

## CMS Messaging in RCL/STL/RUT

RCL/STL/RUT functions may need to send and receive messages to/from another application as part of their system actions. For example, when a dynamic DNS entry is configured, the RCL handler function for the dynamic DNS object will need to send a message to `smd` requesting the launch of the dynamic DNS daemon (`ddnsd`). Another example is when `httpd` does a read of a statistics object that is maintained by `dhcpd`. The STL handler function for that statistics object is called, and that STL handler function will send a request message to `dhcpd` asking for the latest statistics. `dhcpd` then sends a response message containing the latest statistics, which is used by the STL handler function to update the MDM object.

When sending and receiving messages, the RCL/STL/RUT functions use the message handle that was initialized by the application and passed into `libcms_core.so` using `cmsMdm_initWithAcc()`. The relationships between the application, the message handle, and the RCL/STL/RUT functions are shown in [Figure 12](#).

**Figure 12: Application, Message Handle, and RCL/STL/RUT Functions**



There are two ways to create a message for sending: declare the message on the stack or allocate the message from the heap. If the message is declared on the stack, you can declare just the fixed length header or the fixed length header plus a variable length data section (if the length of the variable length data section is known at compile time.) If you need to send a message with a variable length data section, and the length of that variable length data section is not known at compile time, then you must allocate the message from the heap at runtime. If the message was allocated from the heap, the sender is responsible for freeing the message after the message has been sent.

The sections below demonstrate the various methods of allocating a message for sending. Note that even though these code examples appear in the section discussing messaging in the RCL/STL/RUT functions, they can also be used in application code.

## Declaring Message Header on the Stack

When sending a message that just has the fixed length message header, the best way is to allocate the message on the stack. Then the code just needs to fill in the message header and send the message. The code snippet below from `rut_updateLogLevel()` shows how this is done.

```
01 void rut_updateLogLevel(CmsEntityId destEid, const char *logLevel)
02 {
03     CmsMsgHeader msg = EMPTY_MSG_HEADER;
04     CmsRet ret;
05
06     msg.type = CMS_MSG_SET_LOG_LEVEL;
07     msg.src = mdmLibCtx.eid;
08     msg.dst = destEid;
09     msg.flags_request = 1;
10     msg.flags_bounceIfNotRunning = 1;
11     msg.wordData = cmsUtl_logLevelStringToEnum(logLevel);
12
13     ret = cmsMsg_sendAndGetReply(mdmLibCtx.msgHandle, &msg);
```

The declaration and use of the `msg` variable should be clear from the example. Line 10 demonstrates the use of the `bounceIfNotRunning` flag. By setting this flag bit to 1, the sender is telling `smd` to not launch the application (specified by `destEid`) if it is not already running. If the `bounceIfNotRunning` flag is not set, then `smd` will launch the application so that it can receive this message. Note on line 11, this function is utilizing the `wordData` field to send four bytes of information to the receiver. On line 13, the return value from `cmsMsg_sendAndGetReply` is the `wordData` field of the response message.

Because the fixed message header was declared on the stack, it does not need to be freed after `cmsMsg_sendAndGetReply()`.

## Declaring Message Header Plus Variable Length Data Section on the Stack

The following code snippet shows how to declare and fill out a message that has the fixed length header plus a variable length data section. The length of the data section is a fixed data structure (`WatchedWanConnection`), so its size is known at compile time, therefore, we can allocate a buffer to hold both the header and the data section on the stack.

```
01 void rutWan_sendConnectionUpdateMsg(MdmObjectId oid, const InstanceIdStack *iidStack,
02                                     UB00L8 isAddPvc, UB00L8 isStatic)
03 {
04     char buf[sizeof(CmsMsgHeader) + sizeof(WatchedWanConnection)]={0};
05     CmsMsgHeader *msg = (CmsMsgHeader *) buf;
06     WatchedWanConnection *info = (WatchedWanConnection *) &(buf[sizeof(CmsMsgHeader)]);
07     CmsRet ret;
08
09     msg->type = CMS_MSG_WATCH_WAN_CONNECTION;
10     msg->src = mdmLibCtx.eid;
```

```
11  msg->dst = EID_SSK;
12  msg->flags_request = 1;
13  msg->dataLength = sizeof(WatchedWanConnection);
14
15  /* now fill in the data section */
16  info->oid = oid;
17  info->iidStack = *iidStack;
18  info->isAdd = isAddPvc;
19  info->isStatic = isStatic;
20
21  /*
22   * Special case: when sending from RCL/STL/RUT to SSK we cannot expect
23   * a reply message because during bootup initialization, ssk is initializing
24   * the MDM, so ssk is the application that is executing this RCL/STL/RUT
25   * function, so it won't be able to respond to a message sent from this function.
26   */
27  ret = cmsMsg_send(mdmlibCtx.msgHandle, &msg);
28  return;
29 }
```

On line 4, we allocate a buffer that is big enough for the message header and the data section and initialize the entire buffer with zeros. On line 5, we set the message header pointer to the beginning of the buffer. On line 6, we set the pointer to the data section. The data section in the buffer starts after the `CmsMsgHeader`, so we do:

```
&(buf[sizeof(CmsMsgHeader)]).
```

Then we cast that pointer to the appropriate type:

```
(WatchedWanConnection *)
```

This example also illustrates a limitation on sending messages to ssk from the RCL/STL/RUT functions. During system boot, ssk is the application that is responsible for initializing the MDM. During MDM initialization, the RCL/STL/RUT functions will get called for every object instance in the system. If the RCL/STL/RUT functions send a message to ssk, these functions must not block waiting for a response from ssk because ssk is the one that is executing the RCL/STL/RUT functions. So as this example shows, when sending to ssk inside an RCL/STL/RUT function, we are breaking the semantics of the CMS messaging API somewhat by marking the message as a request, but only doing a send and not waiting for a response. The code in ssk also will not send a response message to this request message.

Because the message buffer was declared on the stack, it does not need to be freed after `cmsMsg_send()`.

## Allocating Message Header Plus Variable Length Data Section from the Heap

If the message contains a variable length data section, and the length of the data section is not known at compile time, then the message must be allocated from the heap. If the message is allocated from the heap, it must be freed after the call to `cmsMsg_sendXXX`. The code snippet below demonstrates this scenario.

```

01 UINT32 rut_sendMsgToSmd(CmsMsgType msgType, UINT32 wordData,
02                         void *msgData, UINT32 msgDataLen)
03 {
04     UINT32 ret;
05     CmsMsgHeader *msg;
06     char *data;
07     void *msgBuf;
08
09     if (msgData != NULL && msgDataLen != 0) {
10         msgBuf = cmsMem_alloc(sizeof(CmsMsgHeader) + msgDataLen, ALLOC_ZEROIZE);
11     } else {
12         msgBuf = cmsMem_alloc(sizeof(CmsMsgHeader), ALLOC_ZEROIZE);
13     }
14
15     msg = (CmsMsgHeader *) msgBuf;
16     msg->type = msgType;
17     msg->src = mdmLibCtx.eid;
18     msg->dst = EID_SMD;
19     msg->flags_request = 1;
20     msg->wordData = wordData;
21
22
23     if (msgData != NULL && msgDataLen != 0) {
24         data = (char *) (msg + 1);
25         msg->dataLength = msgDataLen;
26         memcpy(data, (char *)msgData, msgDataLen);
27     }
28
29     ret = cmsMsg_sendAndGetReply(mdmLibCtx.msgHandle, msg);
30
31     CMSMEM_FREE_BUF_AND_NULL_PTR(msgBuf);
32
33     return ret;
34 }

```

This function is able to send a message that does not have a variable length data section or a message that does have a variable length data section. Since we are most interested in the latter case, we can assume that `msgData` is not `NULL` and `msgDataLen > 0`. On line 10, we allocate a buffer big enough to hold the message header plus the variable length data. Line 15 creates a pointer of type `CmsMsgHeader *`. Lines 16–20 fills in the message header. Line 24 creates a pointer to the start of the variable length data section, which starts right after the message header (in the C programming language, we can get to this point by simply incrementing the `msg` pointer by 1.) Line 26 copies the variable length data passed in by the caller into the message buffer.

Note that on line 29, the return value of `cmsMsg_sendAndGetReply()` is declared as a `UINT32` instead of a `CmsRet` enum. This is because `cmsMsg_sendAndGetReply()` will return the word data field of the reply message, and the exact contents and word data in the reply message depends on the message type. Although in many cases the word data field of the reply message is a `CmsRet` enum, it does not have to be a `CmsRet` enum. For example, the `CMS_MSG_START_APP` and `CMS_MSG_RESTART_APP` messages will return the PID of the newly launched application.

Note also on line 29, the second argument is `msg`. Even though `msg` points to the `CmsMessageHeader`, the CMS messaging APIs will know based on the `dataLength` field that there is additional data after the header that needs to be sent.

One line 31, the `msgBuf` is freed by the caller because it was allocated from the heap.

## Scenarios to Avoid

### Unexpected Response Messages

When sending a request message and then receiving a response message from inside the RCL/STL/RUT handler function, it is possible that an unexpected message may be received while waiting for the response message. For example, suppose the STL handler function sent a request message to `dhcpd` requesting the most recent statistics for IP address leases. The STL handler function then calls `cmsMsg_receive()` to get the reply. However, before `dhcpd` can send its reply, `ssk` sends a `WAN_LINK_DOWN` event message to `smd`, and `smd` forwards that message on the communication link. The STL handler function now gets the `WAN_LINK_DOWN` message from `cmsMsg_receive()`, but it does not know how to handle this message since this message was really intended for the application code portion (not the STL handler function) of the application.

To avoid the out-of-order message race condition, it is highly recommended that you use `cmsMsg_sendAndGetReply()`, `cmsMsg_sendAndGetReplyWithTimeout()`, `cmsMsg_sendAndGetReplyBuf()`, or `cmsMsg_sendAndGetReplyBufWithTimeout()` in the RCL/STL/RUT functions. These four functions include code that tracks request message that are sent out. If a message comes in that is not a response to the current request message, these functions store the unexpected messages in an internal queue. When the expected response finally arrives, or if the timeout occurs before the expected response arrives, these four functions will requeue the unexpected messages in the communications link. So the STL handler function will get the response message it waiting for, and when the STL handler function returns, the application code will get the message that was intended for it.

If your RCL/STL/RUT functions have some special requirements that force it to use `cmsMsg_send()` and `cmsMsg_receive()` instead of one of the `sendAndGetReply` variants, you will have to implement the requeuing functionality in your code to avoid the race condition. You can follow/copy the code in `cmsMsg_sendAndGetReplyWithTimeout()`.

## Deadlock in SMD

If you define a message that is sent from a RCL/STL/RUT function to `smd`, make sure that the message does not require `smd` to acquire the lock and access the MDM. If `smd` tries to acquire the lock while trying to process a message from the RCL/STL/RUT, it will never be able to acquire the lock. This is because when the RCL/STL/RUT functions are executing, it is due to some application calling a `cmsObj_xxx` or `cmsPh1_xxx` function. But for the application to call `cmsObj_xxx` or `cmsPh1_xxx`, it must have acquired the lock. So since the application is holding the lock, and calling `cmsObj_xxx` or `cmsPh1_xxx`, which triggers the call to the RCL/STL/RUT function, `smd` can not get the lock.

In general, to avoid this deadlock scenario, `smd` should not acquire the lock nor should it access the MDM. (There is only one exception to this rule, which is when `smd` accesses the MDM to read its debug level and debug destination settings and is only done once during system startup when `smd` is the only application running. However, this access is safe because the STL handler functions for the debug settings do not send a message to `smd`.)

## Deadlock in SSK

Another scenario to avoid is for the RCL/RUT functions to send a request message to `ssk` during system startup and expect a response. Recall from [“Launched by `smd` in Stage 1” on page 142](#) that during system startup, `ssk` is the application responsible for initializing the MDM, which includes the call to `mdm_activateObjects()`. `mdm_activateObjects()` goes through every object in the system and calls their associated RCL handler function. If the RCL handler function, or its helper RUT function, then sends a request message to `ssk` during system startup and expects a response from `ssk`, the RCL/RUT function will never receive the response. This is because the RCL/STL function is being executed by `ssk`, so `ssk` cannot execute the code to receive the message and respond to it because `ssk` is currently executing the RCL/RUT function.

Ideally, RCL/RUT functions should not send request/response messages to `ssk` during system startup. If the RCL/RUT function must send a request message to `ssk` during startup (there is currently only one case of this—when the WAN connection objects send a `CMS_MSG_WATCH_WAN_CONNECTION` message to `ssk`), the RCL/RUT function should just send a request message but do not try to receive a response message. `ssk`, upon receiving the request message, should process it, but not respond to it.

---

## CMS Messaging in Applications

Most application should create a communications link with `smd`. The communications link is established by simply calling `cmsMsg_init()`. The communications link between the application and `smd` allows the application to use the messaging and event notification features of CMS. Applications which want to access the MDM are required to create a communications link with `smd`.

### Initializing and Destroying the Message Handle

As described in “[CMS Messaging API](#)” on page 103, initializing the message handle simply requires the declaration of a `msgHandle` variable and calling `cmsMsg_init()`. This should be done early in the startup code of the application.

Once the `msgHandle` has been initialized, it can be used to send and receive messages through the CMS messaging API.

Before an application exits, it should call `cmsMsg_cleanup()` with the address of the `msgHandle`, e.g., `cmsMsg_cleanup(&msgHandle);`

### Using the Message Handle in a Select Loop

Most nontrivial applications need to monitor multiple file descriptors/network sockets for events and data packets. In this case, the `select` system call is commonly used. The CMS messaging API allows applications to extract the file descriptor from the message handle for use in `select`. The code snippet below demonstrates how to extract the file descriptor from the `msgHandle` and use it in a `select` loop along with other file descriptors. (This code was adapted from `ssk.c`; however, it has been modified to make it more applicable to other applications.)

```
01 #include <sys/select.h>
02
03 SINT32 ssk_main()
04 {
05     CmsRet ret;
06     CmsMsgHeader *msg=NULL;
07     SINT32 commFd;
08     SINT32 n, maxFd;
09     fd_set readFdsMaster, readFds;
10
11     FD_ZERO(&readFdsMaster);
12
13     cmsMsg_getEventHandle(msgHandle, &commFd);
14     FD_SET(commFd, &readFdsMaster);
15     FD_SET(monitorFd, &readFdsMaster);
16
17     maxFd = (commFd > monitorFd) ? commFd : monitorFd;
18
19     while (keepLooping) {
20         readFds = readFdsMaster;
21         n = select(maxFd+1, &readFds, NULL, NULL, NULL);
22         if (n < 0) {
23             /* interrupted by a signal or something, continue */
24             continue;
```



```

25     }
26
27     if (FD_ISSET(monitorFd, &readFds))
28         processMonitorFd();
29
30     if (FD_ISSET(commFd, &readFds)) {
31         if ((ret = cmsMsg_receive(msgHandle, &msg)) != CMSRET_SUCCESS) {
32             /* error handling omitted from example */
33         }
34         /* code gets here if we successfully read in a message */
35         switch(msg->type) {
36             case CMS_MSG_DHCP_STATE_CHANGED:
37                 processDhcpStateChanged(msg);
38                 break;
39             case CMS_MSG_SET_LOG_LEVEL:
40                 cmsLog_setLevel(msg->wordData);
41                 cmsMsg_sendReply(msgHandle, msg, CMSRET_SUCCESS);
42                 break;
43             default:
44                 cmsLog_error("cannot handle msg type 0x%x", msg->type);
45                 break;
46         }
47
48         CMSMEM_FREE_BUF_AND_NULL_PTR(msg);
49     }
50 }
51
52 return 0;
53 }

```

On line 9, we declare two `fd_set` structures: a `readFdsMaster` and a `readFds`. `readFdsMaster` is used to permanently store details of the file descriptors we want to monitor while `readFds` will be the temporary storage for the file descriptors we monitor. On line 11, `readFdsMaster` is initialized to 0.

On line 13, we extract the file descriptor from the CMS message handle using the `cmsMsg_getEventHandle()` function. The file descriptor is stored in `commFd`. On line 14, we use `FD_SET` to store the `commFd` inside the `readFdsMaster`. On line 15, we store the other file descriptor that we need to monitor, `monitorFd`, into `readFdsMaster`. (`monitorFd` is used as an example in this code snippet; you do not have to set `monitorFd` in your application.) In your application, if you have more than two file descriptors that you need to monitor, you would use `FD_SET` to store those file descriptors into `readFdsMaster`.

On line 17, we calculate the maximum file descriptor number that we have in the `readFdsMaster`. Since in this example, we only have two file descriptors, the algorithm to determine which file descriptor is the largest is relatively simple. However, if your application has more than two file descriptors, your algorithm may be more complicated.

On line 20, the contents of the `readFdsMaster` is copied to the `readFds` variable.

On line 21, we call `select` with the `readFds` variable. Note that the first argument to `select` is the largest file descriptor number plus one (new users of `select` commonly forget to add 1 to the `maxFd`). The third argument to `select` allows you to detect if any file descriptors are ready to accept writes of data (we do not use this feature in this example). The fourth argument allows you to detect if there is an error condition on any file descriptors (we do not use this feature in this example). The fifth argument allows you to specify a timeout to `select` by passing in a pointer to a `timeval` structure. In this example, we pass in a `NULL` as the fifth argument, which means `select` will block until one of the file descriptors in `readFds` has something for us to read (or until a signal is received or some other error occurs.)

If we get to line 27, that means there is something on one of the file descriptors for us to read. When `select` returns, it will set `n` to the number of file descriptors that have some activity on it, and it will modify `readFds` to have only the file descriptors that have something ready to read. We use the `FD_ISSET` macro to test if the `monitorFd` has something to read. If so, we call another function to read that file descriptor and process the event.

On line 30, we use `FD_ISSET` to test if there is something to read on the `commFd`, which is the file descriptor associated with the CMS message handle. If there is, we call `cmsMsg_receive()` to read in the message.

On line 35, we begin a switch statement based on the type of the CMS message that we read in. Lines 36–38 handles the `CMS_MSG_DHCPC_STATE_CHANGED` message. This message is an event message, so it does not require a response from `ssk`.

Lines 39–42 handle the `CMS_MSG_SET_LOG_LEVEL` message. This message is a request message, so `ssk` needs to send a response to the sender. `ssk` could build a custom response message to the sender, or it can simply call `cmsMsg_sendReply()`. This function will send a response message back to the sender of the message. The third argument to `cmsMsg_sendReply()` will be inserted into the `wordData` of the response message. This means that if the sender of the message used one of the `cmsMsg_sendAndGetReply` variants, the third argument to `cmsMsg_sendReply()` will be the return value of `cmsMsg_sendAndGetReplyXXX`.

Finally, on line 48, the message that we received on line 31 is freed. Note that we are always required to free the message that we received via `cmsMsg_receive()`, `cmsMsg_receiveWithTimeout()`, `cmsMsg_sendAndGetReplyBuf()`, and `cmsMsg_sendAndGetReplyBufWithTimeout()`. This is true even if we modified the message and sent it out again. Sending a message does not free the original message buffer.

## Registering Interest in an Event

Applications can register their interest for an event with `smd`. When an event message arrives at `smd`, `smd` will send a copy of that event message to all applications that have registered an interest in that event. If the interested application is not currently running and the application is a single-instance application (as declared in `eid.c`), `smd` will launch the application and send it the event message.

An application can register its interest for any event message which is defined in `cms_msg.h`. Additional event messages can also be defined in `cms_msg.h`. All that is required is for another application to detect that event and send the event message to `smd`. `Smd` will then send a copy of the event message to all applications that have registered an interest in that event.

Event messages may contain additional variable length data behind the message header. If an event message contains additional variable length data, that data section is also sent to all interested applications. The content of the variable length data section is specific to the event message.

Once an application has registered its interest for an event, it can check for event messages by calling `cmsMsg_receiveMsg()`, `cmsMsg_receiveMsgWithTimeout()`, or by extracting the file descriptor from the message handle and using that file descriptor in a `select()` loop as described in “CMS Messaging in RCL/STL/RUT” on page 106.

## Simple Event Registration

To register interest in an event, the application sends a `CMS_MSG_REGISTER_EVENT_INTEREST` to `smd`. The `wordData` field should contain the event message of interest. For example, if `tr69c` wants to register its interest for the ACS configuration changed event, it would do the following:

```
01 void registerForEvent()
02 {
03     CmsMsgHeader msg = EMPTY_MSG_HEADER;
04     CmsRet ret;
05
06     msg.type = CMS_MSG_REGISTER_EVENT_INTEREST;
07     msg.src = EID_TR69C;
08     msg.dst = EID_SMD;
09     msg.flags_request = 1;
10     msg.wordData = CMS_MSG_ACS_CONFIG_CHANGED;
11
12     ret = cmsMsg_sendAndGetReply(msgHandle, &msg);
13     /* error checking omitted */
14 }
```



**Note:** The above function does not actually exist in the source code base. It has been adapted from the function `registerInterestInEvent()` in `tr69c`.

If a single instance application exits without unregistering its interest for an event, then if the event occurs, `smd` will launch the application so that it can receive the message. This feature can be useful for `tr69c`, which registers its interest for the `CMS_MSG_ACS_CONFIG_CHANGED` event and then exits after 15 minutes of idleness. Later, if a user configures the ACS server settings, the RCL handler function for the ACS server object would send a `CMS_MSG_ACS_CONFIG_CHANGED` event message to `smd`, and `smd` will launch `tr69c` to receive the message.

If your application does not want to be relaunched to receive an event message, it should unregister its interest in that event using the `CMS_MSG_UNREGISTER_EVENT_INTEREST` message.

If a multiple instance application, e.g., `pppd`, registers for an event and then exits without unregistering its interest in an event, all event interest registrations for that instance of the application are automatically cancelled by `smd`.

## Conditional Event Registration

Many applications will probably be interested in whether a WAN link is up or down or whether a WAN connection is up or down. (WAN link refers to the Layer 2 link layer, and WAN connection refers to the Layer 3 IP layer. A WAN connection up means an IP address has been obtained and configured on the WAN link.)

If an application sends to smd a CMS\_MSG\_REGISTER\_EVENT\_INTEREST message with the wordData field set to CMS\_MSG\_WAN\_CONNECTION\_UP, that means it is interested in any WAN connection coming up. Sometimes, an application might be interested in a specific WAN connection. So for the WAN\_LINK\_UP, WAN\_LINK\_DOWN, WAN\_CONNECTION\_UP, and WAN\_CONNECTION\_DOWN events only, the interface name may be placed in the variable length data section of the CMS\_MSG\_REGISTER\_EVENT\_INTEREST message. In that case, smd will only send an event message to the application if the event type and the interface name matches.

The example code snippet below shows how an application would register its interest for WAN connection ppp0 getting an IP address.

```
01 void registerForConnectionUp(const char *ifName)
02 {
03     CmsMsgHeader *msg;
04     CmsRet ret;
05
06     msg = cmsMem_alloc(sizeof(CmsMsgHeader) + strlen(ifName) + 1, 0);
07     /* error checking for cmsMem_alloc failure omitted */
08
09     msg->type = CMS_MSG_REGISTER_EVENT_INTEREST;
10     msg->src = EID_TR69C;
11     msg->dst = EID_SMD;
12     msg->flags_request = 1;
13     msg->wordData = CMS_MSG_WAN_CONNECTION_UP;
14
15     /* the terminating NULL of the ifName must be included in the message */
16     msg->dataLength = strlen(ifName) + 1; /* e.g. ifName contains ppp0 */
17     strcpy((char *)(msg+1), ifName);
18
19     ret = cmsMsg_sendAndGetReply(msgHandle, msg);
20     /* error checking omitted */
21 }
```

## Requesting a Delayed Event Message

An application may also request a special event message be delivered to it at some time in the future. This is similar to registering for an event, but in this case, the event is that a certain amount of time has passed. To request a delayed event message, an application sends a CMS\_MSG\_REGISTER\_DELAYED\_MSG to smd. The message header's wordData field should contain an ID that is meaningful to the application. The message should contain a RegisterDelayedMsgBody structure in the variable length data section. This structure contains a UINT32, which is the number of milliseconds in the future. When the specified number of milliseconds has elapsed, smd will send a CMS\_MSG\_DELAYED\_MSG to the application. If the application is not running, and it is a single instance application, then smd will launch that application to receive the delayed message.

Delayed messages are once-only. If an application wants to receive a delayed message every *n* milliseconds, it must send another CMS\_MSG\_REGISTER\_DELAYED\_MSG to smd after it receives the last CMS\_MSG\_DELAYED\_MSG.

An application may have multiple delayed messages outstanding at `smd`. Each delayed message is uniquely identified by the ID that is in the `wordData` field of the `CMS_MSG_REGISTER_DELAYED_MSG` message.

The following code shows how to register for a delayed message:

```
01 void requestPeriodicInform(UINT32 interval)
02 {
03     char buf[sizeof(CmsMsgHeader) + sizeof(RegisterDelayedMsgBody)] = {0};
04     CmsMsgHeader *msg;
05     RegisterDelayedMsgBody *body;
06     CmsRet ret;
07
08     msg = (CmsMsgHeader *) buf;
09     body = (RegisterDelayedMsgBody *) (msg+1);
10
11     msg->type = CMS_MSG_REGISTER_DELAYED_MSG;
12     msg->src = EID_TR69C;
13     msg->dst = EID_SMD;
14     msg->flags_request = 1;
15     msg->wordData = PERIODIC_INFORM_TIMEOUT_ID; /* lets tr69c identify which delayed message this is. */
16
17     msg->dataLength = sizeof(RegisterDelayedMsgBody);
18     body->delayMs = interval * MSEC_IN_SEC; /* interval is in secs, but smd wants ms */
19
20     ret = cmsMsg_sendAndGetReply(msgHandle, msg);
21     /* error checking omitted */
22 }
```

Delayed messages can be cancelled by sending a `CMS_MSG_UNREGISTER_DELAYED_MSG` with the ID in the `wordData` field to `smd`.

## Some Useful Messages

This section describes and summarizes some messages that are generally useful to CMS developers. Not all the messages used in CMS are documented here. There are many messages that are only used to implement internal functionality, which are not of general interest to developers.

### CMS\_MSG\_REGISTER\_EVENT\_INTEREST/ CMS\_MSG\_UNREGISTER\_EVENT\_INTEREST

**Table 7: CMS\_MSG\_REGISTER\_EVENT\_INTEREST/CMS\_MSG\_UNREGISTER\_EVENT\_INTEREST**

Message	Description
Description	Allows applications to register or unregister an interest in an event. When an application registers an interest in an event, when the event occurs, the application will receive the event message corresponding to that event.
Event or Request/Response	Request/Response
Sent by	Any application
Send to	<code>smd</code>

**Table 7: CMS\_MSG\_REGISTER\_EVENT\_INTEREST/CMS\_MSG\_UNREGISTER\_EVENT\_INTEREST**

<b>Message</b>	<b>Description</b>
wordData	Contains the message type which represents the event that the application wants to register/unregister interest in. For example, CMS_MSG_WAN_LINK_UP.
payload data	Usually, no payload data is needed. However, for the CMS_MSG_WAN_LINK_UP, CMS_MSG_WAN_LINK_DOWN, CMS_MSG_WAN_CONNECTION_UP, and CMS_MSG_WAN_CONNECTION_DOWN event messages, the application can put a specific interface name in the payload data area, which means the application is interested in that event for the specified interface. See <a href="#">“Conditional Event Registration” on page 116</a> for a more detailed description.

## CMS\_MSG\_WAN\_LINK\_UP/CMS\_MSG\_WAN\_LINK\_DOWN

**Table 8: CMS\_MSG\_WAN\_LINK\_UP/CMS\_MSG\_WAN\_LINK\_DOWN**

<b>Message</b>	<b>Description</b>
Description	This event message is delivered to any application that has registered their interest in this event. Link refers to the Layer 2 link status.
Event or Request/Response	Event
Sent by	ssk
Send to	smd, which will forward the event message to any application that has registered an interest in this event.
wordData	None
payload data	Contains the name of the interface which had the link up/down event. See <a href="#">“Conditional Event Registration” on page 116</a> for a more detailed description.

## CMS\_MSG\_WAN\_CONNECTION\_UP/ CMS\_MSG\_WAN\_CONNECTION\_DOWN

**Table 9: CMS\_MSG\_WAN\_CONNECTION\_UP/CMS\_MSG\_WAN\_CONNECTION\_DOWN**

<b>Message</b>	<b>Description</b>
Description	This event message is delivered to any application that has registered an interest in this event. Connection refers to the Layer 3 status, i.e., whether an IP address has been obtained for this connection.
Event or Request/Response	Event
Sent by	ssk
Send to	smd, which will forward the event message to any application which as register an interest in this event.
wordData	None
payload data	Contains the name of the interface that had the link up/down event. See <a href="#">“Conditional Event Registration” on page 116</a> for a more detailed description.

## CMS\_MSG\_GET\_WAN\_LINK\_STATUS

**Table 10: CMS\_MSG\_GET\_WAN\_LINK\_STATUS**

<b>Message</b>	<b>Description</b>
Description	This message can be used by applications to query the current WAN link status. By using a message to query for the link status, the application does not need to access the MDM directly. This can be very useful if the application has a General Public License (GPL), as GPL applications are not allowed to link with libcms_core.so and access the MDM directly.
Event or Request/Response	Request/Response
Sent by	Any application
Send to	ssk
wordData	In the request message, none. In the response message, will contain a <b>WAN_LINK_xxx</b> code as defined in cms_msg.h.
payload data	The name of the interface that the application is querying.

## CMS\_MSG\_GET\_WAN\_CONN\_STATUS

**Table 11: CMS\_MSG\_GET\_WAN\_CONN\_STATUS**

<b>Message</b>	<b>Description</b>
Description	This message can be used by applications to query the current WAN connection status. By using a message to query for the connection status, the application does not need to access the MDM directly. This can be very useful if the application is GPL, since GPL applications are not allowed to link with libcms_core.so and access the MDM directly.
Event or Request/Response	Request/Response
Sent by	Any application
Send to	ssk
wordData	In the request message, none. In the response message, will contain TRUE (1) if the connection is up, or FALSE (0) if the connection is down.
payload data	The name of the interface that the application is querying.

## CMS\_MSG\_START\_APP

**Table 12: CMS\_MSG\_START\_APP**

<b>Message</b>	<b>Description</b>
Description	Sent by the RCL or RUT functions to request that an application be started.
Event or Request/Response	Request/Response
Sent by	The RCL or RUT function that is executing in the context of an application. Applications should not send this message directly from their application code because system actions should only be done in the RCL/RUT layer.
Send to	smd
wordData	In the request message, the EID of the application that should be started as defined in cms_eid.h. In the response message, will contain the PID of the application that was started, or CMS_INVALID_PID if the application could not be started.
payload data	Any string in the payload data area will be passed to the application as command line arguments.

## CMS\_MSG\_RESTART\_APP

**Table 13: CMS\_MSG\_RESTART\_APP**

<b>Message</b>	<b>Description</b>
Description	Sent by the RCL or RUT functions to request that the current instance of this application be stopped, and a new instance of this application be started.
Event or Request/Response	Request/Response
Sent by	The RCL or RUT function that is executing in the context of an application. Applications should not send this message directly from their application code because system actions should only be done in the RCL/RUT layer.
Send to	smd
wordData	In the request message, the EID of the application that should be restarted as defined in cms_eid.h. Note that if the application is a multiple instance application, the wordData field must contain both the PID of the current application and its EID, OR'ed together using the MAKE_SPECIFIC_EID macro defined in cms_eid.h. The response message will contain the PID of the application that was started or CMS_INVALID_PID, if the application could not be started.
payload data	Any string in the payload data area will be passed to the application as command line arguments.



## CMS\_MSG\_STOP\_APP

**Table 14: CMS\_MSG\_STOP\_APP**

<b>Message</b>	<b>Description</b>
Description	Sent by the RCL or RUT functions to request this application be stopped.
Event or Request/Response	Request/Response
Sent by	The RCL or RUT function that is executing in the context of an application. Applications should not send this message directly from their application code because system actions should only be done in the RCL/RUT layer.
Send to	smd
wordData	The EID of the application that should be restarted as defined in cms_eid.h. Note that if the application is a multiple instance application, the wordData field must contain both the PID of the current application and its EID, OR'ed together using the MAKE_SPECIFIC_EID macro defined in cms_eid.h.
payload data	None

## CMS\_MSG\_IS\_APP\_RUNNING

**Table 15: CMS\_MSG\_IS\_APP\_RUNNING**

<b>Message</b>	<b>Description</b>
Description	Query whether an application is currently running.
Event or Request/Response	Request/Response
Sent by	Any application
Send to	smd
wordData	In the request message, the EID of the application that is being queried as defined in cms_eid.h. Note that if the application is a multiple instance application, the wordData field must contain both the PID of the current application and its EID, OR'd together using the MAKE_SPECIFIC_EID macro defined in cms_eid.h. In the response message, contains CMSRET_SUCCESS if the application is running. CMSRET_OBJECT_NOT_FOUND if the application is not running.
payload data	None

## CMS\_MSG\_REBOOT\_SYSTEM

**Table 16: CMS\_MSG\_REBOOT\_SYSTEM**

<b>Message</b>	<b>Description</b>
Description	Request the system be rebooted
Event or Request/ Response	Request/Response
Sent by	Any application
Send to	smd
wordData	None
payload data	None

## CMS\_MSG\_SYSTEM\_BOOT

**Table 17: CMS\_MSG\_SYSTEM\_BOOT**

<b>Message</b>	<b>Description</b>
Description	This event message to sent to all applications that requested launch-on-boot in their CmsEntityInfo array entry in eid.c (See <a href="#">“Creating a CmsEntityInfo Entry” on page 136</a> for a description of the CmsEntityInfo array). This event message is needed because an application may get launched once during system bootup, then it might exit after some idle period timeout, then get relaunched to handle some other condition. This event message allows the application to distinguish why it was launched.
Event or Request/ Response	Event
Sent by	sm
Send to	Any application that requested launch-on-boot in their CmsEntityInfo array entry, and is currently being launched because of system boot.
wordData	None
payload data	None

## CMS\_MSG\_INTERNAL\_NOOP

**Table 18: CMS\_MSG\_INTERNAL\_NOOP**

<b>Message</b>	<b>Description</b>
Description	This message is used internally by the CMS Messaging subsystem to trigger select(). If your application receives this message, it should just ignore and free the message.
Event or Request/ Response	Event
Sent by	Internals of the CMS Messaging subsystem.
Send to	Any application might receive this message. If it does, it should ignore it and free the message.
wordData	None

**Table 18: CMS\_MSG\_INTERNAL\_NOOP (Cont.)**

<b>Message</b>	<b>Description</b>
payload data	None

## Defining New Messages in a Modular Way

Before the 4.14L.01 release, when a developer needed to define a new message type and perhaps also add a message body structure, the new message type and structure were added to `cms_msg.h`. This is appropriate as long as the message type is commonly used throughout the system. However, many features define message types and message body structures that are only used by the application implementing the feature and the RCL/STL/RUT functions that support that feature. In this case, it would be better to define the message types and message bodies in a separate header file that is only included by the files that need those specific messages. Defining feature specific message types and message body structures outside of `cms_msg.h` has a few advantages:

- Reduced clutter in `cms_msg.h`.
- Customers do not need to modify `cms_msg.h` to define their own message types and message body structures. This eliminates the chance for merge conflicts when customer merges a new `cms_msg.h` from a new release.
- Makes each feature more modular by allowing them to define new message types and message body structures without modifying a "core" CMS file.

There are many message types and message body structures in `cms_msg.h` that could be separated into their own header file. However, to minimize code churn, these "legacy" definitions will probably stay in `cms_msg.h`. New features and new message types and bodies should try to use the modular approach if possible.

The rest of this section describes the three steps required for defining message types and bodies in a modular way and uses a new feature, Modular Software, as an example. (It is purely coincidental that the feature used in this example is "Modular Software." The "Modular" in Modular Software is not related to the modular message type definition. Modular Software is described in a separate application note [see [Reference \[1\] on page 13.](#)])

### Step 1: Allocate a Block of Message Type Numbers

The message type is a 32-bit number. At the top of `cms_msg.h`, this 32-bit range has been divided into reserved range blocks. A partial listing of the `cms_msg.h` is shown below.

```
* system event message types are from          0x10000250-0x100007ff
* system request/response message types are from 0x10000800-0x10000fff
* Voice event messages are from                 0x10002000-0x100020ff
* Voice request/response messages are from       0x10002100-0x100021ff
* GPON OMCI request/response messages are from  0x10002200-0x100022ff
* Modular Software messages are from            0x10002500-0x100025ff
* Homeplug related messages are from            0x10002700-0x100027ff
* Customers should add messages from           0x20000000-0x2ffffff
* All other values are reserved for future use.
```

Broadcom internal developers who need to define new message types should reserve a block of message type numbers between 0x10002800 and 0x1ffffff. Since there are still a lot of numbers available, allocate big blocks of 256 numbers or more to allow for future expansion. External customers can use 0x20000000–0x2ffffff. All other message type numbers are reserved. The block of comments at the top of `cms_msg.h` listing the reserved ranges are the definitive source for the message type number allocations.

However, to protect against possible usage of a reserved block of message type numbers by another developer, you can also add the following definitions in the `CmsMsgType` enum (this is optional):

```
CMS_MSG_MODSW_BEGIN = 0x10002500, /**< start of Modular Software Message types */
CMS_MSG_MODSW_RESERVED = 0x10002501, /**< This range is reserved for Modular Software! */
CMS_MSG_MODSW_END = 0x100025FF, /** end of Modular Software Message Types */
```

## Step 2: Create Your Own Message Header File

The header file name should have the format, `cms_msg_xxx.h`, where `xxx` is the commonly used abbreviation for your feature. For example, the Modular Software feature uses "modsw" for its prefix, so its message header file is `cms_msg_modsw.h`. Note that `cms_msg.h` should not include `cms_msg_modsw.h`. Only files that need the definitions in `cms_msg_modsw.h` should include `cms_msg_modsw.h`.

In the new header file, you must define the feature-specific message types. However, you cannot define them as an enum of `CmsMsgTypes` because `CmsMsgTypes` was already used in `cms_msg.h`. Instead, define the new message types as an enum of `CmsXxxMsgType`, where `XXX` is the prefix for your feature. For example, in `cms_msg_modsw.h`, the definition looks like this:

```
typedef enum
{
    CMS_MSG_REQUEST_DU_STATE_CHANGE = 0x10002500,
    CMS_MSG_REQUEST_EU_STATE_CHANGE = 0x10002501,
} CmsModSwMsgType;
```

Any message body structure definitions should go into this file.

This header file should be placed in `userspace/public/include`.

## Step 3: Using Feature Specific Message Types

Because the feature-specific message types are defined as a different enum than `CmsMsgType`, they might cause compiler warnings or errors when used with the `CmsMsgHeader` structure. To prevent the warnings/errors, use the following patterns:

- Pattern 1: When setting the feature-specific message type into the `type` field of the `CmsMsgHeader`, cast it to a `CmsMsgType`. For example:

```
CmsMsgHeader hdr = EMPTY_MSG_HEADER;
hdr.type = (CmsMsgType) CMS_MSG_REQUEST_DU_STATE_CHANGE;
```

- Pattern 2: when an application is using a switch statement to detect which type of message it received, check for feature-specific messages first, and if there is no match, then check for the generic `CmsMsgTypes`. Refer to the code in `userspace/private/apps/linmosd/linmosd.c` (function `processCmsMsg`) for an example.

## Message Processing in SMD

Message processing in `smd` is done in `linux/oal_event.c`. The code in this file is very generic, so there is no need to change anything in this file if you are just defining additional messages. This file only needs to be modified if you are changing the way messaging works.

Event interests are managed by `smd/event_interest.c`. Delayed messages are handled in `sched_events.c`. Both of these features are also generic, so there is no need to modify them if you are just adding a new event message.

BROADCOM CONFIDENTIAL

## Section 16: Command Line Interface Library

When this document refers to the CLI, it means the CLI shared library libcms\_cli.so and the applications that use them (telnetd, sshd, consoled).

In the CMS architecture, the telnetd, sshd, and consoled management applications are responsible only for implementing the communications protocols (the SSH protocol, Telnet protocol, the serial port protocol). All code that deals with the command line CLI, the menu driven CLI, and accesses to the MDM are in the common shared library libcms\_cli.so. Very few changes in the telnetd, sshd, and consoled applications were made for CMS. The code that deals with the MDM are all in libcms\_cli.so, which is shared by telnet, sshd, and consoled.

This section is currently under development.

---

### Menu Driven CLI

This section is currently under development.

---

### Command Line CLI

This section is currently under development.

## Section 17: TR69C

In CMS, the tr69c application has changed significantly from the tr69c application in the 3.x releases.

The two main changes are:

- The tr69c application may be launched or exit independent of modem reboots.
- The tr69c application does not use an internal data model—it uses the native data model of CMS, which is based on the Broadband Forum data models (TR-098, TR-106, etc.).

These major changes are described in this section.



**Note:** Broadcom only releases a portion of the tr69c source code to its customers, therefore, the discussion of tr69c in this section only covers the source code that is released.

This section assumes the reader is already familiar with the TR-069 protocol, so it will not cover the requirements or concepts of the protocol here (see [“References” on page 13](#) for further information).

### Dynamic Launch of TR69C

In CMS, the tr69c application may be launched by `smd` while the modem is running, and the tr69c application may decide to exit if there is no work for it to do. At a later point in time, if there is more work for tr69c to do, `smd` will launch the tr69c application again. The dynamic launch and exit of tr69c saves memory in the modem because when tr69c is not running, it does not use any system memory, and it is expected that the tr69c application will not be running most of the time. However, implementing this dynamic launch feature (along with some other features) does complicate the startup code of tr69c.

The focus of this section is the code/algorithm of tr69c that is executed when tr69c is launched by `smd`. First some background concepts and features are covered. The entire startup algorithm is described at the end of this section.

## Inform Event Code

When tr69c is launched by smd, it will send an Inform message to the ACS. Inside the inform message, there is an Event Code, which indicates why tr69c is sending an inform message to the ACS. [Table 19](#) lists the various scenarios under which tr69c is launched and the event code in the inform message.

**Table 19: Inform Event Codes**

<b>Event</b>	<b>Event Code</b>
Modem Boot	1 BOOT
Configure ACS URL	0 BOOTSTRAP
Periodic Inform Timeout	2 PERIODIC
Change of active notification parameter	4 VALUE CHANGED
Connection request from ACS	6 CONNECTION REQUEST
Diagnostics completed	8 DIAGNOSTICS COMPLETED

There are other event codes defined in the TR-069 protocol. But these are the event codes relevant to the tr69c startup. During tr69c startup, a global variable called `launchEvent` is used to hold the event code that should be sent to the ACS.

## Bound Interface Name

The tr69c in CMS has been enhanced so that it can be configured to run over a particular IP interface. This configuration parameter is stored in the data model at `InternetGatewayDevice.ManagementServer.X_BROADCOM_COM_BoundIfName`. This parameter can have several values, shown in [Table 20](#).

**Table 20: BoundIfName Values**

<b>Value</b>	<b>Description</b>
Any_WAN	This means tr69c will run over the first IP WAN interface that acquires an IP address. This is the default value. It is also the same behavior as the tr69c in the 3.x releases.
Name of a specific WAN IP interface	This tells tr69c to only run over the specified WAN IP interface. In many cases, a service provider will set up a specific PVC dedicated for tr69c. Thus the service provider will want to set the BoundIfName to the WAN interface name of that PVC.
Any_LAN	This is used during development when the ACS is on a local network.
Loopback	This is used for the unit tests.

The BoundIfName parameter can be configured via TR-069 or the WebUI.



## Generating the Connection Request URL

In CMS, the connection request URL is generated by the STL handler function for the `InternetGatewayDevice.ManagementServer` object, not by the tr69c application. (Refer to `userspace/private/libs/cms_core/linux/stl_tr69c.c`). The tr69c application finds out the connection request URL by doing a read of the `InternetGatewayDevice.ManagementServer.ConnectionRequestURL` parameter. When tr69c reads this parameter, `stl_managementServerObject()` is called. `stl_managementServerObject()` will generate a `connectionRequestURL` if it has not been generated already. Depending on the setting of the `BoundIfName` parameter, it may not be possible to generate a connection request URL yet. For example, if `BoundIfName` is set to `Any_WAN` but no WAN interfaces are up. In this case, `stl_managementServerObject()` leaves the connection request URL as NULL.

If the ACS URL is changed (meaning tr69c is configured to use a different ACS) or the `BoundIfName` parameter is changed, the `connectionRequestURL` parameter is zeroed out by `rc1_managementServerObject()`. This forces `stl_managementServerObject()` to regenerate the connection request URL.

## TR69C Startup Algorithm

The tr69c startup algorithm is implemented in `userspace/private/apps/tr69c/main/main.c`. First, tr69c initializes the CMS messaging, timer, MDM, and logging facilities. The global `launchEvent` variable is set to **6 Connection Request**. This is just an initial setting. `launchEvent` may be changed as the startup processing continues.

tr69c then zeroizes a global `ACSState` structure called `acsState`. (This variable name is misleading because `acsState` holds more information than just the ACS state, but the name is retained to be consistent with the old tr69c code.) The `acsState` structure holds a lot of information about the current tr69c state as well as data from the MDM. tr69c often gets MDM data from `acsState` instead of reading it from the MDM.

Next, `retrieveTR69StatusItems()` is called. This function reads data from the Persistent Scratch Pad and stores the data into `acsState`.

Next, `updateTr69cCfgInfo()` is called. This function reads from the MDM and stores some of the values in the `acsState`. As a side effect, this function could also change the `launchEvent` global variable under two scenarios. First, if the ACS URL is changed, then the `launchEvent` is set to 0 `Bootstrap`. Second, if periodic inform was previously not enabled and is now enabled, then the `launchEvent` is set to 2 `Periodic`.

Next, `receiveBootupMessage()` is called. This function reads from the CMS message channel which connects tr69c to smd and processes any messages smd might have sent to tr69c. Unless smd launched tr69c due to a connection request from the ACS, smd will send a message to tr69c indicating why smd launched tr69c. The main purpose of this function is to set the `launchEvent` variable appropriately.

At this point, tr69c has collected all of the saved state data from the Persistent Scratch Pad, configuration data from the MDM, and message from smd.

tr69c next registers its interest for various events, such as `ACS_CONFIG_CHANGED` and `TR69_ACTIVE_NOTIFICATION`.

Next, `checkStartupPreReqs()` is called. This function determines whether the modem is in a state where tr69c can start to run. Specifically, two conditions must be met. First, the ACS URL must be defined. Second, based on the `BoundIfName` parameter, the appropriate interface must be up. If `BoundIfName` is set to `Any_WAN` and no WAN IP interfaces are up yet, then tr69c registers an interest in the any WAN connection up event. If `BoundIfName` is set to a specific WAN connection, and that WAN connection is not up yet, then tr69c registers its interest for that particular WAN connection up.

If `checkStartupPreReqs()` returns `TRUE`, then tr69c can start normal processing by sending out an `Inform` message with the event code indicated by `launchEvent`. If `checkStartupPreReqs()` returns `FALSE`, then tr69c cannot start processing yet. tr69c will exit at this point. Because tr69c has registered interest for various events, `smd` will launch it again when those events occur. At that point, tr69c will go through the entire startup algorithm again, and if on that next launch the startup prerequisites are met, tr69c will start normal processing.

---

## Use of CMS Data Model

In the 3.x releases, tr69c had an internal data model which was specified by the Broadband Forum. tr69c then had to convert reads and writes from the ACS, which uses the Broadband Forum data model, to the modem's data model, which was called the PSI.

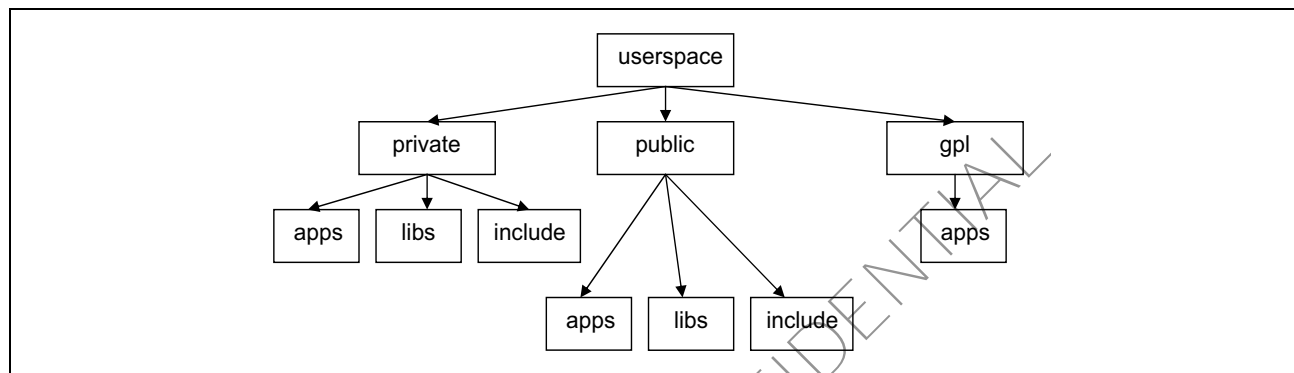
In CMS, the internal data model of the modem is the data model specified by the Broadband Forum. Therefore, the tr69c in CMS no longer has to maintain its own copy of the data model, nor does it have to translate reads and writes from the ACS from the Broadband Forum data model to the internal data model of the modem. tr69c is greatly simplified thanks to CMS's native support for the Broadband Forum data model.

Any customization to the data model should be done using the Data Model Designer. The customization will be visible to tr69c and the ACS as well as to all the other applications in the modem.

## Section 18: Directory Layout and the GPL

In the CMS directory layout, all userspace code is under a directory called userspace. Under userspace, there are three directories: private, public, and gpl. Under each of those directories, there are three more directories: apps, libs, and include. (Currently, there are no GPL libraries or header files.) [Figure 13 on page 131](#) illustrates the new directory layout.

**Figure 13: CMS Directory Layout**



All applications and libraries (and their associated header files) that are licensed under the GPL should be placed under the gpl directory. Broadcom will release all source files under the gpl directory as required by the terms of the GPL. It is suggested that Broadcom customers, if possible per applicable license terms, should release all source code in the gpl directory to their customers: the end user or consumer.

All applications and libraries (and their associated header files) with a proprietary license should be placed under the private directory. Broadcom will release most of the source files under the private directory to Broadcom customers; some files will be released as binary-only object files (for example MDM and Wireless code). Broadcom customers, in turn, must not release any source files in the private directory to their customers, the end-user, or consumer. (Some private header files might have to be released to enable compilation of public applications. This will be dealt with in the consumer release script.)

All applications and libraries (and their associated header files) that do not have a GPL or proprietary license will go under the public directory. Examples of license that should go into the public directory are non-GPL open source licenses, such as the BSD license, the OpenSSL license, the WIDE license, etc. Some CMS libraries will also go under the public directory. These CMS libraries will be made available under a GPL license but with a linking exception (the specific language is available in the header files).

## OS Independence

As much as possible, operating system (OS)-independent code and OS-dependent code should be separated. This will ease the porting of CMS to another OS.

Each library and application have a dedicated directory. The OS-dependent code is put into a subdirectory. For example, Linux-specific/dependent code is put into a subdirectory called linux. All OS-dependent functions are declared in a header file called oal.h, which is put in the OS independent directory. Using the Makefile system, these files can be compiled in the appropriate OS subdirectory and linked with the OS-independent code in the main directory.

## GPL Issues

One of the major factors motivating the reorganization of the directory structure is to make clear the relationship among the GPL code, the proprietary code, and the “glue” libraries that are needed by both the GPL and proprietary code. The GPL requires that any derivative code of the GPL code be subject to the GPL license. This means that if a proprietary library links with a GPL application in a way that is viewed as a “derivative work,” or “based on” the GPL application, then the source code of the proprietary library must be distributed under the same terms as the original GPL application. Because Broadcom wants to protect the intellectual property contained in our proprietary libraries, we do not allow GPL applications to link with proprietary libraries in a manner that results in creation of a derivative work of a GPL application.

## License of Broadcom-developed Libraries in the Public Directory

Libcms\_msg.so is placed under the public directory. The code of libcms\_msg.so was developed by Broadcom, and is made available under a GPL license but with a linking exception (the specific language is available in the header file). Thus an application can link with libcms\_msg.so and that application is not required to be licensed under the GPL.

## Applications in the Public Directory

Applications in the public directory (licensed under BSD type licenses) are under licensing terms that permit linking with other code without requiring such code to be licensed under a BSD type license. The applicable license term should be reviewed to gain an understanding of the permitted use of the non-GPL open source application. It should be noted that if a non GPL open source application links with a GPL application in a way that is viewed as a “derivative work,” or “based on” the GPL application, then the source code of the non-GPL open source application must be distributed under the same terms as the original GPL application. If possible, non-GPL open source applications should just link with public libraries.

## Summary of Linking Rules

- Private applications can link with private libraries and public libraries.
- GPL applications can link with GPL libraries (currently, there are none) and public libraries.
- GPL applications must not link with private libraries in a manner giving rise to a derivative work or modifications which then would be subject to the GPL.
- The applicable licensing terms of each application should be reviewed for permitted use.
- Do not copy source code across the gpl, public, and private directories, especially from a gpl directory to a public or private directory. For example, do not copy code from busybox in the gpl directory and put it in the cms\_core library in the private directory.

---

## The GPL and Kernel Drivers and Modules

The preceding discussion on GPL issues applies only to userspace code. The applicability of the GPL to proprietary kernel drivers and modules do not have widespread agreement in the open source community. Since CMS does not include the kernel, kernel drivers, and kernel modules, GPL issues for kernel code, kernel drivers, and kernel modules are outside the scope of this document.

BROADCOM CONFIDENTIAL

## Section 19: Adding a New Application or Command

Sometimes, to implement a feature, you will need to port or develop a new userspace executable. In CMS, userspace executables can be roughly divided into two categories: applications and commands. Applications are userspace executables that run for a relatively long time (i.e., more than several seconds). Applications do tasks continuously or on an ongoing basis. Examples of applications are `httpd`, `pppd`, `dhcpcd`, and `dnspoxy`. Commands are userspace executables that run for a relatively short period of time (i.e., a couple of milliseconds to a few seconds). Commands do a specific thing and then exit. Examples of commands are `ifconfig`, `iptables`, `atmctl`, `brctl` etc.

The build system treats applications and commands the same; they are just userspace executables. But in CMS, applications and commands are handled differently. Each section below will discuss issues involved in adding a new application or command, and will clearly indicate whether the discussion applies to both applications and commands, or to just one or the other.

### Adding the Application/Command to the Build System

First the new application or command must be added to the build system. To do this:

1. Determine if this new application or command has a proprietary, public, or gpl license. Depending on the license, place the application/command under `userspace/private/apps`, `userspace/public/apps`, or `userspace/gpl/apps`. (Even though the directory is called “apps”, it contains both applications and commands.)
2. Create an empty file called `.autodetect` in the app directory you just created. For example, if you are creating a proprietary application called `myapp`, you would create the `.autodetect` file in `userspace/private/apps/myapp`. There is no need to modify the upper layer Makefiles as they automatically detect your directory due to the presence of the `.autodetect` file and call “make” in your directory.

If the application or command comes from an existing package with an existing Makefile, you may want to just adapt the Makefile to the build system. The exact steps depend on the Makefile, but generally, you will need to modify the Makefile to use a cross compiler and to install the executable into the correct target directory. If the application or command will call CMS APIs, then it will also need to include header files from the appropriate directories and link with the appropriate libraries. Refer to the Makefiles of existing applications/commands for examples.



**Note:** Always consider licensing issues when using the CMS APIs. See [“GPL Issues” on page 132](#) for a discussion of GPL licensing issues.

If you are developing this application or command, you should use the Makefile from an existing CMS application as a template and modify it for your application or command. A good example of a Makefile for a proprietary application can be found at `userspace/private/apps/ssk/Makefile`. A good example of a Makefile for a public application can be found at `userspace/public/apps/bcm_boot_launcher`. A good example of a Makefile for a GPL application can be found at `userspace/gpl/apps/dproxy-nexgen`.

## Allocating an Entity ID

Any application or command that uses the CMS Logging service, the CMS Messaging service, or accesses the MDM must have an Entity ID. Many applications use the CMS Logging service, the CMS Messaging service, or access the MDM, so these applications have an EID assigned to them. Some applications have no interactions with CMS; they do not use the CMS Logging service, CMS Messaging service, nor do they access the MDM. These applications do not need an EID. Some commands also use the CMS Messaging service, so they must have an EID. Currently, there are no commands that access the MDM.



**Note:** Only applications with a proprietary or public license can access the MDM. Furthermore, if a public application accesses the MDM, it must not link with any GPL libraries or contain any GPL code. See [“GPL Issues” on page 132](#) for more information about licensing issues.

If your application interfaces with the user to configure the system, or implements a network management protocol such as `tr69c` or `snmpd`, it is considered a "Management Application" and must use an EID between 1 and 15. Note that previous versions of this document required any application that accesses the MDM to have an EID between 1 and 15, but this small range of EIDs was exhausted as the number of applications that accessed the MDM grew to greater than 15. So we created this more restricted category of applications called Management Applications, which is a much smaller set of applications.

If your application is not a Management Application, then its EID must be greater than 15 and should be in the range described by the comment at the top of `userspace/public/include/cms_eid.h`.

- EIDs 1–3999 are reserved for Broadcom use.
- EIDs 4000–4999 are reserved for kernel threads.
- EIDs 5000–9999 are reserved for third-party/customer use.
- All other EIDs are reserved for future allocation.

Once you have picked an EID for your application, you may add it to the `CmsEntityId` enum in `cms_eid.h`, or you may reserve a range of EIDs and put that in the comment block at the top of `cms_eid.h`. Updating the `cms_eid.h` file is optional.

## Creating a CmsEntityInfo Entry

Only applications or commands that have an Entity ID require this step.

Starting in the 4.14L.01 release, applications/commands that have an EID must fill in a CmsEntityInfo Entry. (In previous releases, there was an array of CmsEntityInfo structures in `userspace/public/libs/cms_util/eid.c`, but this array has been deleted.) In 4.14L.01, the CmsEntityInfo Entry is a text file that can be in one of two places:

- In `userspace/public/libs/cms_util/scripts/eid_bcm_*.txt`.
- In the scripts subdirectory of the application/command itself, in a file called `eid*.txt`. This file must be installed in the `/etc/cms_entity_info.d` directory of the final image. See `userspace/private/apps/linmosd/scripts/eid_bcm_linmosd.txt` and `Makefile` for an example.

The CmsEntityInfo structure contains information about the application/command, such as its EID, name, optional arguments it needs at startup, whether it accesses the MDM, etc. At system bootup time, `smd` reads in all the `eid*.txt` files in `/etc/cms_entity_info.d` so that it knows about all the CMS-aware applications and commands in the system.

A CmsEntityInfo Entry begins with the keyword "BEGIN" and ends with the keyword "END". Between the BEGIN and END keywords are one or more key/value pairs separated by the equal sign ("="). Values containing white spaces must be surrounded by double quotes. Any optional keywords not set in the CmsEntityInfo Entry get the default value. Possible keywords are listed in [Table 21](#).

**Table 21: CmsEntityInfo Entry Keywords**

Keyword	Required/Optional	Comments
eid	Required	This can be a number, e.g. 530, or an EID_xxx defined in the CmsEntityId enum in <code>cms_eid.h</code> .
accessBit	Required if eid is between 1 and 15; else optional	If accessBit is required, it is a number corresponding to the bit position of the EID (EID of 12 would be 0x0800). If accessBit is not required, it should be omitted.
name	Required	Name of the application. This name is used to find the executable in the file system.
path	Optional	Smd uses the paths defined in the PATH environment variable to find the executable, so you only need to set this if your executable is not on the standard PATH.
runArgs	Optional	If your executable requires any arguments when it is started, you can specify it here. Note that arguments that include white space must be surrounded by double quotes, e.g. "-x 500".



**Table 21: CmsEntityInfo Entry Keywords (Cont.)**

<b>Keyword</b>	<b>Required/Optional</b>	<b>Comments</b>
flags	Optional	<p>EIF flags listed in cms_eid.h. Multiple flags must be joined by " " with no spaces in between. Some examples of uses and meanings of the EIF flags:</p> <ul style="list-style-type: none"> <li>• If an application accesses the MDM (regardless of whether it is a management application), it must set the EIF_MDM flag.</li> <li>• If the application uses the CMS Messaging service, it must set the EIF_MESSAGING_CAPABLE flag.</li> <li>• If the application wants smd to open a server socket for it and launch the application when a connection comes in, it must set the EIF_SERVER flag. Additionally, if the server socket is a TCP socket, it must also set the EIF_SERVER_TCP flag.</li> <li>• If the application has multiple instances running at the same time, it must set the EIF_MULTIPLE_INSTANCES flag.</li> </ul>
backlog	Required if flags contains EIF_SERVER_TCP; else optional	Backlog value for the TCP server socket.
port	Required if flags contains EIF_SERVER; else optional	Port number to use for the server socket.
maxMemUsage	Optional	This field is currently ignored.
normalMemUsage	Optional	This field is currently ignored.
minMemUsage	Optional	This field is currently ignored.
schedPolicy	Optional	If flags includes EIF_SET_SCHED, smd sets the scheduling policy of this application/command when it is launched. Valid values are SCHED_FIFO, SCHED_RR, or SCHED_NORMAL.
schedPriority	Optional	<p>If flags include EIF_SET_SCHED, smd sets the scheduling priority of this application/command when it is launched. This field is used when schedPolicy is SCHED_RR or SCHED_FIFO. Valid values are 1–99, but use of one of the following three symbols is recommended:</p> <ul style="list-style-type: none"> <li>• BCM_RTPRIO_HIGH</li> <li>• BCM_RTPRIO_VOIPDSP</li> <li>• BCM_RTPRIO_DATA</li> </ul> <p>Refer to kernel/linux-3.4rt/include/linux/bcm_realtime.h</p>
cpuMask	Optional	If flags includes EIF_SET_CPUMASK, smd restricts this application/command to run on a specific CPU. 0x1 means run only on CPU0. 0x2 means run only on CPU1.
cpuGroupName	Optional	If flags includes EIF_SET_CGROUPS, smd places this application/command in the specified cgroup.

Note that the CmsEntityInfo Entry for all applications are always defined and loaded into memory even if the application/feature is not compiled into the system image. This represents a slight memory overhead.

You can tell `smd` to print out all the `CmsEntityInfo` Entries that it knows about by typing **dumpeid** from the CMS CLI.

The most minimal `CmsEntityInfo` entry is shown below:

```
BEGIN
eid = EID_DNSPROBE
name = dnsprobe
END
```

This entry is a bit more complicated:

```
BEGIN
eid = EID_DHCP
name = dhcp
runArgs = -f
flags = EIF_MESSAGING_CAPABLE|EIF_MULTIPLE_INSTANCES
END
```

## Patterns for Applications

This section contains common patterns for applications. You do not need to use these patterns; however, by following these patterns, you are more likely to create an application that works well in the CMS architecture. Most commands do not need to follow any of these patterns. However, if a command uses the CMS Messaging service, it should use the parts of the pattern that initialize the CMS message handle (`msgHandle` in the code below).

## Applications that Access the MDM

The following code is adapted from `userspace/private/apps/httpd/main.c`. Refer to that file for the complete code. Only the relevant pieces of code are shown here.

```
01 void *msgHandle=NULL;
02
03 SINT32 main(SINT32 argc, char *argv[])
04 {
05     SINT32 c, logLevelNum;
06     SINT32 shmId=UNINITIALIZED_SHM_ID;
07     CmsRet ret;
08     CmsLogLevel logLevel=DEFAULT_LOG_LEVEL;
09     UB00L8 useConfiguredLogLevel=TRUE;
10     UB00L8 openServerSocket=FALSE;
11
12     cmsLog_initWithName(EID_HTTPD, argv[0]);
13
14     while ((c = getopt(argc, argv, "ov:p:m:x:")) != -1)
15     {
16         switch(c)
17         {
18             case 'o':
19                 openServerSocket = TRUE;
20                 break;
```

```

21
22     case 'p':
23         serverPort = atoi(optarg);
24         break;
25
26     case 'm':
27         shmId = atoi(optarg);
28         break;
29
30     case 'v':
31         logLevelNum = atoi(optarg);
32         if (logLevelNum == 0) logLevel = LOG_LEVEL_ERR;
33         else if (logLevelNum == 1) logLevel = LOG_LEVEL_NOTICE;
34         else logLevel = LOG_LEVEL_DEBUG;
35         cmsLog_setLevel(logLevel);
36         useConfiguredLogLevel = FALSE;
37         break;
38
39     case 'x':
40         exitOnIdleTimeout = atoi(optarg);
41         break;
42
43     default:
44         cmsLog_error("bad arguments, exit");
45         usage(argv[0]);
46         exit(-1);
47 }
48 }
49
50 ret = cmsMsg_initWithFlags(EID_HTTPD, 0, &msgHandle); /* error checking omitted */
51 ret = cmsMdm_initWithAcc(EID_HTTPD, NDA_ACCESS_HTTPD, msgHandle, &shmId); /* err checking
omitted */
52 initLoggingFromConfig(useConfiguredLogLevel);
53 web_main(openServerSocket); /* run the httpd server, return when done */
54
55 /* cleanup before exit */
56 cmsMdm_cleanup();
57 cmsMsg_cleanup(&msgHandle);
58 cmsLog_cleanup();
59 return 0;
60 }

```

On line 1, we declare a void pointer for the CMS message handle (`msgHandle`). This is a global variable that needs to be shared throughout the application.

On line 12, we initialize the CMS Logging subsystem with the CMS Entity ID of this application, which is `EID_HTTPD`.

Note that in 4.14L.01, `cmsLog_initWithName()` was introduced. `cmsLog_initWithName()` takes the name of the application as the second argument and is slightly more efficient in terms of CPU and memory usage. Use of `cmsLog_initWithName()` is recommended, but `cmsLog_init()` still works.

On line 14, we call `getopt()`, which is a standard Linux command line parsing function. The third argument of `getopt()` is a string that contains all valid option letters that can be passed in to this application. If a letter is followed by a semicolon, then the option letter must also be followed by another argument. For example, the letter “v” is followed by a semicolon, so if you run `httpd` with the `-v` argument, the `-v` must be followed by another argument; in this case the other argument is the verbosity level. See the `usage()` function for details. On the other hand, the letter “o” is not followed by a semicolon. So if you start `httpd` with the `-o` option, the `-o` should not have another argument. The ordering of the option letters does not matter. Multiple options letters can be specified on the command line. In the case of `httpd`, all options are optional. Other applications may require some options to be specified on the command line (although `getopt()` does not have a mechanism to enforce this, the program must implement some logic to enforce that requirement.)

Lines 16–48 is a switch statement that processes each of the options. The “o”, “p”, and “x” options are specific to `httpd` and are not generally applicable to other applications, so they will not be discussed further.

Lines 26–28 process the “m” option, which is used to pass in the ID of the shared memory region to the application. All applications that access the MDM must support this option. Applications indicate that they will access the MDM by setting the `EIF_MDM` flag in the flags key of their `CmsEntityInfo` entry. When `smd` launches an application that accesses the MDM, `smd` prepends the `-m` option and the shared memory ID to the value of the “runArgs” key in the application’s `CmsEntityInfo` entry. The shared memory ID is stored in the `shmId` variable, which is passed into `cmsMdm_init()` on line 51.

Lines 30–37 processes the “v” option, which is used to specify the verbosity of the application. Almost all CMS applications support the `-v` option. The verbosity is set on line 35 using the `cmsLog_setLevel()` function. Also, if the `-v` option was specified, `useConfiguredLogLevel` is set to `FALSE`. This variable is used in the call to the `initLoggingFromConfig()` function on line 52. Many CMS applications have an object defined in the Data Model under `InternetGatewayDevice.X_BROADCOM_COM_AppCfg`. Those objects can store the debug level in the configuration file so that the application can have the same debug level across reboots.

`InitLoggingFromConfig()` will load the log level from the config file and use that log level only if `useConfiguredLogLevel` is set to `FALSE`. So if the application is not launched with a `-v` option, then the log level that was stored in the config file will take effect. However, if the application is launched with a `-v` option, then the log level specified in the `-v` option will have precedence over the log level in the config file.

Lines 43–46 catch unknown options.

After all the options are processed, the CMS Messaging subsystem is initialized on line 50. Note that in 4.14L.01, `cmsMsg_initWithFlags()` was introduced. `cmsMsg_initWithFlags()` takes an extra second flags argument. Refer to `cms_msg.h` for more details. `cmsMsg_initWithFlags()` is slightly more efficient in terms of CPU and memory usage. Use of `cmsMsg_initWithFlags()` is recommended, but `cmsMsg_init()` still works.

On line 51, the MDM is initialized. Note that in 4.14L.01, `cmsMdm_initWithAcc()` was introduced. `cmsMdm_initWithAcc()` takes an extra second argument that is only needed if the application is a management application. See [“Allocating an Entity ID” on page 135](#) for which applications are considered management applications. Applications that are not management applications should set this second argument to 0. Refer to `cms_mdm.h` for more details. `cmsMdm_initWithAcc()` is slightly more efficient in terms of CPU and memory usage. Use of `cmsMdm_initWithAcc()` is recommended, but `cmsMdm_init()` still works.

All applications that access the MDM must call `cmsMdm_initWithAcc()` or `cmsMdm_init()`.

On line 53, we call `web_main()`, which does all the real `httpd` processing. This function does not return until the `httpd` decides to exit. `Httpd` has an exit-on-idle timeout, which means that if it has been idle for a configured number of seconds, it will exit.

If or when `web_main()` returns, the various subsystems are cleaned up. Notice that the cleanup order is the opposite of the initialization order.

The `httpd` initialization code is a good example to use because it contains almost all the possible initialization code that a CMS application needs. The logging initialization code is not strictly required for CMS applications, but it does allow the application to log messages in a flexible and configurable way. The message initialization code is not required if the application will not access the MDM and does not need to communicate with any other application in CMS, but it is easy to implement, so it is recommended. Finally, the MDM initialization code is required if the application will access the MDM. If the application will not access the MDM, it must not call `cmsMdm_init()`.

## Applications that Do Not Access the MDM

The design pattern for applications that do not access the MDM is similar to the design pattern above, except that applications that do not access the MDM should not initialize the MDM or do anything with the `shmId`. So for applications that do not access the MDM, follow the design pattern above, with the following changes:

- Omit line 6 (`shmId` variable declaration)
- Modify line 14, delete the “m:” portion of the string.
- Omit lines 26–28 (option “m” processing)
- Omit line 51 (`cmsMdm_init`)
- Omit line 56 (`cmsMdm_cleanup`)
- Omit the lines that are specific to `httpd`, such as “o”, “p”, and “x” option processing.

## Applications Launched by SMD During Bootup

Applications that are launched by `smd` during system bootup and create a message handle will have a `CMS_MSG_SYSTEM_BOOT` event message in the message handle as soon as the application starts. The purpose of the `CMS_MSG_SYSTEM_BOOT` message is to allow the application to know whether it was started at the time of system bootup or was due to some other reason. The application can simply read this message and free it.

---

## Starting the Application

An application may be started under various scenarios, which are described below. See also [Section 7: “System Bootup,” on page 54](#).

In the CMS architecture, applications should not be started in the `cgi_XXX` code of `httpd`, nor should they be started in the DAL layer.

### From `inittab` and `bcm_boot_launcher`

As described in [Section 7: “System Bootup,” on page 54](#), after the kernel finishes booting, it creates the first userspace process, called “init.” The init process reads `inittab` and starts the `bcm_boot_launcher` process, which executes scripts in `/etc/rc3.d` that begin with the letter “S” in lexicographical order. This means an application that has no interaction or dependencies with CMS can simply install a script in `/etc/rc3.d` to start itself.

### Launched by `smd` in Stage 1

An application may create a `CmsEntityInfo` Entry (see [“Creating a CmsEntityInfo Entry” on page 136](#)) and set the `EIF_LAUNCH_IN_STAGE_1` flag in the “flags” key of its `CmsEntityInfo` Entry. `Smd` launches this application very early during `smd` startup. Note that during stage 1, the MDM has not yet been initialized and CMS applications are not yet running.

When an application is launched in stage 1 and if the application has the `EIF_MESSAGING_CAPABLE` bit set in its `CmsEntityInfo` entry, `smd` will send the application a `CMS_MSG_SYSTEM_BOOT` message indicating that it was launched because of system boot. The application is responsible for retrieving this message from its message handle and freeing it.

### Launched by `smd` in Stage 2

If an application has the `EIF_LAUNCH_ON_BOOT` bit set in its `CmsEntityInfo` entry in the `entityInfoArray`, then it will be launched by `smd` after the MDM has been initialized during the bootup sequence. If an application needs to launch during bootup, this is usually the correct time for it to launch.

Launch on boot in stage 2 can be used by applications that must wait for a particular WAN connection to be established before it can do its work. One example of such an application is `tr69c`. `Tr69c` sets the `EIF_LAUNCH_ON_BOOT` bit in its `CmsEntityInfo` entry. When the system boots, `smd` will launch `tr69c`. `Tr69c` will then register its interest in a particular WAN Connection Up event (see [“Registering Interest in an Event” on page 114](#)). `Tr69c` then exits. When the WAN connection is established, the `smd` will launch `tr69c` and send it the WAN Connection Up message.

When an application is launched during stage 2 and if the application has the `EIF_MESSAGING_CAPABLE` bit set in its `CmsEntityInfo` entry, `smd` will send the application a `CMS_MSG_SYSTEM_BOOT` message indicating that it was launched because of system boot. The application is responsible for retrieving this message from its message handle and freeing it. (This message is needed because the application needs to know whether it was launched because of system bootup or if it was launched because of an event message or delayed message.)

## Event Interest or Delayed Message

As described previously, an application may register an interest in a particular event. It may also request that a message be delivered to itself at a later time. The application may then exit. When the event occurs or when the specified later time is reached, `smd` will launch the application and deliver the message to the application. This feature is often used in conjunction with the launched by `smd` in stage 2 feature (see above).

## Applications with Server Sockets

Some applications, such as `httpd`, `snmpd`, `telnetd`, `ftpd`, have a TCP or UDP server socket that accepts new connections from remote clients across the LAN or WAN. In CMS, these applications can indicate that they have a server socket by setting the `EIF_SERVER` bit in their `CmsEntityInfo` entry. Additionally, if the server socket is a TCP socket, the `EIF_SERVER_TCP` bit and the backlog field must also be set. For both TCP and UDP sockets, the `port number` field in the `CmsEntityInfo` must also be set.

Once this information is set in the `CmsEntityInfo` entry for the application, during system bootup, `smd` will open the server socket for the application. (The application does not need to set the `EIF_LAUNCH_ON_BOOT` bit). When a client request comes into the server socket, `smd` will launch the application to process the client. The application will inherit the server socket from `smd` at file descriptor `CMS_DYNAMIC_LAUNCH_SERVER_FD`, which is defined in `userspace/public/include/cms_params.h`.

## Application Started by RCL Handler Functions

An application may also be started to implement a feature. For example, `syslogd` (a daemon that logs system events) is started if the feature is enabled in the data model. These applications are started from the RCL handler function of the data model object for that feature or an RUT function called from the RCL handler function. In the case of `syslogd`, it is started when `rcl_syslogCfgObject()` calls `rut_restarts syslogd()`, which then calls `rut_sendMsgToSmd()`.

`rut_sendMsgToSmd()` is a helper function that can be used by the RCL handler functions or the RUT functions to send messages to `smd`. It is often used to send `CMS_MSG_START_APP` or `CMS_MSG_RESTART_APP` messages to `smd`. (But it is not limited to only those messages.)

`CMS_MSG_START_APP` should only be used only if the caller does not wish to stop a currently running instance of the application first. The application's Entity ID should be provided as the second argument to `rut_sendMsgToSmd()`. (This is true even if the application can have multiple instances.) Any command line arguments that need to be provided to the program should be the third argument, and the length of the command line string plus the terminating null should be provided as the fourth argument. If the application does not need command line arguments, then the third and fourth arguments to `rut_sendMsgToSmd()` should be `NULL` and `0`, respectively. `Smd` will return the PID of the launched application in response to the `CMS_MSG_START_APP` message. If `smd` was unable to launch the application, it will return `CMS_INVALID_PID` (defined as `0` in `userspace/public/include/linux/os_defs.h`). The following lines from `rut_restarts syslogd()` demonstrate the use of `rut_sendMsgToSmd()` and the `CMS_MSG_START_APP` message. (This code snippet has been heavily modified from the original function so that the only the relevant lines are shown.)

```
01 CmsRet rut_restarts syslogd(const _SyslogCfgObject *syslogCfg)
02 {
03     char args[BUFLen_128];
04
```

```

05    snprintf(args, sizeof(args), "-L -l %d",
06              cmsUtl_syslogLevelToNum(syslogCfg->localLogLevel));
07
08    {
09        SINT32 pid;
10        pid = rut_sendMsgToSmd(CMS_MSG_START_APP, EID_SYSLOGD, args, strlen(args)+1);
11        if (pid == CMS_INVALID_PID) {
12            cmsLog_error("failed to start syslogd");
13            ret = CMSRET_INTERNAL_ERROR;
14        }
15    }
16    /* remainder of function is omitted */

```

In most cases, CMS\_MSG\_RESTART\_APP should be used to start an application because this message causes smd to stop the specified existing instance of the application before starting the new one. Use of the CMS\_MSG\_RESTART\_APP message is slightly different, depending on whether the application can have multiple instances running or not. If the application cannot have multiple instances, the use of CMS\_MSG\_RESTART\_APP is the same as CMS\_MSG\_START\_APP. However, if the application can have multiple instances, the second argument to rut\_sendMsgToSmd() is the PID of the currently running instance of the application OR'ed with the Entity ID. A macro called MAKE\_SPECIFIC\_EID has been provided in cms\_eid.h to combine the PID and the EID. The following lines from rut\_restartdhcpc() demonstrate the use of the CMS\_MSG\_RESTART\_APP message for an application that supports multiple instances.

```

01 SINT32 rutWan_restartDhcpc(const char *ifName, SINT32 previousPid)
02 {
03     char cmdLine[BUFLen_128];
04     SINT32 pid;
05     UINT32 specificEid;
06
07     specificEid = MAKE_SPECIFIC_EID(previousPid, EID_DHCPC);
08     snprintf(cmdLine, sizeof(cmdLine), "-i %s", ifName);
09     if ((pid = rut_sendMsgToSmd(CMS_MSG_RESTART_APP, specificEid,
10                               cmdLine, strlen(cmdLine)+1)) == CMS_INVALID_PID)
11     {
12         cmsLog_error("failed to start or restart dhcpc on %s", ifName);
13     }
14
15     /* remainder of function is omitted */

```

Note that during smd startup, applications with the EIF\_LAUNCH\_IN\_STAGE\_1 bit set are launched first (stage 1). Currently, only ssk is launched during stage 1. Then, as ssk is initializing the MDM, applications that are associated with various data model objects, such as the pppd from WANPPPCConnection, are launched from their RCL handler functions. After ssk has initialized the MDM, smd launches the applications with the EIF\_LAUNCH\_ON\_BOOT flag set (stage 2).



## Stopping the Application

An application may stop running and exit under a variety of scenarios, which are described below.

### Self Exit

As described in “Starting the Application” on page 142, an application may be launched on bootup, register for an event or a delayed message, and then exit.

Applications with server sockets may also exit after a certain amount of idle time.

### Applications Stopped by RCL Handler Functions

Applications that were started by RCL handler functions should also be stopped by the RCL handler function when the feature is disabled. Analogous to starting an application, stopping an application from the RCL handler function is done by sending a CMS\_MSG\_STOP\_APP message to smd. The rut\_sendMsgToSmd() can be used to send this message. The first argument to rut\_sendMsgToSmd() is the message type, CMS\_MSG\_STOP\_APP. If the application can only have a single instance running, the second argument to rut\_sendMsgToSmd() is just the EID of the application. If the application can have multiple instances running, then the second argument is the PID of the running instance OR'ed with the entity id of the application. Use the MAKE\_SPECIFIC\_EID to do the or operation. The third and fourth argument to rut\_sendMsgToSmd() should be NULL and 0, respectively. The following code snippet from rc1\_wanIpConnectionObject() shows how to stop dhcpc, which is a multiple instance application.

```
01 {
02     UINT32 specificEid;
03
04     specificEid = MAKE_SPECIFIC_EID(currObj->X_BROADCOM_COM_DhcpcPid, EID_DHCPC);
05     if ((ret = rut_sendMsgToSmd(CMS_MSG_STOP_APP, specificEid,
06                               NULL, 0)) != CMSRET_SUCCESS)
07     {
08         cmsLog_error("failed to stop dhcpc");
09     }
10
11     /* remainder of function is omitted */
```



**Note:** Unlike CMS\_MSG\_START\_APP and CMS\_MSG\_RESTART\_APP, smd returns a CmsRet enum in response to CMS\_MSG\_STOP\_APP.

Currently, smd sends a SIGTERM signal to the application when it wants an application to terminate. If the process does not exit within 5 seconds of the SIGTERM, it sends a SIGKILL (which cannot be blocked or ignored).

---

## Executing the Command

RCL, STL, and RUT functions may need to execute a command to configure the system. They can use the `rut_doSystemAction` command to execute that command.



**Note:** The `rut_doSystemAction` command will wait for the command to terminate before returning to the caller, so commands run with `rut_doSystemAction` should run and terminate in a relatively short amount of time (i.e., a few milliseconds to a few seconds, at most).

The following code snippet from `rcl_adslwan.c` demonstrates the use of `bcmSystem`.

```
01  snprintf(cmdStr, sizeof(cmdStr), "ifconfig %s up", newObj->X_BROADCOM_COM_IfName);  
02  rut_doSystemAction("rcl", cmdStr);
```

The `cmdStr` variable was declared to be a char buffer of 128 bytes at the beginning of the function.

The first argument to `rut_doSystemAction` is just an identifier for debug messages to indicate which function called `rut_doSystemAction`.

Commands should only be executed in the RCL, STL, and RUT functions. They should not be executed in the `cgi_xxx` code of `httpd` or in the DAL.

## Section 20: The CMS Utilities Library

The CMS utility library is called `libcms_util.so`. It can be used by GPL, open source, or proprietary applications.

The CMS utility library contains a collection of small but useful APIs, which are briefly described in the section below. (More details will be provided in future document revisions). Each CMS utility API has a header file in `userspace/public/include`. Applications can just include the `userspace/public/include/cms_utils.h`, which includes all of the utilities header files.

### Logging (`cms_log.h`)

The CMS Logging service provides a much better way to print debug and error messages from your application. Use of CMS logging instead of `printfs` is strongly recommended.

To use the CMS Logging service in its most basic form, follow these steps:

1. Define an Entity ID (EID) for your application or command. See [Section 19: "Adding a New Application or Command," on page 134](#) for details about selecting an EID.
2. Create a `CmsEntityInfo` entry for your application/command. See [Section 19: "Adding a New Application or Command," on page 134](#).
3. Initialize the CMS Logging service at the beginning of your application with `cmsLog_initWithName()`. See ["Patterns for Applications" on page 138](#) for an example.
4. By default, the log level is to output "error" messages only. You can change the log level with the `cmsLog_setLevel()` function. See ["Patterns for Applications" on page 138](#) for an example.
5. Instead of using `printf` in your code, write your log statements with:
  - `cmsLog_error("This is an error message!! Return code = %d", retVal);`
  - `cmsLog_notice("This is an important message, but not an error.");`
  - `cmsLog_debug("this is a debug message, arg1=%d arg2=%s arg3=0x%x\n", arg1, arg2, arg3);`

Note that all the `cmsLog_xxx` functions can take a variable number of arguments and formats, just like `printf`.

The output of the `cmsLog_debug` message above will look something like this:

```
httpd:debug:98.611:web_main:512:this is a debug message, arg1=1, arg2=hello, arg3=0x30
```

The CMS Logging service automatically prepends several useful fields to the log line separated by the colon character ":". The first field is the name of the application (`httpd`). The second field is the logging level of the line (`debug`). The third field (`98.611`) is a timestamp in the format of seconds.milliseconds. The fourth field (`web_main`) and the fifth field (`512`) are the function name and line number of the log line.

Many CMS applications have an even more sophisticated implementation of the CMS logging system that allows developers, QA staff, and field engineers to dynamically set the log level of the application (see ["Using Log Messages" on page 155](#)). This can be very useful for debugging problems on a running system.

To implement this capability, follow these steps:

1. Define a data-model object for your application. See [Section 4: “The Data Model,” on page 28](#) and [Section 5: “Data Model Designer,” on page 36](#) for instructions on how to define a new object. See `data-model/cms-dm-tr98.xml` and search for `X_BROADCASTOM_COM_AppCfg.LinmosdCfg` to see an example.
2. Define RCL and STL handler functions for your new object. See [Section 12: “Runtime Configuration Layer,” on page 87](#) for a discussion of RCL and [Section 14: “System Status Layer,” on page 100](#) for a discussion of STL. In this case, the RCL and STL functions are very simple. Refer to `userspace/private/libs/cms_core/linux/rcl_linmods.c` and `stl_linmods.c` for how to implement these RCL and STL functions.
3. In your application, write code to receive and process `CMS_MSG_SET_LOG_LEVEL`, which tells your application to set a different log level. Refer to `userspace/private/apps/linmosd/linmosd.c` for example.
4. Add code for your application in the `processLogLevelCmd` in `userspace/private/libs/cms_cli/cli_cmd.c`.

## Memory Allocation (cms\_mem.h)

This API allows you to allocate memory. It has been enhanced to also include shared memory allocations, which is needed in the RCL, STL, and RUT functions. The memory API also includes debug features, such as buffer underflow/overflow checking, poisoned frees, and memory leak detection.

Due to its debugging features, use of CMS memory allocation functions instead of `malloc` is strongly recommended. CMS Entity ID is not required for CMS memory allocation functions. There is no initialization function.

Instead of `malloc`, simply call:

```
cmsMem_alloc(num_bytes, flags);
```

where `flags` is either 0 or `ALLOC_ZEROIZE`. (There is another flag, `ALLOC_SHARED_MEM`, which must be used by functions in the RCL/STL/RUT layers. See [Section 12: “Runtime Configuration Layer,” on page 87](#) and the discussion of `mdmLibCtx.allocFlags`.)

All memory allocated by `cmsMem_alloc()` must be freed with `cmsMem_free()`. In an application, do not mix `cmsMem_alloc/cmsMem_free` with `malloc` and `free`. To avoid accidental mixing, you should use one set only.

There are other functions and macros for dealing with strings, such as:

- `cmsMem_strdup`
- `cmsMem_strdupFlags`
- `CMS_MEM_REPLACE_STRING`
- `CMS_MEM_REPLACE_STRING_FLAGS`
- `REPLACE_STRING_IF_NOT_EQUAL`
- `REPLACE_STRING_IF_NOT_EQUAL_FLAGS`

All of these functions and macros are built on top of `cmsMem_alloc` and `cmsMem_free`.

## Timestamps (cms\_tms.h)

This API allows you to get a simple monotonically increasing timestamp from the Linux kernel. The timestamp is not related to the wall clock, so it is not affected by time adjustments made by NTP or the command line. The API also provides functions to calculate elapsed time from one timestamp to another, add some amount of time to a timestamp, etc.

## Timer (cms\_tmr.h)

This is a timer library that allows you to check when a timer has expired. It requires the use of `select()`.

## Assert (cms\_ast.h)

This API allows you to put assertions in your code. During development time, if an assertion is violated, the program will be aborted immediately with a debug message. For production builds, these asserts can be disabled.

## Doubly-linked List (cms\_dlist.h)

This API is a set of macros that allow you to build doubly-linked lists. In CMS, it not necessary to implement doubly-linked list code; just use these macros as they have been tested.

## Persistent Scratch Pad (cms\_psp.h)

All applications are allowed to use a small area of the persistent Flash to store any data they might have. The scratch pad space is given on a first-come-first-served basis. The size of the scratch pad in bytes is defined by the `SP_MAX_LEN` constant, which is set to 8192.

## Base64 (cms\_base64.h)

TR-69 defines a data type called Base64, which is an ASCII printable encoding of binary data (see [“Broadband Forum Parameter Types and Settings” on page 44](#)). The functions in this API allow binary data to be converted to Base64 data and back.

## HexBinary (cms\_hexbinary.h)

Various functions for HexBinary (ascii) to binary conversions.

## **XML (cms\_xml.h)**

Various functions for XML manipulations.

## **LED (cms\_led.h)**

Various functions to control the state of the LEDs.

## **LZW Compression/Decompression (cms\_lzw.h)**

Various functions for compressing and decompressing using the LZW algorithm.

BROADCOM CONFIDENTIAL

## Section 21: Board and Device Control Libraries

The 3.x DSL modem software uses fork and exec of a command to get status from the kernel. This method causes spikes in memory consumption because a shell process must be created, then the command file is decompressed from the squashfs file system, and another process is created for the command itself.

In the CMS architecture, the core code from these useful commands (atmctl, adslctl, wlctl) are made into shared libraries. The STL handler functions, which are responsible for gathering statistics and status, will make function calls into those shared libraries instead of forking and executing a command. The original commands will still be available, but the bulk of their code will be in the shared libraries, so they will link against those shared libraries as well.

This approach of turning the internal logic of a command into a shared library cannot be applied to commands released under the GPL because any code that links with GPL code (even via a shared library) must also be released under GPL. Because the CMS STL handler functions are part of a Broadcom-proprietary library, the STL handler functions cannot link with shared libraries that fall under the GPL licence.

BROADCOM CONFIDENTIAL

# Section 22: Doxygen API Documentation

CMS uses Doxygen to generate API documentation. Typically, when a developer writes code, specially formatted comment blocks are used in the \*.h and \*.c files. Doxygen then parses these comment blocks and generates cross-referenced HTML documents.

## Writing Doxygen-Compatible Comment Blocks

### Documenting a Function

The following code is an example of how to write doxygen-compatible comments for a function prototype (from cms\_obj.h):

```
01  /** Get the next instance of the object.
02  *
03  *   This function can only be used for objects that have multiple
04  *   instances. If an object can only have one instance, use cmsObj_get().
05  *   This function just calls cmsObj_getNextInSubTree with parentIidStack = NULL.
06  *
07  *   @param oid          (IN)      The oid of the MdmObject to get.
08  *   @param iidStack     (IN/OUT)  The instance information for the current instance
09  *                                 of the object. If the iidStack is freshly
10  *                                 initialized (empty), then the first instance
11  *                                 of the object will be returned.
12  *   @param obj          (OUT)     On successful return, obj points to the next
13  *                                 MdmObject. The caller must free this object by
14  *                                 calling cmsObj_free (not cmsMem_free).
15  *   @return CmsRet enum.
16  */
17  CmsRet cmsObj_getNext(MdmObjectId oid,
18                      InstanceIdStack *iidStack,
19                      void **mdmObj);
```

Note on line 1, the comment starts with `/**` instead of the usual `/*`. This tells doxygen that this is a doxygen-compatible comment block. The comment block should start with a one or two sentence summary of the purpose of the function.

On line 2, there is an empty line. This tells doxygen that the summary has ended and that the detailed description of the function will start on the next line.

Lines 3–5 provide the detailed description of the function. The detailed description section can contain multiple sentences.

Lines 7–14 list the parameters of the function. Doxygen requires that each parameter description begins with `@param variable-name`. For example, `@param oid`. In CMS, we also add (IN), (IN/OUT), or (OUT) after the variable name to clearly specify whether the value is passed in, passed in and out, or passed out. Finally, the description of the parameter. The description can span multiple lines.



If the function has a return value, then a “@return description of the return value” should be the last thing in the comment block.

## Documenting a Structure

The following code shows how to write doxygen-compatible comments for a structure definition (from public/include/linux/os\_defs.h):

```
01 /** A structure to keep track of instance information.
02  *
03  *   External callers can treat this as an opaque handle.
04  *   Note the instance array must be of type UINT32 because
05  *   the instance id's are constantly increasing, so we
06  *   cannot save space by defining instance to be
07  *   an array of UINT8's.
08  */
09 typedef struct
10 {
11     UINT8 currentDepth;           /**< next index in the instance array
12                                   to fill. 0 means empty. */
13     UINT32 instance[MAX_MDM_INSTANCE_DEPTH]; /**< Array of instance id's */
14 } InstanceIdStack.
```

Just as with function prototypes, the comment block above a structure definition should also begin with `/**`. The first sentence or two should be a brief summary of the structure. A blank comment line (line 2) separates the brief description from the detailed description (lines 3–7). The detailed description is optional.

Each of the fields of the structure should be documented using a comment line that begins with `/**<`. The field comment can span multiple lines.

## Documenting a macro, #define, or typedef

The following is an example of how to write doxygen compatible comments for typedef (from public/include/linux/os\_defs.h). The format is the same for macros and #defines.

```
01 /** A number to identify a MdmObject (but not the specific instance of
02  *   the object.
03  *
04  *   MdmObjectId's are defined in mdm_oid.h
05  */
06 typedef UINT16 mdmObjectId
```

The comment block is the same as the function comment block. For macros, it may be possible to also specify a `@param` in the comment block.

## Documenting an Enumeration

The following code shows how to write doxygen-compatible comments for an enumeration (from cms\_mdm.h).

```

01  /*!\enum MdmParamTypes
02  *  \brief Possible types for parameters in the MDM.
03  *
04  *  The first 6 (MPT_STRING through MPT_BASE64) are from the TR-098 spec.
05  *  The next 3 (MPT_HEX_BINARY through MPT_UNSIGNED_LONG64) were introduced in
06  *  TR-106 issue 1, Admendment 2, Sept 2008.
07  *  Broadcom does not define any additional types for now.
08  */
09  typedef enum
10  {
11      MPT_STRING = 0,    /**< string. */
12      MPT_INTEGER = 1,   /**< Signed 32 bit integer */
13      MPT_UNSIGNED_INTEGER = 2, /**< Unsigned 32 bit integer */
14      MPT_BOOLEAN = 3,   /**< Either 1 (true) or 0 (false). */
15      MPT_DATE_TIME = 4, /**< string, in UTC, in ISO 8601 date-format */
16      MPT_BASE64 = 5,    /**< Base64 string representation of binary data. */
17      MPT_HEX_BINARY = 6, /**< Hex string representation of binary data */
18      MPT_LONG64 = 7,    /**< Signed 64 bit integer */
19      MPT_UNSIGNED_LONG64 = 8, /**< Unsigned 64 bit integer */
20  } MdmParamTypes;

```

Unlike the other doxygen-compatible comment blocks, the first line of the comment block for an enumeration must begin with `/*!\enum name-of-the-enum`. The second line contains the one sentence brief description of the enum, which must be marked by the `\brief` keyword. The detailed description section is like the comment block for functions. Finally, each member of the enum should be commented with `/**<`, just like the fields in a structure definition.

## Generating Documentation with Doxygen

To generate Doxygen documentation, simply `cd` into the top level directory and type **make doxygen\_docs**.

The Doxygen documentation is installed under `docs/doxygen/html`. These HTML files can be accessed directly by pointing your browser to **file:///<path-to-your-source-root>/docs/doxygen/html/files.html**.

The HTML files can also be copied to a public web server, to allow people to access a common set of documentation over the network.

If you change any header files/doxygen comment blocks, you can regenerate or update the doxygen documentation by typing **make doxygen\_docs** again.

Currently, all header files under `userspace/public/include`, `userspace/private/include`, and `userspace/private/libs/cms_core` are scanned by Doxygen. The list of directories that are scanned can be modified by changing the Makefile in `hostTools`. Look for the section dealing with Doxygen and add, modify, or delete the lines that look like:

```
sed -i 's,\(INPUT[ \t]*=\),\1 $(BUILD_DIR)/userspace/public/include,' Doxyfile;
```

## Section 23: Debugging

After your code has been written and compiled it is time to run it and debug it. This section describes some methods that can be used to debug your CMS code.

Look also in the HOWTO directory of the reference software release for latest documents on debugging.

### Using Log Messages

You can control the verbosity of CMS applications at runtime. By default, all applications only display messages that are logged with `cmsLog_error()`. However, if you are debugging a particular application, e.g., `httpd`, the verbosity level of the `httpd` application can be increased by logging into CLI (via serial console, telnet, or sshd) and typing **loglevel set httpd Debug** or **loglevel set httpd Notice**.

Setting the log level to Debug will cause all messages that are logged with `cmsLog_debug()`, `cmsLog_notice()`, and `cmsLog_error()` to be displayed. Setting the log level to Notice will cause all messages logged with `cmsLog_notice()` and `cmsLog_error()` to be displayed. To switch back to the normal log level (where only `cmsLog_error()` messages are displayed), type **loglevel set httpd Error**.

Note that when you set the log level, it is not automatically saved to the config file. If you want the log level setting to be saved to the config file (so that it will take effect on the next reboot of the modem), you should type **save**, which causes the configuration to be saved to Flash memory.

Using this method to debug requires an understanding of the various roles of CMS applications. For example, if you are interested in seeing debugging information from the `cgi_xxx` or DAL code, you would obviously set the debug level on the `httpd` application. Also, if you are interested in seeing how the RCL handler functions are running as a result of setting configuration changes from the WebUI, you would also set the debug level on the `httpd` application. However, if you are interested in seeing applications being launched, you would set the debug level on `smd` because `smd` is responsible for launching applications. If you are interested in seeing how asynchronous link events, e.g., link up, are handled, you would set the debug level on `ssk` because `ssk` is the applications that handles those asynchronous events.

### Startup Debugging

Sometimes the error you are trying to debug occurs before the CLI becomes available. In this event, you will not be able to set the `loglevel` before the error occurs. The error may also occur before any application is able to read their configured log level in the MDM. For example, when the modem boots up, during the activate objects phase, all RCL handler functions are called. If there is a bug in any of the RCL handler functions, the `ssk` process may terminate before any debug log messages can be printed.

To enable debugging in these situations:

1. Type **make menuconfig**.
2. Go to **Debug selection** and enable the **Enable CMS Startup Debug** option.
3. Rebuild your image for this change to take effect.

Selecting **Enable CMS Startup Debug** is equivalent to running `smd` and `ssk` at the debug log level. After the system has successfully booted, you can set the `loglevel` parameters of `smd` and `ssk` back to the Error level by going to the CLI and typing **loglevel set smd Error** and **loglevel set ssk Error**.

---

## Dumping the MDM

During debugging, it may be useful to inspect the contents of the MDM. To do this, log into the modem via serial console, telnet, or sshd and type **dumpmdm**. This will cause the entire MDM to be dumped.

You can also inspect the contents of the configuration file in Flash memory by typing **dumpcfg**. Note that the configuration file is a subset of the MDM. Not all parameters inside the MDM need to be written to the configuration file. For example, statistics parameters inside the MDM are not written to the configuration file. In general, only read/write parameters with values that differ from the default value are written out to the configuration file.

You can also see what parameters would be written out to the config file from the MDM in shared memory by typing **dumpcfg dynamic**. This command is useful when debugging issues where parameters that should be written to the configuration file do not seem to get written out to the configuration file. To look for differences between what is in shared memory and what is in the config Flash, type **dumpcfg dynamic** and **dumpcfg**.

---

## Dumping the Persistent Scratch Pad

During debugging, it may be useful to inspect the contents of the Persistent Scratch Pad. Some applications, such as `tr69c` and `ppp`, store state information in the persistent scratch pad.

Log into the modem via serial port console, telnet, or sshd and type **psp** to get a list of command options. The options are:

- `psp list`, which lists all keys in the persistent scratch pad.
- `psp dump <key>`, which dumps the data contents of the specified key.
- `psp delete <key>`, which deletes the specified key and its data contents.
- `psp clearall`, which deletes all keys and data from the persistent scratch pad.

---

## Memory Leak Tracing

An advanced memory leak tracing feature was introduced in the 4.04L.01 release. By default, this feature is not enabled. To enable this feature, you need to go to make `menuconfig`, and enable **CMS Memory Leak Tracing** in the **Debug selection** section.

To use this feature, the developer must have a basic understanding of the way memory allocation works in CMS. In CMS, every memory allocation call will go through `cmsMem_alloc()`. Code can call `cmsMem_alloc()` directly, or they may call other APIs such as `cmsMem_strdup()`, `cmsObj_get()`, `cmsObj_getNext()`, or they may do something which causes other memory allocation calls such as `mdm_dupObject()`. In all cases, memory allocation calls will go through `cmsMem_alloc()`.

When the memory leak tracing feature is enabled, `cmsMem_alloc()` will record a sequence number plus the last 15 functions in the stack up to the `cmsMem_alloc` call. (The number of functions that is recorded is defined in `public/libs/cms_util/memory.c`, `#define NUM_STACK_ENTRIES`). The memory allocator keeps track of which buffers are still allocated. An application can call the following APIs to inspect the buffers that are still allocated and their stack trace, which should help to identify the source of the memory leak:

- `cmsMem_dumpTraceAll()`, which dumps all the allocated buffers and their stack traces.
- `cmsMem_dumpTrace50()`, which dumps the last 50 allocated buffers and their stack traces.
- `cmsMem_dumpTraceClones()`, which may be the most useful of this set of APIs. This function will compare the stack traces of all the still allocated buffers and print out only buffers that have five or more clones (buffers with identical stack traces). The `cmsMem_dumpTraceClones()` API uses the fact that if an operation leaks one buffer, cloned buffers can be detected by running the same operation five or more times. The number of clones can be changed by modifying `#define CLONE_COUNT_THRESH` in `userspace/public/libs/cms_util/memory.c`.

These functions are declared in `userspace/public/include/cms_mem.h`.

Note that when an application calls `cmsObj_set`, `cmsObj_set` will call `mdm_dupObject` to make a duplicate of the object. The current object in the MDM is freed, and the new duplicate is then inserted into the MDM. This new duplicate is not freed until the next call to `cmsObj_set()`. If the object contains many non-null string parameters, `mdm_dupObject` will call `cmsMem_alloc` once for the object itself, and once for every non-null string parameter in the object. So if there are seven non-null string parameters in the object, there will be a total of eight clones in the memory allocator. These eight clones will be reported as clones by `cmsMem_dumpTraceClones()` even though this is not a memory leak because this object and its seven associated string buffers will be freed during the next `cmsObj_set()`. There are three ways to avoid this issue:

- By increasing the `CLONE_COUNT_THRESH` value.
- By calling `cmsMem_dumpTraceClones()` once before performing the memory leaking operations and once after the memory leaking operations and eliminating any clones that are reported in both the dump before the operations were done and in the dump after the operations. The clones that are in the dump after the operations are likely to be real leaks.
- By checking the sequence numbers in the allocation records. If the sequence numbers are sequential, they were probably caused by an `mdm_dupObject` operation, which may mean it is not a leak.

The previous example illustrates a more general point, which is this memory leak feature only records the stack trace of the caller of `cmsMem_alloc()`. In many cases, the caller to `cmsMem_alloc()` is also responsible for freeing the buffer. However, there are other valid use cases where the caller is not responsible for freeing the buffer. The developer must use their knowledge of how the program is supposed to work to track down the source of the memory leak.

Some CLI commands have been added to help identify memory leaks in `ssk`, `httpd`, `tr69c`, and the CLI. The format of the command is:

```
meminfo <app_name> <command>
```

So if you want `tr69c` to dump out its clones, then type:

```
meminfo tr69c traceclones
```

Once the clones and their stack traces are printed out, you can decode the function addresses by using the `symfind-userspace.sh` script in `hostTools/scripts/misc`. Be sure to change the `VIEW_PATH` variable in that script to point to your build environment.

## Enabling Shell Timeout

Sometimes you want to run a script in the busybox shell for a long time (e.g., 24 hours). By default, the busybox shell does not time out. To enable the idle timeout feature in the busybox shell, edit the file at `targets/fs.src/etc/profile`. Change the line that looks like:

```
export TIMEOUT=0
```

to:

```
export TMOUT=300
```

where 300 is the number of seconds before timeout. You can change this value to whatever you want.

You need to rebuild your image for this change to take effect.



**Note:** TMOUT setting only controls the timeout behavior of the busybox shell (with the `#` prompt). To control to exit-on-idle behavior of `consoled`, `telnetd`, `sshd`, `httpd`, and `tr69c`, look in `userspace/public/include/cms_params.h` and search for `EXIT_ON_IDLE_TIMEOUT`.

# Appendix A: Acronyms and Abbreviations

Table 22 defines some terms that are used throughout the document.

**Table 22: Acronyms and Abbreviations**

<b>Abbreviation/ Term</b>	<b>Definition</b>
Application	An application can be a management application, or it can be any process that implements a feature. Typically, an application will run for a long time. Some examples of applications are pppd, dhcpc, dhcpcd, and ddnsd.
Command	This is a userspace executable that runs very briefly, performs a specific task, and then exits. Examples of commands are iptables, ebtables, adsctl, wlctl, brctl, and ifconfig.
config file	The part of the MDM that is written out to Flash memory. The entire MDM does not need to be written to Flash memory because the MDM contains read-only parameters that will be regenerated next time the modem boots.
Data Model	The definition of what configuration parameters exist and how they are organized. CMS uses the data model defined in TR-98 [3] and extended by TR-104 [4] and TR-111 [5]. The data model describes what can be configured but does not describe what is actually configured, nor does it contain any actual user configuration settings. The data model is basically the same for all modems. Refer to data-model/cms-dm-*.xml.
Entity ID	An identifier associated with a particular application. Like a Process ID (PID), but it does not identify a specific instance of an application.
full path name or MDM full path or full path	<p>The name of an object or parameter starting from InternetGatewayDevice with all instance numbers specified.</p> <p>An example of a full path name to an object is: InternetGatewayDevice.WANDevice.1.WanConnectionDevice.3.WanPPPConnection.6</p> <p>An example of a full path name to a parameter is: InternetGatewayDevice.WANDevice.1.WanConnectionDevice.3.WanPPPConnection.6.LinkStatus</p>
generic path name	<p>This is similar to a full path name, but with the instance numbers stripped out. Generic path names are used within the CMS to identify objects (but not the specific instance of an object). An example of a generic path name is: InternetGatewayDevice.WANDevice.{i}.WanConnectionDevice.{i}.WanPPPConnection.{i}.</p> <p>Instead of using a generic path name to identify an object, CMS normally uses an MDM Object ID (MDMOID) to identify an object. Neither a generic path name nor a MDM Object ID is sufficient to identify the specific instance of an object. However, a generic path name or MDM Object ID along with an Instance ID Stack (iidStack) is enough to identify a specific instance of an object.</p>
InstanceIdStack or iidStack	A compact representation of all the instance numbers embedded in a full path name.
Management Application	Any application that modifies the configuration of the system. Management applications include tr69c, httpd, telnetd, sshd, consoled, snmp, tr64, and upnp.



**Table 22: Acronyms and Abbreviations (Cont.)**

<b>Abbreviation/ Term</b>	<b>Definition</b>
MDM	Memory Data Model. The realization of the data model in system memory using C data structures. The MDM is readable and writable by all management applications through the Object Layer or Parameter Handling Layer APIs. Very early in the system bootup sequence, the MDM reflects the data model. However, later in the bootup sequence and during the runtime operation of the modem, the MDM is populated by settings from the config file and settings from management applications. The MDM contains what is actually configured on each CPE system and is likely to be different among CPE systems. There is also an MDM API that is used to access the MDM. It should be clear from the context whether we are referring to the MDM API or the MDM.
MdmObject	This means either a C data structure containing a group of parameter names in the data model or the block of memory that contains the data for the C data structure.
MdmObjectId or OID	A #define that is used by various API functions to identify which object it should be operating on. There is one #define for each object in the data model. The #define associates a symbol with a number. For example, #define MDMOID_DEVICEINFO 2. Refer to userspace/public/include/mdm_objectid.h.
object	This is a very generic term. In this document, object typically means MdmObject.
ODL	Object Dispatch Layer. This acts as an intermediate layer between the Object and Parameter Handling layers and the RCL and STL.
OID	See MdmObjectId
QDM	Query Data Model. This is a new API layer and library introduced in 4.14L.XX. See <a href="#">Section 11: "Query Data Model (QDM) Library"</a> .
parameter	This is a very generic term. In this document, a parameter typically refers to a single configuration or status item at the leaf of the data model. For example, DestinationAddress or LinkStatus. Each parameter belongs to a specific MdmObject.
RCL	Runtime Config Layer. Contains handler functions that are called when a MdmObject is created, modified, or deleted.
RUT	Runtime Utility. These are helper functions for the RCL and STL handler functions. They reduce the amount of code in the RCL and STL handler functions, thus making the RCL and STL handler functions easier to understand.
STL	System Status Layer. Contains handler functions that are called when a MdmObject is read, or if statistics need to be cleared.
Type 0 object	This object has exactly one instance in the system. This instance cannot be deleted and no additional instances can be added. In the generic path name of a type 0 object, there will be no .{i}. anywhere in the path name. An example of a type 0 object is InternetGatewayDevice.DeviceInfo.  Type 0 objects can only be descendents of other type 0 objects. It can never be a descendent of a type 1 or type 2 object.
Type 1 object	There may be zero or more instances of this object in the system. This object is always a descendent (child, grandchild, etc..) of a type 2 object. Instances of a type 1 object cannot be added or deleted, however, when an instance of a type 2 object is added, all descendents of the type 2 object that are type 1 will be created as a side effect of the type 2 object being added. Similarly, when a type 2 object is deleted, all type 1 descendents of the type 2 object will also be automatically deleted. The generic path name of a type 1 object will not end in .{i}. but it will have one or more .{i}. in the generic path name. An example of a type 1 object is: InternetGatewayDevice.LANDevice.{i}.LanHostConfigManagement.



**Table 22: Acronyms and Abbreviations (Cont.)**

<b>Abbreviation/ Term</b>	<b>Definition</b>
Type 2 object	<p>There may be zero or more instances of this object in the system. The generic path name of a type 2 object will always end in <code>.{i}</code>. Type 2 objects are further subdivided into Dynamic Instance and Multiple Instance objects. Dynamic instance type 2 objects can be added and deleted by the management applications. An example of a dynamic instance type 2 object is: <code>InternetGatewayDevice.WANDevice.WanConnectionDevice.{i}</code>.</p> <p>Multiple instance type 2 objects can only be added or deleted by internal system applications, such as ssk. An example of a multiple instance type 2 object is: <code>InternetGatewayDevice.LANDevice.{i}.LanHostConfigManagement.Hosts.{i}</code>.</p>

BROADCOM CONFIDENTIAL

## Appendix B: Compiling the Data Model Designer

For Broadcom designers, make sure you have javac installed and that it is on your path. Type **javac -version** to confirm the version number. Javac 1.5 or greater is required.

To compile the Data Model Designer, cd into hostTools/DataModelDesigner and type **make**. Do not be alarmed if a compile line is not displayed for every java file in the directory. The Java compiler works behind the scenes and compiles files without displaying the information line by line. The result of the compile is a file called DataModelDesigner.jar.

BROADCOM CONFIDENTIAL

BROADCOM CONFIDENTIAL

Broadcom® Corporation reserves the right to make changes without further notice to any products or data herein to improve reliability, function, or design.

Information furnished by Broadcom Corporation is believed to be accurate and reliable. However, Broadcom Corporation does not assume any liability arising out of the application or use of this information, nor the application or use of any product or circuit described herein, neither does it convey any license under its patent rights nor the rights of others.

**Broadcom Corporation**

5300 California Avenue

Irvine, CA 92617

© 2014 by BROADCOM CORPORATION. All rights reserved.

CPE-SWUM102-R July 3, 2014



Phone: 949-926-5000

Fax: 949-926-5203

E-mail: [info@broadcom.com](mailto:info@broadcom.com)

Web: [www.broadcom.com](http://www.broadcom.com)