Yuma® Developer Manual

YANG-Based Unified Modular Automation Tools

Table Of Contents

Yuma Developer Manual

1   Preface

1.1   Legal Statements

Copyright 2009 - 2010, Netconf Central, Inc., All Rights Reserved.

1.2   Restricted Rights Legend

This software is provided with RESTRICTED RIGHTS.

Use, duplication, or disclosure by the U.S. Government is subject to restrictions set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs(c)(1) and (2) of the Commercial Computer Software - Restricted Rights at 48 CFR 52.227-19, as applicable.

The "Manufacturer" for purposes of these regulations is Netconf Central, Inc., 374 Laguna Terrace, Simi Valley, California 93065 U.S.A.

1.3   Additional Resources

This document assumes you have successfully set up the software as described in the printed document:

> Yuma Tools® Installation Guide
>
> Yuma Tools® Quickstart Guide

Depending on the version of Yuma you purchased, other documentation includes:

> Yuma Tools® User Manual

To obtain additional support you may email InterWorking Labs at this e-mail address

> yuma-support@iwl.com

There are several sources of free information and tools for use with YANG and/or NETCONF.

The following section lists the resources available at this time.

1.3.1   WEB Sites

- **Yuma Tools Home Page**
  - http://yuma.iwl.com/
  - Official home page for Yuma Tools information
- **Netconf Central**
  - http://www.netconfcentral.org/
  - Free information on NETCONF and YANG, tutorials, on-line YANG module validation and documentation database

- **Yang Central**
  - ◦ http://www.yang-central.org
  - ◦ Free information and tutorials on YANG, free YANG tools for download
- **NETCONF Working Group Wiki Page**
  - ◦ http://trac.tools.ietf.org/wg/netconf/trac/wiki
  - ◦ Free information on NETCONF standardization activities and NETCONF implementations
- **NETCONF WG Status Page**
  - ◦ http://tools.ietf.org/wg/netconf/
  - ◦ IETF Internet draft status for NETCONF documents
- **libsmi Home Page**
  - ◦ http://www.ibr.cs.tu-bs.de/projects/libsmi/
  - ◦ Free tools such as smidump, to convert SMIv2 to YANG
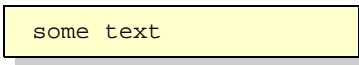
1.3.2   MAILING LISTS

- **NETCONF Working Group**
  - ◦ http://www.ietf.org/html.charters/netconf-charter.html
  - ◦ Technical issues related to the NETCONF protocol are discussed on the NETCONF WG mailing list. Refer to the instructions on the WEB page for joining the mailing list.
- **NETMOD Working Group**
  - ◦ http://www.ietf.org/html.charters/netmod-charter.html
  - ◦ Technical issues related to the YANG language and YANG data types are discussed on the NETMOD WG mailing list. Refer to the instructions on the WEB page for joining the mailing list.

1.4   Conventions Used in this Document

The following formatting conventions are used throughout this document:

Documentation Conventions

Convention	Description
--foo	CLI parameter foo
<foo>	XML parameter foo
foo	**yangcli** command or parameter
$$FOO	Environment variable FOO
$$foo	**yangcli** global variable foo
`some text`	Example command or PDU
some text	Plain text

2   Software Overview
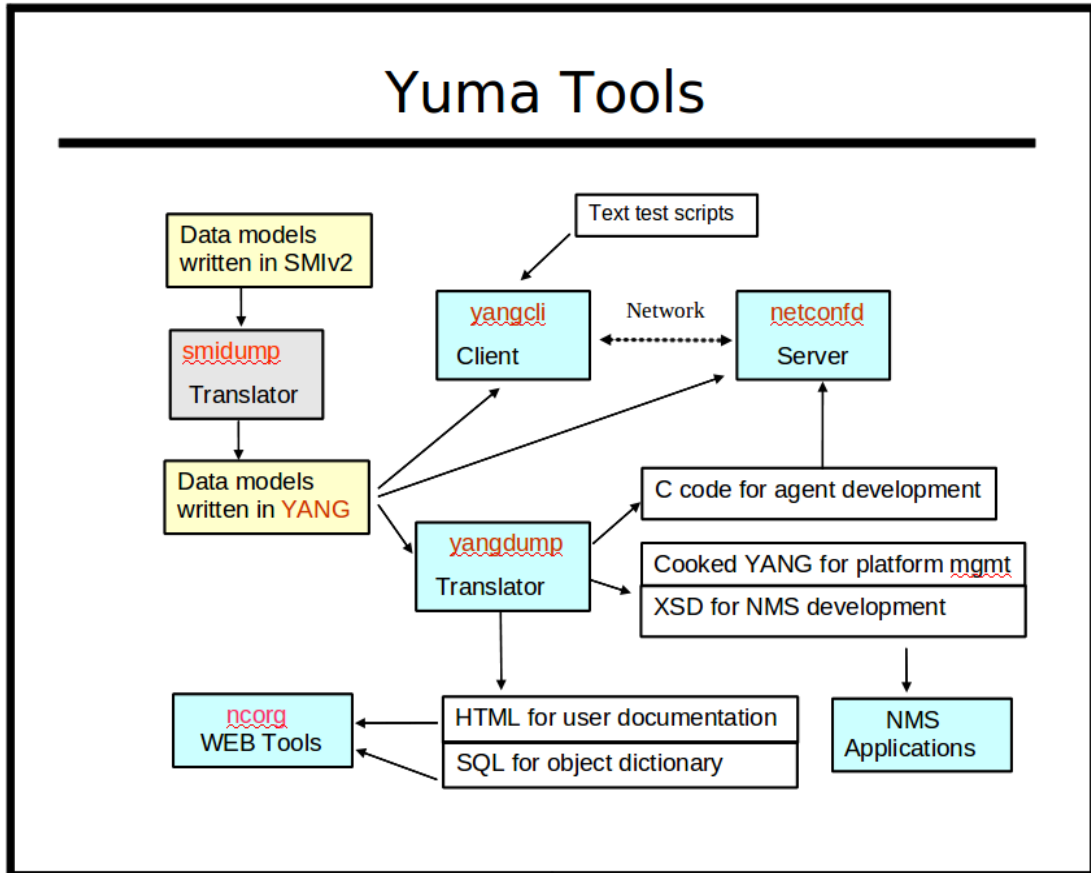
2.1   Introduction

Refer to section 3 of the Yuma User Manual for a complete introduction to Yuma Tools.

This section focuses on the software development aspects of NETCONF, YANG, and the **netconfd** server.

2.1.1   INTENDED AUDIENCE

This document is intended for developers of server instrumentation library software, which can be used with the programs in the Yuma suite. It covers the design and operation of the **netconfd** server, and the development of server instrumentation library code, intended for use with the **netconfd** server.

2.1.2   WHAT DOES YUMA DO?

The Yuma Tools suite provides automated support for development and usage of network management information. Refer to the Yuma User Guide for an introduction to the YANG data modeling language and the NETCONF protocol.

This section describes the Yuma development environment and the basic tasks that a software developer needs to perform, in order to integrate YANG module instrumentation into a device.

This manual contains the following information:

- Yuma Development Environment

- Yuma Runtime Environment

- Yuma Source Code Overview

- Yuma Server Instrumentation Library Development Guide

Yuma Tools programs are written in the C programming language, using the 'gnu99' C standard, and should be easily integrated into any operating system or embedded device that supports the Gnu C compiler.

2.1.3   WHAT IS A YUMA ROOT?
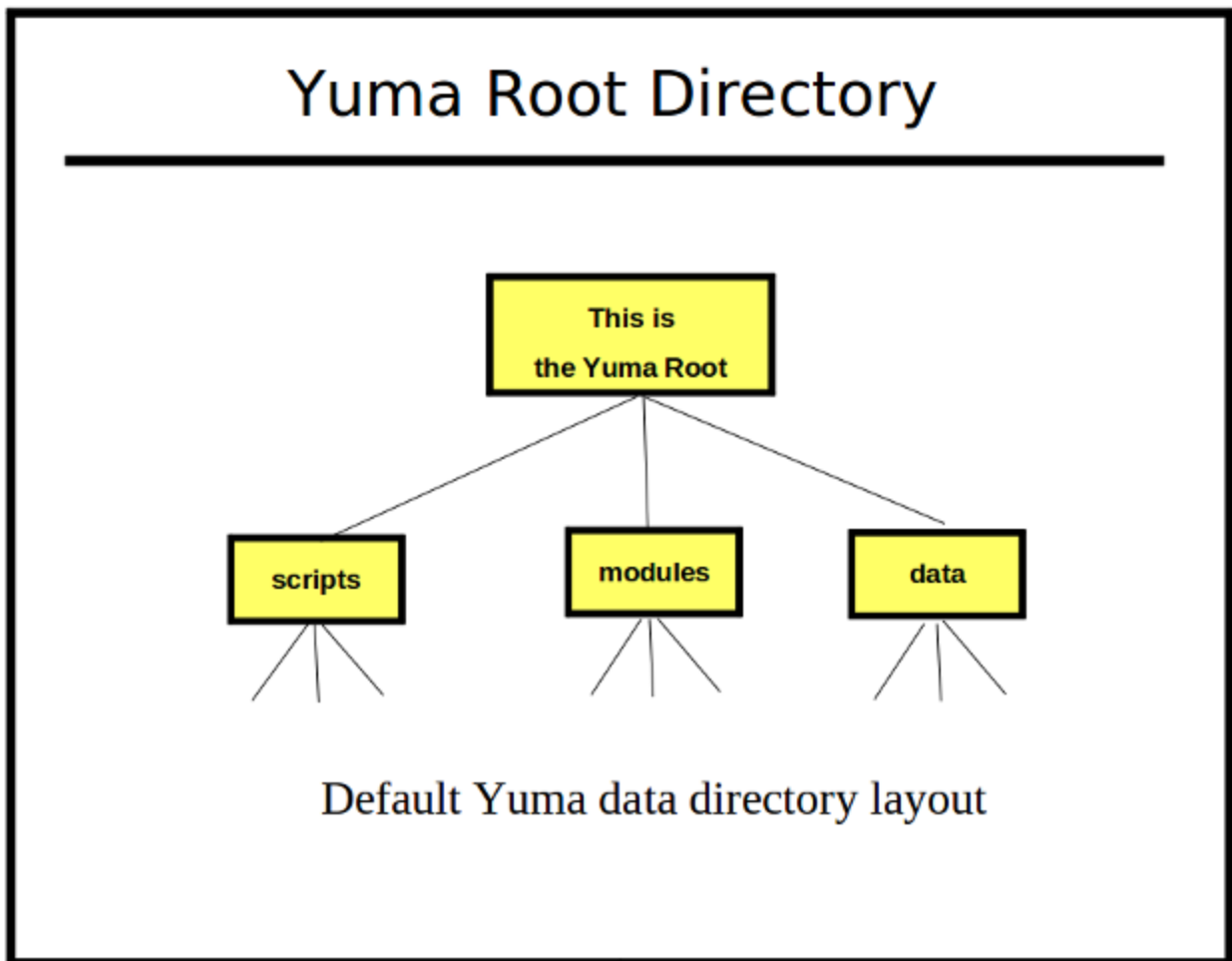
The Yuma Tools programs will search for some types of files in default locations

- **YANG Modules**: The 'modules' sub-directory is used as the root of the YANG module library.
- **Client Scripts**: The yangcli program looks in the 'scripts' sub-directory for user scripts.
- **Program Data**: The yangcli and netconfd programs look for saved data structures in the 'data' sub-directory.

2.1.4   SEARCHING YUMA ROOTS
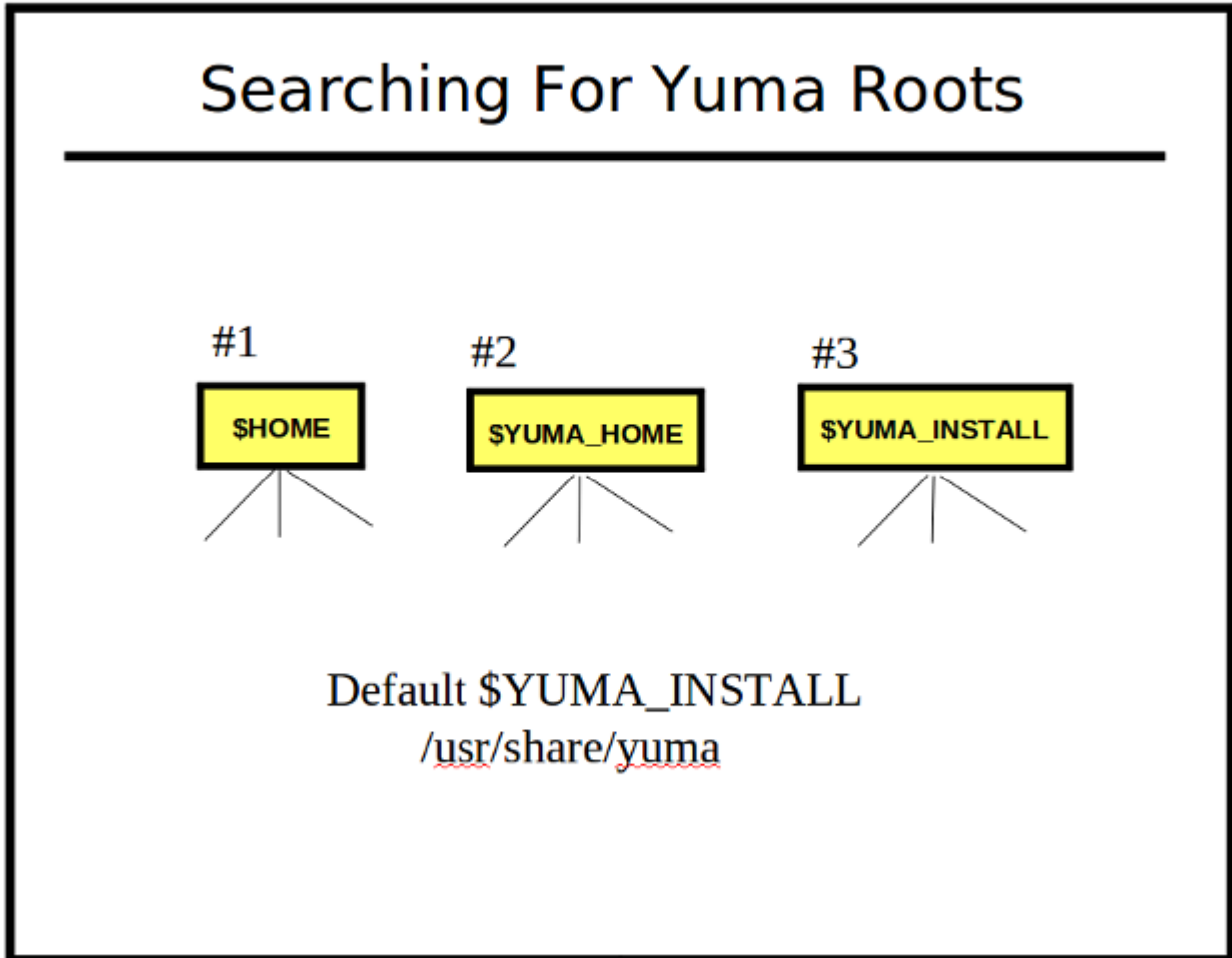
1) $HOME Directory

The first Yuma root checked when searching for files is the directory identified by the $HOME environment variable. If a '**$HOME/modules**', '**$HOME/data**'. and/or '**$HOME/scripts**' directory exists, then it will be checked for the specified file(s).

2) The $YUMA_HOME Directory

The second Yuma root checked when searching for files is the directory identified by the $YUMA_HOME environment variable. This is usually set to private work directory, but a shared directory could be

used as well. If a '**$YUMA_HOME/modules**', '**$YUMA_HOME/data**'. and/or '**$YUMA_HOME/scripts**' directory exists, then it will be checked for the specified file(s).

3) The $YUMA_INSTALL Directory

The last Yuma root checked when searching for files is the directory identified by the $YUMA_INSTALL environment variable. If it is not set, then the default value of '/**usr/share/yuma**' is used instead. This is usually set to the public directory where all users should find the default modules. If a '**$YUMA_INSTALL/modules**', '**$YUMA_INSTALL/data**'. and/or '**$YUMA_INSTALL/scripts**' directory exists, then it will be checked for the specified file(s).

2.1.5 WHAT IS A SIL?

A SIL is a Server Instrumentation Library. It contains the 'glue code' that binds YANG content (managed by the **netconfd** server), to your networking device, which implements the specific behavior, as defined by the YANG module statements.

The **netconfd** server handles all aspects of the NETCONF protocol operation, except data model semantics that are contained in description statements. The server uses YANG files directly, loaded at boot-time or run-time, to manage all NETCONF content, operations, and notifications.

Callback functions are used to hook device and data model specific behavior to database objects and RPC operations. The **yangdump** program is used to generate the initialization, cleanup, and 'empty' callback functions for a particular YANG module. The callback functions are then completed (by you), as required by the YANG module semantics. This code is then compiled as a shared library and made available to the **netconfd** server. The 'load' command (via CLI, configuration file, protocol operation) is used (by the operator) to activate the YANG module and its SIL.

2.1.6 BASIC DEVELOPMENT STEPS

The steps needed to create server instrumentation for use within Yuma are as follows:

- **Create the YANG module data model** definition, or use an existing YANG module.

  ◦ Validate the YANG module with the **yangdump** program and make sure it does not contain any errors. All warnings should also be examined to determine if they indicate data modeling bugs or not.

  > ◦ Example `toaster.yang`

- Make sure the **$$YUMA_HOME** environment variable is defined, and pointing to your Yuma development tree.

- **Create a SIL development subtree**

  ◦ Generate the directory structure and the Makefile with the **make_sil-_ir** script, installed in the **/usr/bin** directory. This step will also call yangdump to generate the initial H and C files file the SIL.

  > ◦ Example: `mydir>` **`make_sil_dir toaster`**

- **Use your text editor to fill in the device-specific instrumentation** for each object, RPC method, and notification. Almost all possible NETCONF-specific code is either handled in the central stack, or generated automatically. so this code is responsible for implementing the semantics of the YANG data model.

- You can mostly ignore the developer manual if you look in the auto-generated code for C comments to 'begin add your code here' and end add your code here'. All database operations and YANG machine-readable constructs are handled by the server, so the only code needed are the hooks from the data model to the device instrumentation.

- **Compile your code**

  - Use the 'make' command in the SIL 'src' directory. This should generate a library file in the SIL 'lib' directory.

    ```
    Example:  mydir/toaster/src> make
    ```

- **Install the SIL library so it is available to the netconfd server.**

  - Use the 'make install' command in the SIL 'src' directory.

    ```
    Example:  mydir/toaster/src> make install
    ```

- **Run the netconfd server** (or build it again if linking with static libraries)

- **Load the new module**

  - Be sure to add a 'load' command to the configuration file if the module should be loaded upon each reboot.

    ```
    yangcli Example: load toaster
    ```

- The **netconfd** server will load the specified YANG module and the SIL and make it available to all sessions.

2.2   Yuma Source Files

This section describes the files that are contained in the **yuma-source** package.

The important C include files are copied into **/usr/include/yuma** when the **yuma-dev** package is installed. The full set of installation sources is installed in **/usr/share/yuma/src**, if the **yuma-source** package is installed. Yuma tools will check the **$YUMA_HOME/src** sub-tree before checking this default installation location. This allows a working copy of the Yuma sources to be used instead of the installation copy, in case it has been modified for a particular embedded system (for example).

This section lists the files that are included within the **netconf/src** directory.

2.2.1   SRC/NCX DIRECTORY

This directory contains the code that is used to build the **libncx.so** binary shared library that is used by all Yuma Tools programs. It handles many of the core NETCONF/YANG data structure support, including all of the YANG/YIN, XML, and XPath processing. The following table describes the purpose of each file. Refer to the actual include file (e.g., ncx.h in /usr/include/yuma) for more details on each external function in each C source module.

src/ncx C Modules

C Module	Description
b64	Encoding and decoding the YANG binary data type.
blob	Encoding and decoding the SQL BLOB data type.
bobhash	Implementation of the BOB hash function.
cap	NETCONF capability definitions and support functions
cfg	NETCONF database data structures and configuration locking support.
cli	CLI parameter parsing data driven by YANG definitions.
conf	Text .conf file encoding and decoding, data driven by YANG definitions.
def_reg	Hash-driven definition registry for quick lookup support of some data structures. Contains back-pointers to the actual data.
dlq	Double linked queue support
ext	YANG extension data structure support
grp	YANG grouping data structure support
help	Automatic help text, data-driven by YANG definitions
log	System logging support
ncx_appinfo	Yuma Netconf Extensions (NCX) support
ncx	YANG module data structure support, and some utilities
ncx_feature	YANG feature and if-feature statement data structure support
ncx_list	Support for the **ncx_list_t** data structure, used for YANG bits and **ncx:xsdlist** data types.
ncxmod	File Management: Controls finding and searching for YANG/YIN files, data files, and script files
ncx_num	Yuma **ncx_num_t** data structure support. Used for processing value nodes and XPath numbers.
ncx_str	Yuma string support.
obj	Yuma object **(obj_template_t)** data structure access
obj_help	Automated object help support used with help module
op	NETCONF operations definitions and support functions
rpc	NETCONF <rpc> and <rpc-reply> data structures and support functions
rpc_err	NETCONF <rpc-error> data structures and support functions.
runstack	Script execution stack support for yangcli scripts
send_buff	NETCONF send buffer function
ses	NETCONF session data structures and session access functions
ses_msg	Message buffering support for NETCONF sessions
status	Error code definitions and error support functions
tk	Token chain data structures used for parsing YANG, XPath and

C Module	Description
	other syntaxes.
top	Top-level XML node registration support. The <rpc> and <hello> elements are registered by the server. The <hello>, <rpc-reply> , and <notification> elements are registered by the client.
tstamp	Time and date stamp support functions
typ	YANG typedef data structures and access functions
val	Yuma value tree data structures and access functions
val_util	High-level utilities for some common SIL tasks related to the value tree.
var	User variable support, used by **yangcli** and (TBD) XPath
xml_msg	XML message data structures and support functions
xmlns	XML Namespace registry
xml_util	XML parse and utility functions
xml_val	High level support functions for constructing XML-ready **val_value_t** data structures
xml_wr	XML output support functions and access-control protected message generation support
xpath1	XPath 1.0 implementation
xpath	XPath data structures and support functions
xpath_wr	Support for generating XPath expression content within an XML instance document
xpath_yang	Special YANG XPath construct support, such as path expressions and instance identifiers
yang	YANG definitions and general support functions
yang_ext	YANG parsing and validation of the extension statement
yang_grp	YANG parsing and validation of the grouping statement
yang_obj	YANG parsing and validation of the rpc, notification, and data definition statements
yang_parse	Top level YANG parse and validation support
yang_typ	YANG typedef and type statement support
yin	YANG to YIN mapping definitions
yinyang	YIN to YANG translation

2.2.2   SRC/PLATFORM DIRECTORY

This directory contains platform support include files and Makefile support files. It is used by all Yuma C modules to provide an insulating layer between Yuma programs and the hardware platform that is used. For example the **m__getMem, m__getObj**, and **m__freeMem** macros are used instead of **malloc** and **free** functions directly.

The following table describes the files that are contained in this directory:

src/platform Files

File	Description
curversion.h	File generated during the build process to get the SVNVERSION number
platform.profile	Included by Makefiles for build support
platform.profile.cmn	Included by Makefiles for build support
platform.profile.depend	Included by Makefiles for dependency generation support
procdefs.h	Platform definitions. Contains basic data types and macros used throughout the Yuma code. All C files include this file before any other Yuma files.
setversion.sh	Shell script to generate the curversion.h file

2.2.3   SRC/AGT DIRECTORY

This directory contains the NETCONF server implementation and built-in module SIL code. A static library called **libagt.a** is built and statically linked within the **netconfd** program.

The following table describes the C modules contained in this directory:

src/agt C Modules

C Module	Description
agt_acm	NETCONF access control implementation. Contains the **yuma-nacm** module SIL callback functions.
agt	Server initialization and cleanup control points. Also contains the **agt_profile_t** data structure.
agt_cap	Server capabilities. Generates the server <capabilities> element content.
agt_cb	SIL callback support functions.
agt_cli	Server CLI and .conf file control functions.
agt_connect	Handles the internal <ncx-connect> message sent from the **netconf-subsystem** to the **netconfd** server.
agt_hello	Handles the incoming client <hello> message and generates the server <hello> message.
agt_if	Yuma Interfaces module implementation. Contains the **yuma-interfaces** module SIL callback functions.
agt_ncx	NETCONF protocol operation implementation. Contains the **yuma-netconf** module SIL callback functions.
agt_ncxserver	Implements the ncxserver loop, handling the IO between the server NETCONF sessions and the **netconf-subsystem** thin client program.
agt_not	NETCONF Notifications implementation. Contains the **notifications** and **nc-notifications** modules SIL

C Module	Description
	callback functions.
agt_proc	**/proc** system monitoring implementation. Contains the **yuma-proc** module SIL callback functions.
agt_rpc	NETCONF RPC operation handler
agt_rpcerr	NETCONF <rpc-error> generation
agt_ses	NETCONF session support and implementation of the Yuma Session extensions. Contains the **yuma-mysession** module SIL callback functions.
agt_signal	Server signal handling support
agt_state	Standard NETCONF monitoring implementation. Contains the **ietf-netconf-monitoring** SIL callback functions.
agt_sys	Server system monitoring and notification generation. Contains the **yuma-system** module SIL callback functions.
agt_timer	SIL periodic timer callback support functions
agt_top	Server registration and dispatch of top-level XML messages
agt_tree	Subtree filtering implementation
agt_util	SIL callback utilities
agt_val	Server validation, commit, and rollback support for NETCONF database operations
agt_val_parse	Incoming <rpc> and <config> content parse and complete YANG constraint validation
agt_xml	Server XML processing interface to **ncx/xml_util** functions
agt_xpath	XPath filtering implementation

2.2.4   SRC/MGR DIRECTORY

This module contains the NETCONF client support code. It handles all the basic NETCONF details so a simple internal API can be used by NETCONF applications such as **yangcli**.   A static library called **libmgr.a** is built and statically linked within the **yangcli** program.

The following table describes the C modules contained in this directory:

src/mgr C Modules

C Module	Description
mgr	Client initialization and cleanup control points. Also contains manager session control block data structure support functions.
mgr_cap	Generate the client NETCONF <capabilities> element content

C Module	Description
mgr_hello	Handles the incoming server <hello> message and generates the client <hello> message.
mgr_io	Handles SSH server IO support for client NETCONF sessions
mgr_not	Handles incoming server <notification> messages
mgr_rpc	Generate <rpc> messages going to the NETCONF server and process incoming <rpc-reply> messages from the NETCONF server.
mgr_ses	Handles all aspects of client NETCONF sessions.
mgr_signal	Client signal handler
mgr_top	Client registration and dispatch of top-level XML messages
mgr_val_parse	Incoming <rpc-reply>, <notification>, and <config> content parse and complete YANG constraint validation.
mgr_xml	Client XML processing interface to **ncx/xml_util** functions

2.2.5   SRC/SUBSYS DIRECTORY

This directory contains the **netconf-subsystem** program. This is a thin-client application that just transfers input and output between the SSH server and the NETCONF server. It contains one C source module called **netconf-subsystem**. This is a stand-alone binary that is part of the **yuma-server** package. It is installed in the **/usr/sbin/** directory.

2.2.6   SRC/NETCONFD DIRECTORY

This directory contains the **netconfd** program, which implements the NETCONF server. It contains one C module called **netconfd**, which defines the NETCONF server 'main' function. This is a stand-alone binary that is part of the **yuma-server** package. It is installed in the **/usr/sbin/** directory.

2.2.7   SRC/YANGCLI DIRECTORY

This directory contains the **yangcli** program, which is the Yuma NETCONF client program. This is a stand-alone binary that is part of the **yuma-client** package. It is installed in the **/usr/bin/** directory.

The following table describes the C modules contained in this directory:

src/yangcli C Modules

C Module	Description
yangcli	NETCONF client program, provides interactive and script-based CLI, based on YANG modules.
yangcli_autoload	Uses the server capabilities from the <hello> message to automatically load any missing YANG modules from the server, and apply all features and deviations.
yang_autolock	Provides protocol exchange support for the high-level

C Module	Description
	get-locks and release-locks commands
yangcli_cmd	Main local command processor
yangcli_list	Implements yangcli 'list' command
yangcli_save	Implements yangcli 'save' command
yangcli_show	Implements yangcli 'show' command
yangcli_tab	Implements context-sensitive tab word completion
yangcli_util	Utilities used by other yangcli C modules

2.2.8   SRC/YANGDIFF DIRECTORY

This directory contains the **yangdiff** program, which is the Yuma YANG module compare program. This is a stand-alone binary that is part of the **yuma-client** package. It is installed in the **/usr/bin/** directory.

The following table describes the C modules contained in this directory:

src/yangdiff

C Module	Description
yangdiff	YANG module semantic compare program
yangdiff_grp	Implements semantic diff for YANG grouping statement
yangdiff_obj	Implements semantic diff for YANG data definition statements
yangdiff_typ	Implements semantic diff for YANG typedef and type statements
yangdiff_util	Utilities used by the other **yangdiff** C modules

2.2.9   SRC/YANGDUMP DIRECTORY

This directory contains the **yangdump** program, which is the Yuma YANG compiler program. This is a stand-alone binary that is part of the **yuma-client** package. It is installed in the **/usr/bin/** directory.

The following table describes the C modules contained in this directory:

src/yangdump C Modules

C Module	Description
c	Implements SIL C file generation
c_util	Utilities used for SIL code generation
h	Implements SIL H file generation
html	Implements YANG to HTML translation
sql	Implements SQL generation for YANG module WEB Docs

C Module	Description
xsd	Implements YANG to XSD translation
xsd_typ	Implements YANG typedef/type statement to XSD simpleType and complexType statements
xsd_yang	YANG to XSD translation utilities
yangdump	YANG module compiler
yangdump_util	Utilities used by all yangdump C modules
yangyin	Implements YANG to YIN translation

2.3   Server Design

This section describes the basic design used in the **netconfd** server.

Initialization:

Version 0.11-2
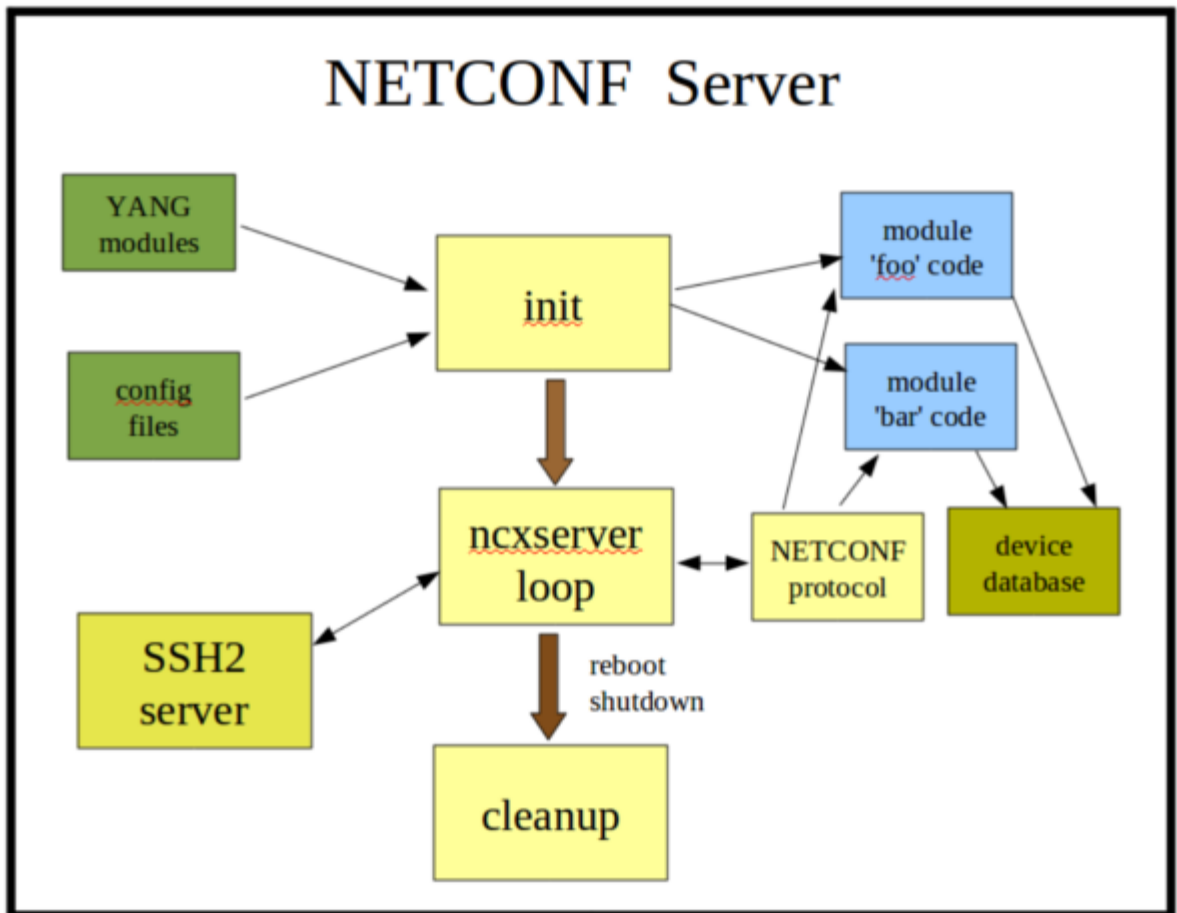
The **netconfd** server will process the YANG modules, CLI parameters, config file parameters, and startup device NETCONF database, then wait for NETCONF sessions.

ncxserver Loop:

The SSH2 server will listen for incoming connections which request the 'netconf' subsystem.

When a new session request is received, the **netconf-subsystem** program is called, which opens a local connection to the **netconfd** server, via the ncxserver loop. NETCONF <rpc> requests are processed by the internal NETCONF stack. The module-specific callback functions (blue boxes) can be loaded into the system at build-time or run-time. This is the device instrumentation code, also called a server implementation library (SIL). For example, for **libtoaster**, this is the code that controls the toaster hardware.

Cleanup:

If the <shutdown> or <reboot> operations are invoked, then the server will cleanup. For a reboot, the init cycle is started again, instead of exiting the program.

2.3.1   YANG Native Operation
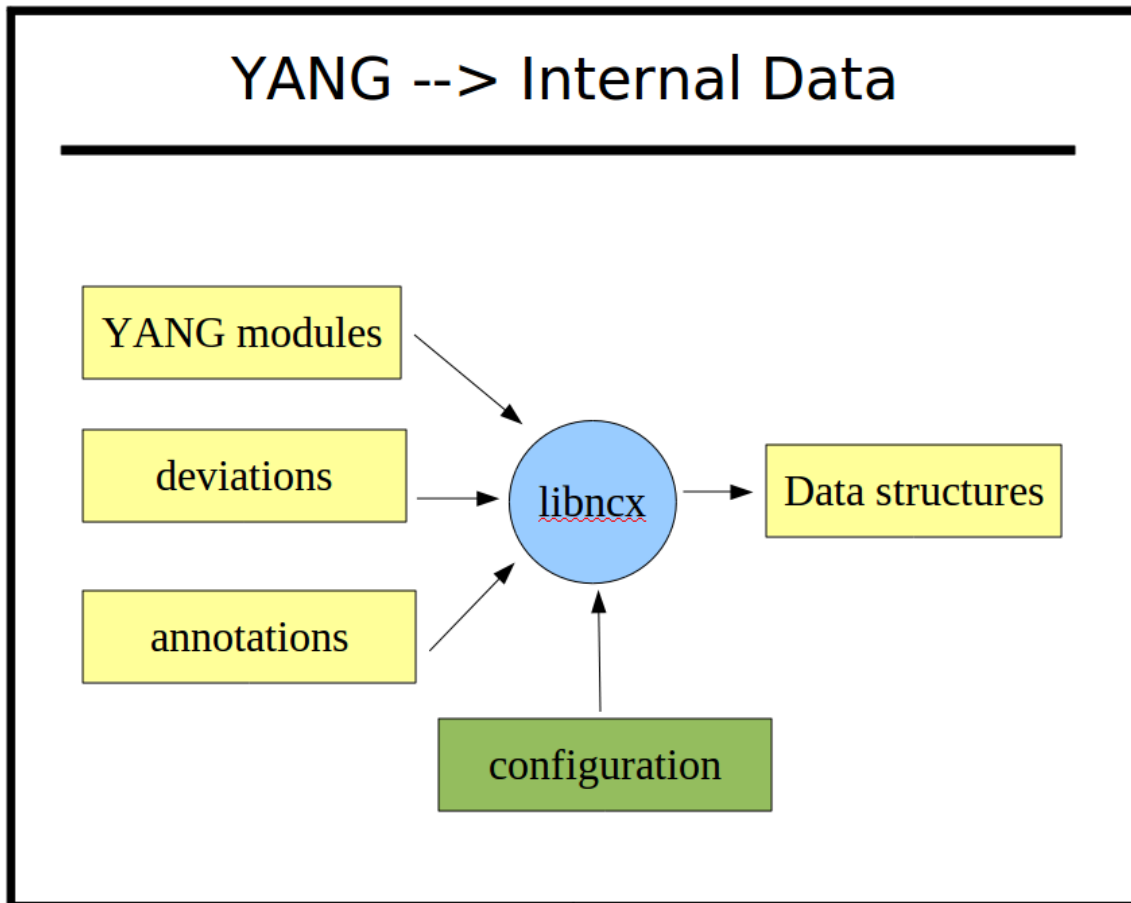
Yuma uses YANG source modules directly to implement NETCONF protocol operations automatically within the server. The same YANG parser is used by all Yuma programs. It is located in the 'ncx' source directory (libncx.so). There are several different parsing modes, which is set by the application.

In the 'server mode', the descriptive statements, such as 'description' and 'reference' are discarded upon input. Only the machine-readable statements are saved. All possible database validation, filtering, processing, initialization, NV-storage, and error processing is done, based on these machine readable statements.

For example, in order to set the platform-specific default value for some leaf, instead of hard-coded it into the server instrumentation, the default is stored in YANG data instead. The YANG file can be altered, either directly (by editing) or indirectly (via deviation statements), and the new or altered default value specified there.

In addition, range statements, patterns, XPath expressions, and all other machine-readable statements are all processed automatically, so the YANG statements themselves are like server source code.

YANG also allows vendor and platform-specific deviations to be specified, which are like generic patches to the common YANG module for whatever purpose needed. YANG also allows annotations to be defined and added to YANG modules, which are specified with the 'extension' statement. Yuma uses some extensions to control some automation features, but any module can define extensions, and module instrumentation code can access these annotation during server operation, to control device behavior.

There are CLI parameters that can be used to control parser behavior such as warning suppression, and protocol behavior related to the YANG content, such as XML order enforcement and NETCONF protocol operation support. These parameters are stored in the server profile, which can be customized for each platform.

2.3.2   YANG OBJECT TREE

YANG Object Tree
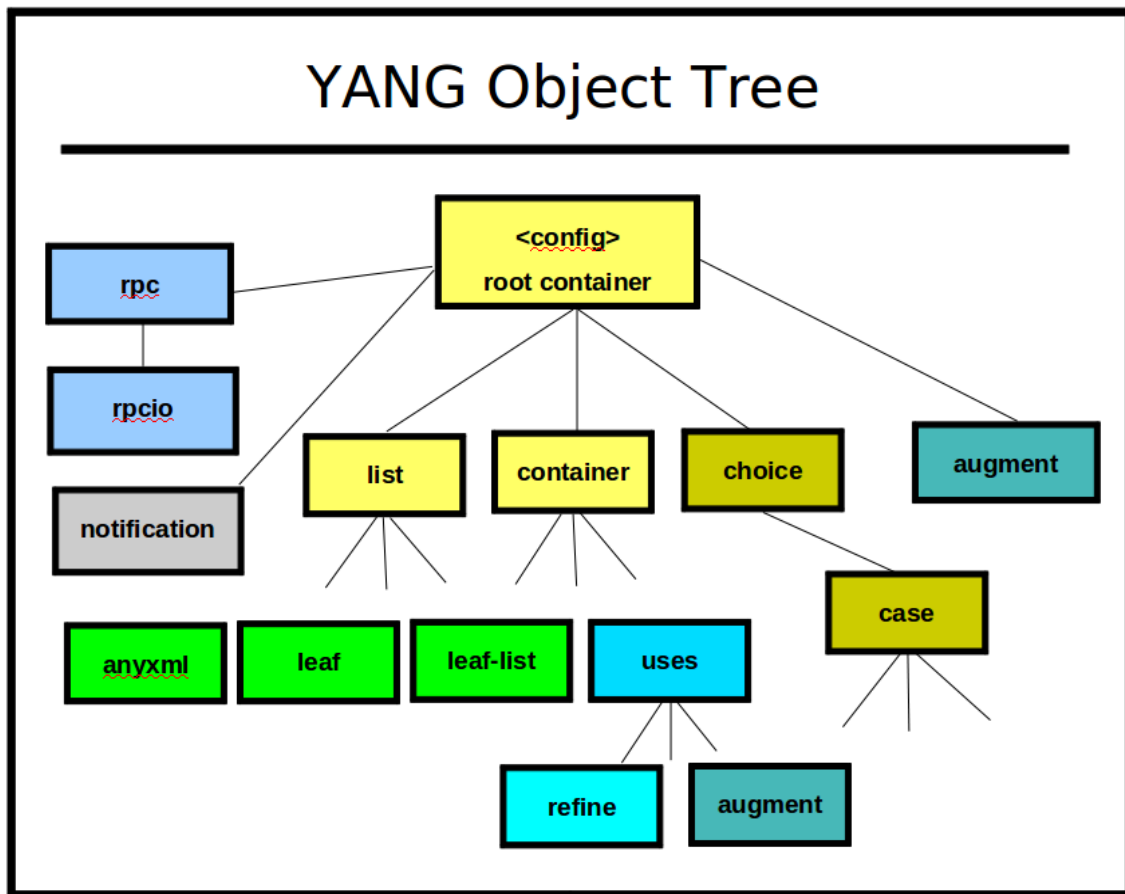
The YANG statements found in a module are converted to internal data structures.

For NETCONF and database operations, a single tree of **obj_template_t** data structures is maintained by the server. This tree represents all the NETCONF data that is supported by the server. It does not represent any actual data structure instances. It just defines the data instances that are allowed to exist on the server.

Raw YANG vs. Cooked YANG:

Some of the nodes in this tree represent the exact YANG statements that the data modeler has used, such as 'augment', 'refine', and 'uses', but these noeds are not used directly in the object tree. They exist in the object tree, but they are processed to produce a final set of YANG data statements, translated into 'cooked' nodes in the object tree. If any deviation statements are used by server implementation of a YANG data node (to change it to match the actual platform implementation of the data node), then these are also 'patched' into the cooked YANG nodes in the object tree.

2.3.3   YANG DATA TREE
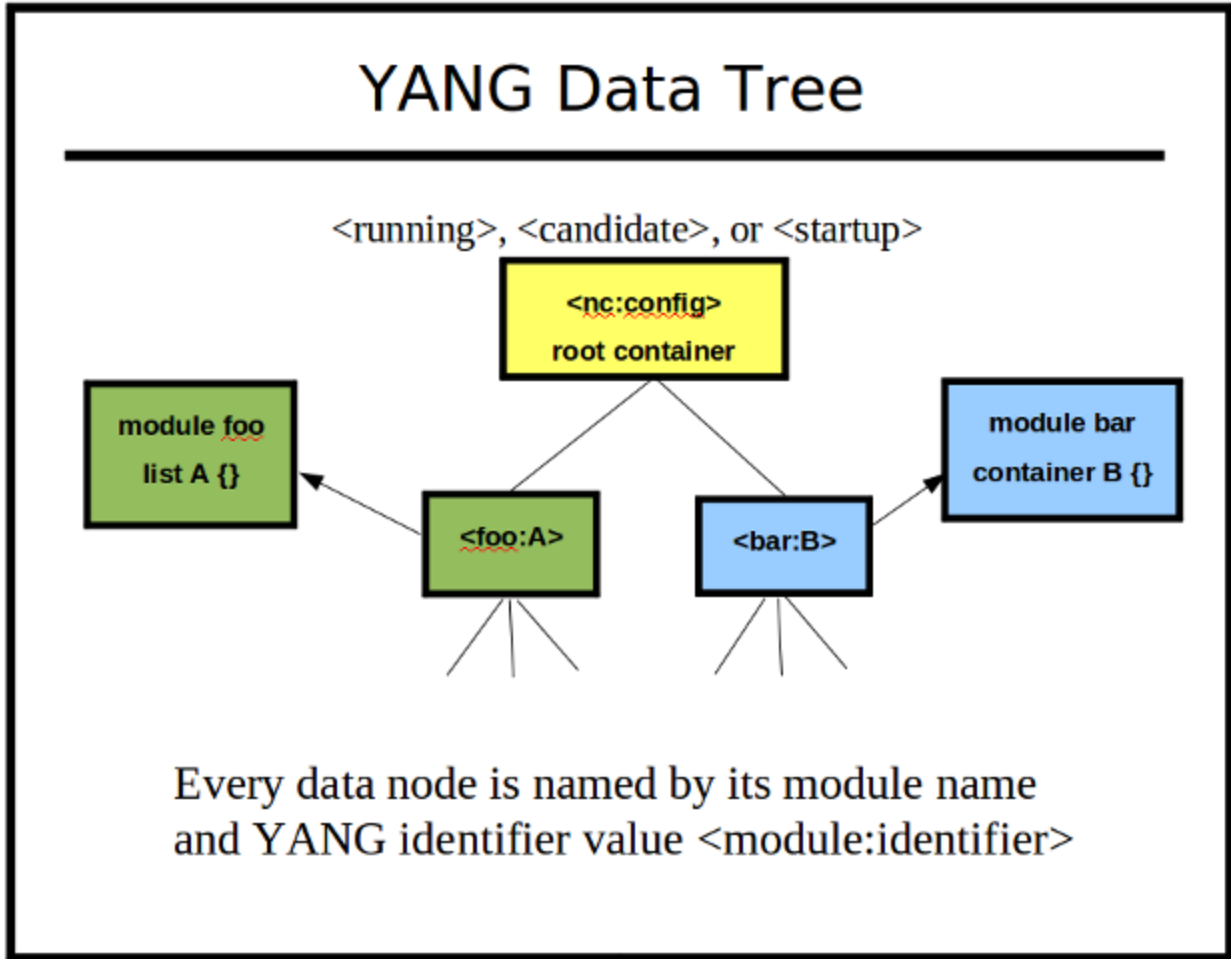
A YANG data tree represents the instances of 1 or more of the objects in the object tree.

Each NETCONF database is a separate data tree. A data tree is constructed for each incoming message as well. The server has automated functions to process the data tree, based on the desired NETCONF operation and the object tree node corresponding to each data node.

Every NETCONF node (including database nodes) are distinguished with XML Qualified Names (QName). The YANG module namespace is used as the XML namespace, and the YANG identifier is used as the XML local name.

Each data node contains a pointer back to its object tree schema node. The value tree is comprised of the **val_value_t** structure. Only real data is actually stored in the value tree. For example, there are no data tree nodes for choices and cases. These are conceptual layers, not real layers, within the data tree.

The NETCONF server engine accesses individual SIL callback functions through the data tree and object tree. Each data node contains a pointer to its corresponding object node.

Each data node may have several different callback functions stored in the object tree node. Usually, the actual configuration value is stored in the database, However, **virtual data nodes** are also supported. These are simply placeholder nodes within the data tree, and usually used for non-configuration nodes, such as counters. Instead of using a static value stored in the data node, a callback function is used to retrieve the instrumentation value each time it is accessed.

2.3.4 SERVICE LAYERING

All of the major server functions are supported by service layers in the 'agt' or 'ncx' libraries:

- **Memory management**: macros in **platform/procdefs.h** are used instead of using direct heap functions. The macros **m__getMem** or **m__getObj** are used by Yuma code to allocate memory. Both of these functions increment a global counter called **malloc_count**. The macro **m__free** is used to delete all malloced memory. This macro increments a global counter called **free_count**. When a Yuma program exists, it checks if **malloc_count** equals **free_count**, and if not, generates an error message. If this occurs, the **MEMTRACE=1** parameter can be added to the make command to activate 'mtrace' debugging.

- **Queue management**: APIs in **ncx/dlq.h** are used for all double-linked queue management.

- **XML namespaces**: XML namespaces (including YANG module namespaces) are managed with functions in **ncx/xmlns.h**. An internal 'namespace ID is used internally instead of the actual URI.

- **XML parsing**: XML input processing is found in **ncx/xml_util.h** data structures and functions.

- **XML message processing**: XML message support is found in **ncx/xml_msg.h** data structures and functions.

- **XML message writing with access control**: XML message generation is controlled through API functions located in **ncx/xml_wr.h**. High level (value tree output) and low-level (individual tag output) XML output functions are provided, which hide all namespace, indentation, and other details. Access control is integrated into XML message output to enforce the configured data access policies uniformly for all RPC operations and notifications. The access control model cannot be bypassed by any dynamically loaded module server instrumentation code.

- **XPath Services**: All NETCONF XPath filtering, and all YANG XPath-based constraint validation, is handled with common data structures and API functions. The XPath 1.0 implementation is native to the server, and uses the object and value trees directly to generate XPath results for NETCONF and YANG purposes. NETCONF uses XPath differently than XSLT, and libxml2 XPath processing is memory intensive. These functions are located in **ncx/xpath.h**, **ncx/xpath1.h**, and **ncx/xpath_yang.h**. XPath filtered <get> responses are generated in **agt/agt_xpath.c**.

- **Logging service**: Encapsulates server output to a log file or to the standard output, filtered by a configurable log level. Located in **ncx/log.h**. In addition, the macro **SET_ERROR()** in **ncx/status.h** is used to report programming errors to the log.

- **Session management**: All server activity is associated with a session. The session control block and API functions are located in **ncx/ses.h**. All input, output, access control, and protocol operation support is controlled through the session control block (ses_cb_t).

- **Timer service**: A periodic timer service is available to SIL modules for performing background maintenance within the main service loop. These functions are located in **agt/agt_timer.h**.

- **Connection management**: All TCP connections to the netconfd server are controlled through a main service loop, located in **agt/agt_ncxserver.c**. It is expected that the 'select' loop in this file will be replaced in embedded systems. The default netconfd server actually listens for local <ncx-connect> connections on an AF_LOCAL socket. The openSSH server listens for connections on port 830 (or other configured TCP ports), and the netconf-subsystem thin client acts as a conduit between the SSH server and the **netconfd** server.

- **Database management**: All configuration databases use a common configuration template, defined in **ncx/cfg.h**. Locking and other generic database functions are handled in this module. The actual manipulation of the value tree is handled by API functions in **ncx/val.h**, **ncx/val_util.h, agt/agt_val_parse.h,** and **agt/agt_val.h**.

- **NETCONF operations**: All standard NETCONF RPC callback functions are located in **agt/agt_ncx.c**. All operations are completely automated, so there is no server instrumentation APIs in this file.

- **NETCONF request processing**: All <rpc> requests and replies use common data structures and APIs, found in **ncx/rpc.h** and **agt/agt_rpc.h**. Automated reply generation, automatic filter processing, and message state data is contained in the RPC message control block.

- **NETCONF error reporting:** All <rpc-error> elements use common data structures defined in ncx/rpc_err,h and agt/agt_rpcerr.h. Most errors are handled automatically, but 'description statement' semantics need to be enforced by the SIL callback functions. These functions use the API functions in **agt/agt_util.h** (such as agt_record_error) to generate data structures that will be translated to the proper <rpc-error> contents when a reply is sent.

- **YANG module library management:** All YANG modules are loaded into a common data structure (ncx_module_t) located in ncx/ncxtypes.h.   The API functions in **ncx/ncxmod.h** (such as ncxmod_load_module) are used to locate YANG modules, parse them, and store the internal data structures in a central library. Multiple versions of the same module can be loaded at once, as required by YANG.

2.3.5 SESSION CONTROL BLOCK

Once a NETCONF session is started, it is assigned a session control block for the life of the session. All NETCONF and system activity in driven through this interface, so the **ncxserver** loop can be replaced in an embedded system.

Each session control block (ses_scb_t) controls the input and output for one session, which is associated with one SSH user name. Access control (see yuma-nacm.yang) is enforced within the context of a session control block. Unauthorized return data is automatically removed from the response. Unauthorized <rpc> or database write requests are automatically rejected with an 'access-denied' error-tag.

The user preferences for each session are also stored in this data structure. They are initially derived from the server default values, but can be altered with the <set-my-session> operation and retrieved with the <get-my-session> operation.

2.3.6 SERVER MESSAGE FLOWS
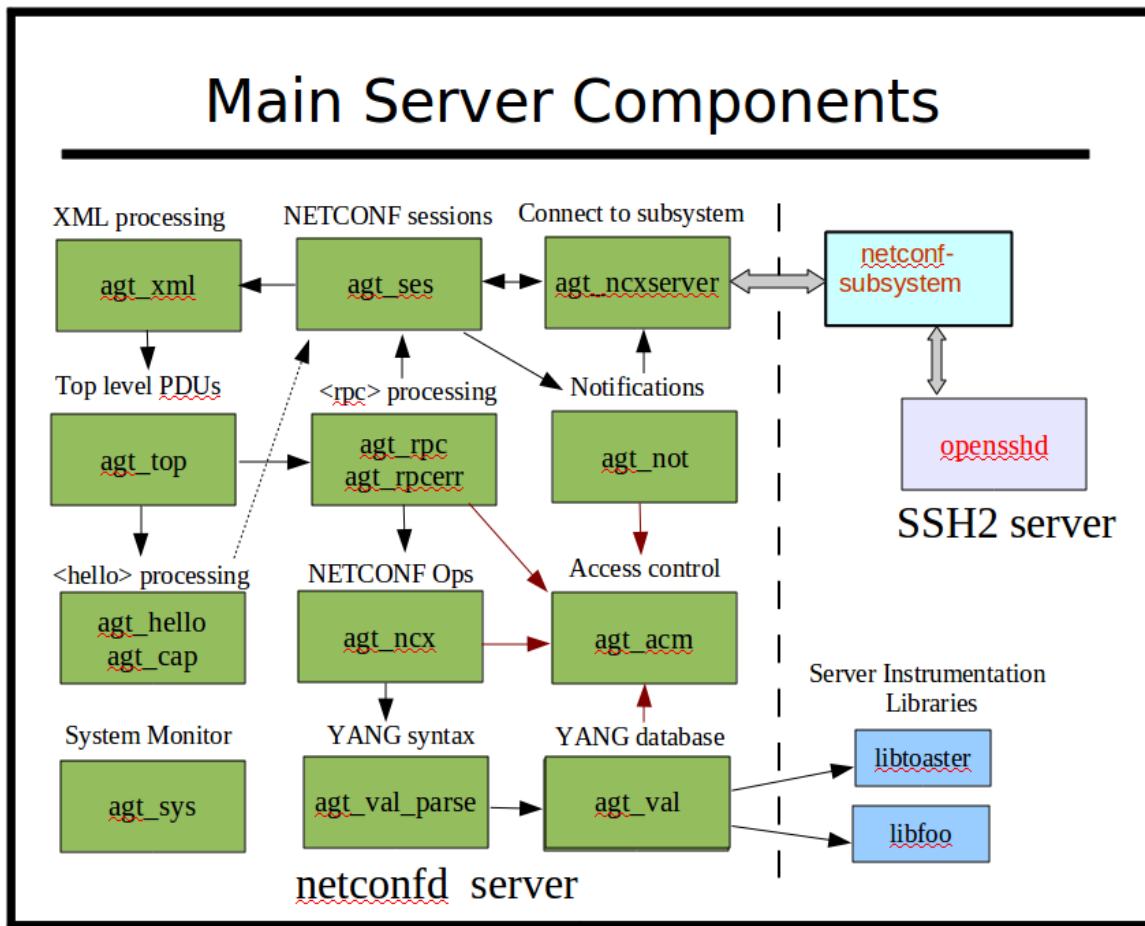
The **netconfd** server provides the following type of components:
- NETCONF session management
- NETCONF/YANG database management
- NETCONF/YANG protocol operations
- Access control configuration and enforcement
- RPC error reporting
- Notification subscription management
- Default data retrieval processing
- Database editing
- Database validation
- Subtree and XPath retrieval filtering
- Dynamic and static capability management
- Conditional bbject management (if-feature, when)
- Memory management
- Logging management

- Timer services

All NETCONF and YANG protocol operation details are handled automatically within the **netconfd** server. All database locking and editing is also handled by the server. There are callback functions available at different points of the processing model for your module specific instrumentation code to process each server request, and/or generate notifications. Everything except the 'description statement' semantics are usually handled

The server instrumentation stub files associated with the data model semantics are generated automatically with the **yangdump** program. The developer fills in server callback functions to activate the networking device behavior represented by each YANG data model.

2.3.7   MAIN NCXSERVER LOOP
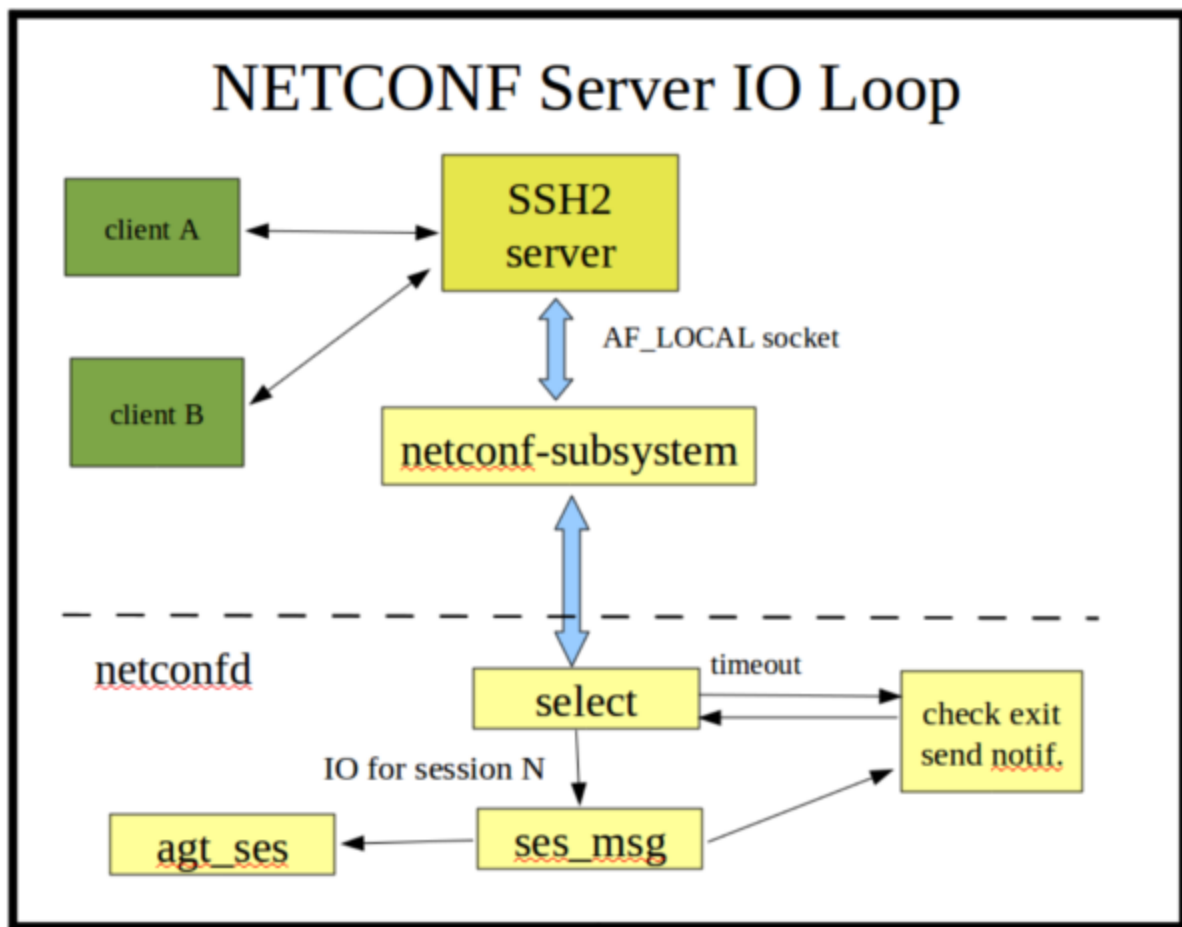
The **ncxserver** loop does very little, and it is designed to be replaced in an embedded server that has its own SSH server:

- A client request to start an SSH session results in an SSH channel being established to an instance of the **netconf-subsystem** program.

- The **netconf-subsystem** program will open a local socket (/tmp/ncxserver.sock) and send a proprietary <ncxconnect> message to the netconfd server, which is listening on this local socket with a select loop (in agt_ncxserver.c).

- When a valid <ncxconnect> message is received by **netconfd**, a new NETCONF session is created.

- After sending the <ncxconnect> message, the **netconf-subsystem** program goes into 'transfer mode', and simply passes input from the SSH channel to the **netconfd** server, and passes output from the **netconfd** server to the SSH server.

- The **ncxserver** loop simply waits for input on the open connections, with a quick timeout. Each timeout, the server checks if a reboot, shutdown, signal, or other event occurred that needs attention.

- Notifications may also be sent during the timeout check, if any events are queued for processing. The **--max-burst** configuration parameter controls the number of notifications sent to each notification subscription, during this timeout check.

- Input <rpc> messages are buffered, and when a complete message is received (based on the NETCONF End-of-Message marker), it is processed by the server and any instrumentation module callback functions that are affected by the request.

When the agt_ncxserver_run function in agt/agt_ncxserver.c is replaced within an embedded system, the replacement code must handle the following tasks:

- Call **agt_ses_new_session** in **agt/agt_ses.c** when a new NETCONF session starts.

- Call **ses_accept_input** in **ncx/ses.c** with the correct session control block when NETCONF data is received.

- Call **agt_ses_process_first_ready** in **agt/agt_ses.c** after input is received. This should be called repeatedly until all serialized NETCONF messages have been processed.

- Call **agt_ses_kill_session** in **agt/agt_ses.c** when the NETCONF session is terminated.

- The following functions are used for sending NETCONF responses, if responses are buffered instead of sent directly (streamed).

  ○ **ses_msg_send_buffs** in **ncx/ses_msg.c** is used to output any queued send buffers.

- The following functions need to be called periodically:

  ○ **agt_shutdown_requested** in **agt/agt_util.c** to check if the server should terminate or reboot

  ○ **agt_ses_check_timeouts** in **agt/agt_ses.c** to check for idle sessions or sessions stuck waiting for a NETCONF <hello> message.

  ○ **agt_timer_handler** in **agt/agt_timer.c** to process server and SIL periodic callback functions.

  ○ **send_some_notifications** in **agt/agt_ncxserver.c** to process some outgoing notifications.

2.3.8   SIL CALLBACK FUNCTIONS
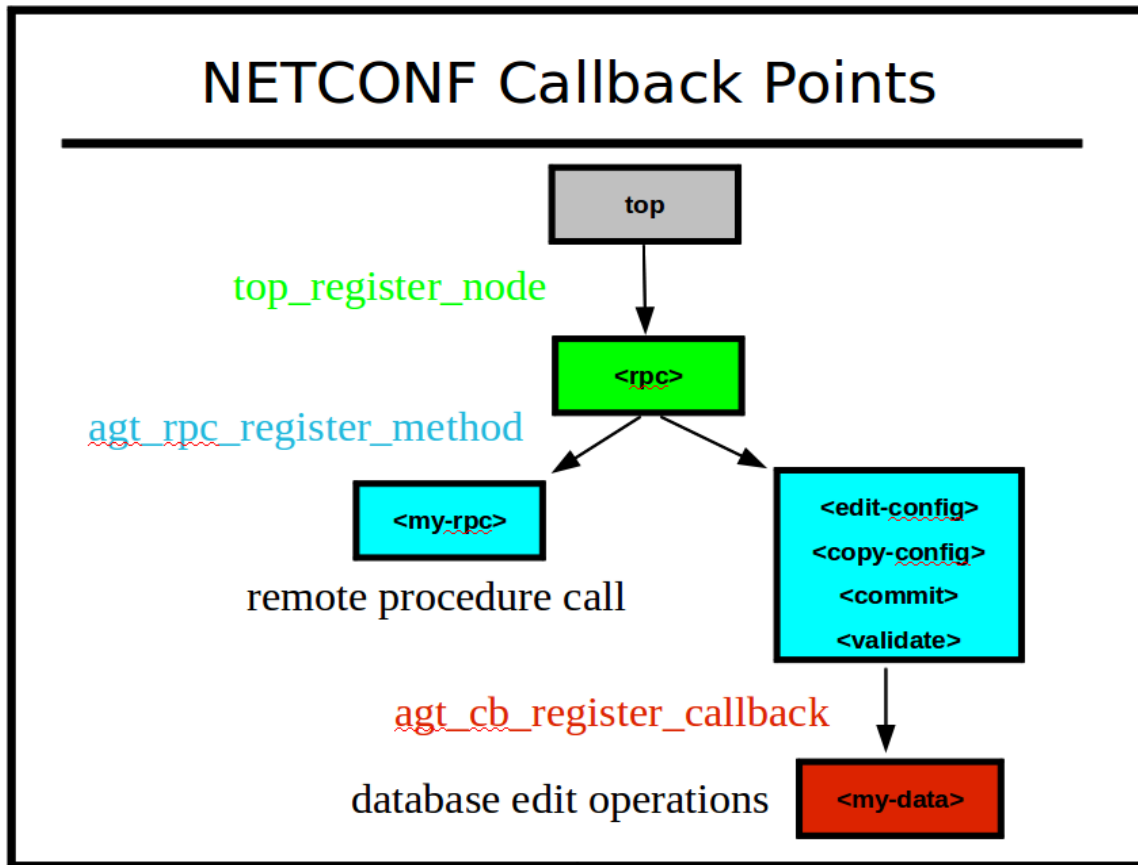
- Top Level: The top-level incoming messages are registered, not hard-wired, in the server message processing design. The **agt_ncxserve**r module accepts the <ncxconnect> message from **netconf-subsystem**. The **agt_rpc** module accepts the NETCONF <rpc> message. Additional messages can be supported by the server using the **top_register_node** function.

- All RPC operations are implemented in a data-driven fashion by the server. Each NETCONF operation is handled by a separate function in **agt_ncx.c**. Any proprietary operation can be automatically supported, using the **agt_rpc_register_method** function.

  ◦ Note: Once the YANG module is loaded into the server, all RPC operations defined in the module are available. If no SIL code is found, these will be dummy 'no-op' functions. This mode can be used to provide some server simulation capability for client applications under development.

- All database operations are performed in a structured manner, using special database access callback functions. Not all database nodes need callback functions. One callback function can be used for each 'phase', or the same function can be used for multiple phases. The **agt_cb_register_callback** function in **agt/agt_cb.c** is used by SIL code to hook into NETCONF database operations.

2.4   Server Operation

This section briefly describes the server internal behavior for some basic NETCONF operations.

2.4.1   INITIALIZATION

The file **netconfd/netconfd.c** contains the initial 'main' function that is used to start the server.

- The common services support for most core data structures is located in 'libncx.so'. The '**ncx_init**' function is called to setup these data structures. This function also calls the bootstrap_cli function in ncx/ncx.c, which processes some key configuration parameters that need to be set right away, such as the logging parameters and the module search path.

- Most of the actual server code is located in the 'agt' directory. The '**agt_init1**' function is called to initialize core server functions. The configuration parameters are processed, and the server profile is completed.

  ○ The **agt_profile_t** data structure in agt/agt.h is used to contain all the vendor-related boot-time options, such as the database target (candidate or running). The **init_server_profile** function can be edited if the Yuma default values are not desired. This will insure the proper factory defaults for server behavior are used, even if no configuration parameters are provided.

- The function **init_server_profile** in **agt/agt.c** is used to set the factory defaults for the server behavior.

The **agt_init1** function also loads the core NETCONF protocol, **netconfd** CLI, and YANG data type modules.

- Note: **netconfd** uses **yuma-netconf.yang**, not **ietf-netconf.yang** to support a data-driven implementation. The only difference is that the yuma version adds some data structures and extensions (such as ncx:root), to automate processing of all NETCONF messages.

After the core definition modules are loaded successfully, the **agt_cli_process_input** function in **agt/agt_cli.c** is called to process any command line and/or configuration file parameters that have been entered.

- Note: Any defaults set in the G module definitions will be added to the CLI parameter set. The **val_set_by_default** function in **ncx/val.c** can be used to check if the node is set by the server to the YANG default value. If not set, and the node has the YANG default value, then the client set this value explicitly. This is different than the **val_is_default** function in **ncx/val.c**, which just checks if the node contains the YANG default value.

All the configuration parameters are saved, and those that can be processed right away are handled. The **agt_cli_get_valset** function in **agt/agt_cli.c** can be used to retrieve the entire set of load-time configuration parameters.

2.4.2   LOADING MODULES AND SIL CODE

YANG modules and their associated device instrumentation can be loaded dynamically with the **--module** configuration parameter. Some examples are shown below:

```
module=foo

module=bar

module=baz@2009-01-05

module=~/mymodules/myfoo.yang
```

- The **ncxmod_find_sil_file** function in **ncx/ncxmod.c** is used to find the library code associated with the each module name. The following search sequence is followed:

  ○ Check the **$YUMA_HOME/target/lib** directory

  ○ Check each directory in the **$YUMA_RUNPATH** environment variable or **--runpath** configuration variable.

- ◦ Check the **/usr/lib/yuma** directory

- If the module parameter contains any sub-directories or a file extension, then it is treated as a file, and the module search path will not be used. Instead the absolute or relative file specification will be used.

- If the first term starts with an environment variable or the tilde (~) character, and will be expanded first

- If the 'at sign' (@) followed by a revision date is present, then that exact revision will be loaded.

- If no file extension or directories are specified, then the module search path is checked for YANG and YIN files that match. The first match will be used, which may not be the newest, depending on the actual search path sequence.

- The **$YUMA_MODPATH** environment variable or **--modpath** configuration parameter can be used to configure one or more directory sub-trees to be searched.

- The **$YUMA_HOME** environment variable or **--yuma-home** configuration parameter can be used to specify the Yuma project tree to use if nothing is found in the currect directory or the module search path.

- The **$YUMA_INSTALL** environment variable or default Yuma install location (/usr/share/yuma/modules) will be used as a last resort to find a YANG or YIN file.

The server processes **--module** parameters by first checking if a dynamic library can be found which has an 'soname' that matches the module name. If so, then the SIL phase 1 initialization function is called, and that function is expected to call the ncxmod_load_module function.

If no SIL file can be found for the module, then the server will load the YANG module anyway, and support database operations for the module, for provisioning purposes. Any RPC operations defined in the module will also be accepted (depending on access control settings), but the action will not actually be performed. Only the input parameters will be checked, and <or> or some <rpc-error> returned.

2.4.3   CORE MODULE INITIALIZATION

The **agt_init2** function in **agt/agt.c** is called after the configuration parameters have been collected.

- ◦ Initialize the core server code modules

- ◦ Static device-specific modules can be added to the agt_init2 function after the core modules have been initialized

- ◦ Any 'module' parameters found in the CLI or server configuration file are processed.

- ◦ The **agt_cap_set_modules** function in **agt/agt_cap.c** is called to set the initial module capabilities for the **ietf-netconf-monitoring** module

2.4.4   STARTUP CONFIGURATION PROCESSING

After the static and dynamic server modules are loaded, the **--startup** (or **--no-startup**) parameter is processed by **agt_init2** in **agt/agt.c**:

- If the --startup parameter is used and includes any sub-directories, it is treated as a file and must be found, as specified.

- Otherwise, the **$YUMA_DATAPATH** environment variable or **--datapath** configuration parameter can be used to determine where to find the startup configuration file.

- If neither the **--startup** or **--no-startup** configuration parameter is present, then the data search path will be used to find the default **startup-cfg.xml**

- The **$YUMA_HOME** environment variable or --yuma-home configuration parameter is checked if no file is found in the data search path. The **$YUMA_HOME/data** directory is checked if this parameter is set.

- The **$YUMA_INSTALL** environment variable or default location (/etc/yuma/) is checked next, if the startup configuration is still not found.

It is a fatal error if a startup config is specified and it cannot be found.

As the startup configuration is loaded, any SIL callbacks that have been registered will be invoked for the association data present in the startup configuration file.. The edit operation will be OP_EDITOP_LOAD during this callback.

After the startup configuration is loaded into the running configuration database, all the stage 2 initialization routines are called. These are needed for modules which add read-only data nodes to the tree containing the running configuration. SIL modules may also use their 'init2' function to create factory default configuration nodes (which can be saved for the next reboot).

2.4.5 PROCESS AN INCOMING <RPC> REQUEST

5 Phase RPC Processing Model
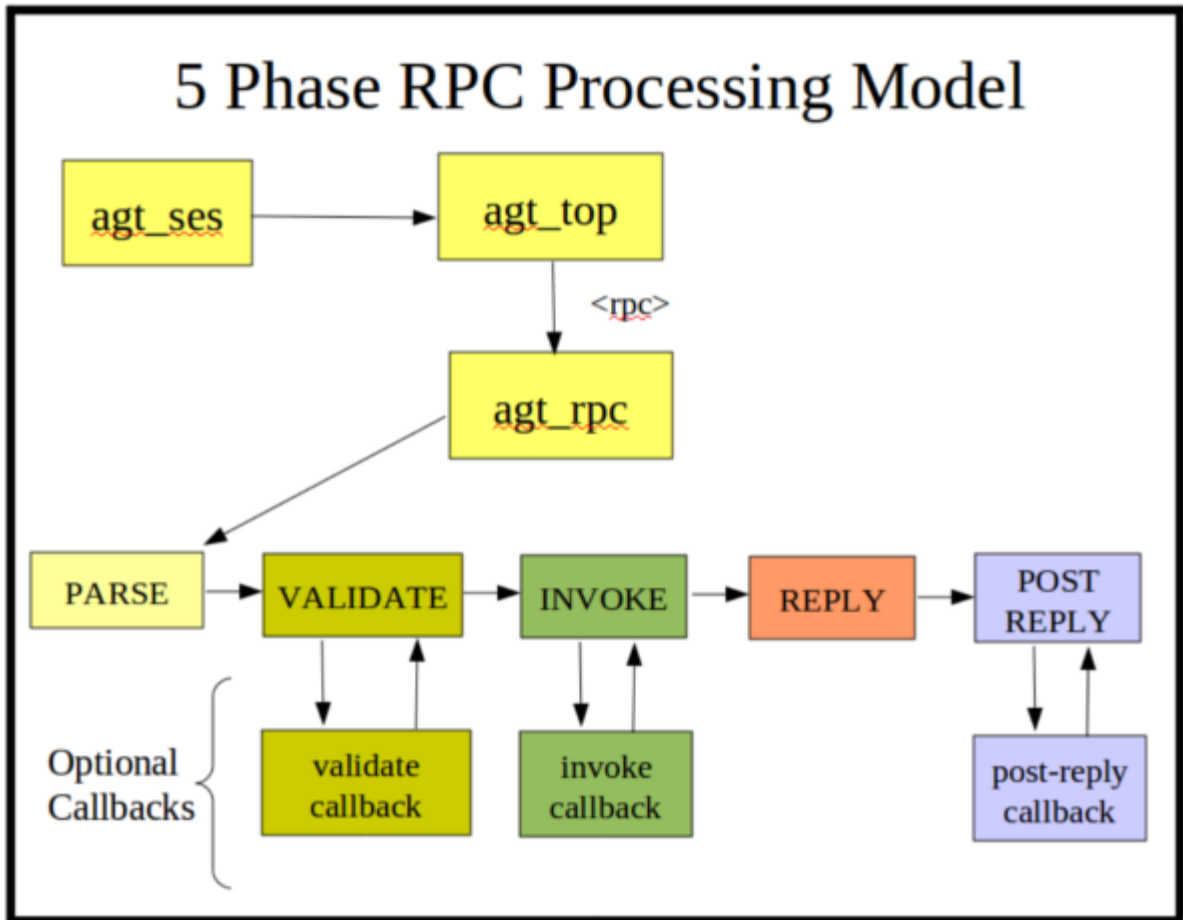
- **PARSE Phase:** The incoming buffer is converted to a stream of XML nodes, using the **xmlTextReader** functions from **libxml2**. The agt_val_parse function is used to convert the stream of XML nodes to a **val_value_t** structure, representing the incoming request according to the YANG definition for the RPC operation. An **rpc_msg_t** structure is also built for the request.

- **VALIDATE Phase:** If a message is parsed correctly, then the incoming message is validated according to the YANG machine-readable constraints. Any description statement constraints need to be checked with a callback function. The **agt_rpc_register_method** function in **agt/agt_rpc.c** is used to register callback functions.

- **INVOKE Phase:** If the message is validated correctly, then the invoke callback is executed. This is usually the only required callback function. Without it, the RPC operation has no affect. This callback will set fields in the **rpc_msg_t** header that will allow the server to construct or stream the <rpc-reply> message back to the client.

- **REPLY Phase:** Unless some catastrophic error occurs, the server will generate an <rpc-reply> response. If any <rpc-error> elements are needed, they are generated first. If there is any response data to send, that is generated or streamed (via callback function provided earlier) at this time. Any unauthorized data (according to to the **yuma-nacm.yang** module configuration) will be silently dropped from the message payload. If there were no errors and no data to send, then an <ok> resonse is generated.

- **POST_REPLY Phase:** After the response has been sent, a rarely-used callback function can be invoked to cleanup any memory allocation or other data-model related tasks. For example, if

the **rpc_user1** or **rpc_user2** pointers in the message header contain allocated memory then they need to be freed at this time.

2.4.6   EDIT THE DATABASE

3 Phase Database Editing Model
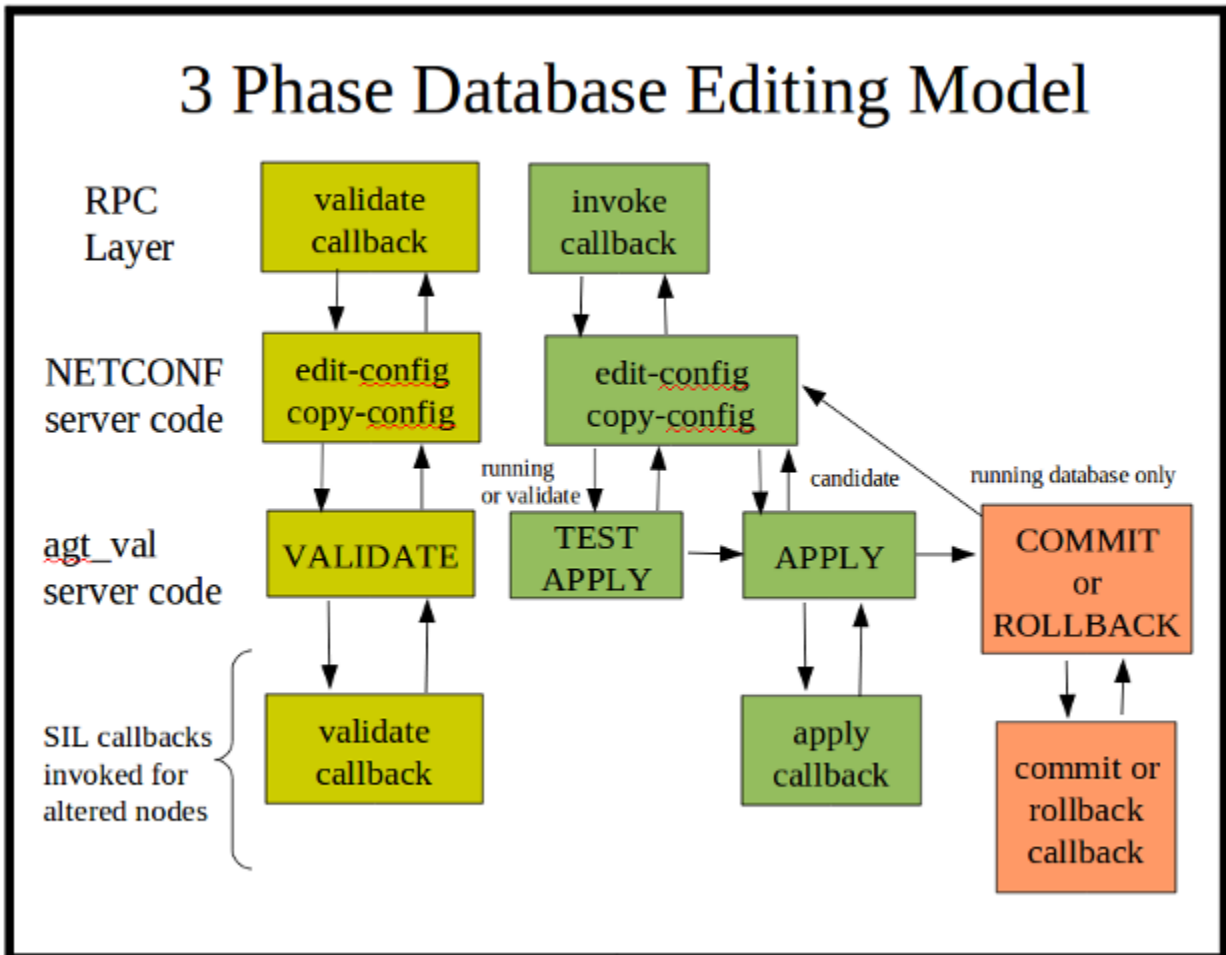
- **Validate Phase:** The server will determine the edit operation and the actual nodes in the target database (candidate or running) that will be affected by the operation. All of the machine-readable YANG statements which apply to the affected node(s) are tested against the incoming PDU and the target database. If there are no errors, the server will search for a SIL validate callback function for the affected node(s). If the SIL code has registered a database callback function for the node or its local ancestors, it will be invoked. This SIL callback function usually checks additional constraints that are contained in the YANG description statements for the database objects.

- **Test-Apply and Apply Phase:** If the validate phase completes without errors, then the requested changes are applied to the target database. If the target database is the running configuration, or if the edit-config 'test-option' parameter is set to 'test-then-set' (the default if **--with-validate**=true), then the test-apply phase is executed first. This is essentially the same as the real apply phase, except that changes are made to a copy of the target database. Once all objects have been altered as requested, the entire test database is validated, including all

cross-referential integrity tests. If this test completes without any errors, then the procedure is repeated on the real target database.

- ◦ Note: This phase is used for the internal data tree manipulation and validation only. It is not used to alter device behavior. Resources may need to be reserved during the SIL apply callback, but the database changes are not activated at this time.

- **Commit or Rollback Phase:** If the validate and apply phases complete without errors, then then the server will search for SIL commit callback functions for the affected node(s) in the target database. This SIL callback phase is used to apply the changes to the device and/or network. It is only called when a commit procedure is attempted. This can be due to a <commit> operation, or an <edit-config> or <copy-config> operation on the running database.

  - ◦ Note: If there are errors during the commit phase, then the backup configuration will be applied, and the server will search for a SIL callback to invoke with a 'rollback operation'. The same procedure is used for confirmed commit operations which timeout or canceled by the client.

2.4.7 SAVE THE DATABASE

The following bullets describe how the server saves configuration changes to non-volatile storage:

- If the **--with-startup=true** parameter is used, then the server will support the :startup capability. In this case, the <copy-config> command needs to be used to cause the running configuration to be saved.

- If the **--with-startup=false** parameter is used, then the server will not support the :startup capability. In this case, the database will be saved each time the running configuration is changed.

- The <copy-config> or <commit> operations will cause the startup configuration file to be saved, even if nothing has changed. This allows an operator to replace a corrupted or missing startup configuration file at any time.

- The database is saved with the **agt_ncx_cfg_save** function in **agt/agt_ncx.c.**

- The **with-defaults** 'explicit' mode is used during the save operation to filter the database contents.

  - ◦ Any values that have been set by the client will be saved in NV-storage.

  - ◦ Any value set by the server to a YANG default value will not be saved in the database.

  - ◦ If the server create a node that does not have a YANG default value (e.g., containers, lists, keys), then this node will be saved in NV storage.

- If the **--startup=filespec** parameter is used, then the server will save the database by overwriting that file. The file will be renamed to backup-cfg.xml first.

- If the **--no-startup** parameter is used, or no startup file is specified and no default is found, then the server will create a file called 'startup-cfg.xml', in the following manner:

  - ◦ If the **$YUMA_HOME** variable is set, the configuration will be saved in **$YUMA_HOME/data/startup-cfg.xml**.

  - ◦ Otherwise, the configuration will be saved in **$HOME/.yuma/startup-cfg.xml.**

- The database is saved as an XML instance document, using the <config> element in the NETCONF 'base' namespace as the root element. Each top-level YANG module supported by the server, which contains some explicit configuration data, will be saved as a child node of the <nc:config> element. There is no particular order to the top-level data model elements.

2.5   Built-in Server Modules

There are several YANG modules which are implemented within the server, and not loaded at run-time like a dynamic SIL module. Some of them are NETCONF standard modules and some are Yuma extension modules.

2.5.1   IETF-INET-TYPES.YANG

This module contains the standard YANG Internet address types. These types are available for commonly used management object types. A YANG module author should check this module first, before creating any new data types with the YANG typedef statement.

There are no accessible objects in this module, so there are no SIL callback functions. The YANG data-types are supported within the Yuma engine core modules, such as **ncx/val.c** and **ncx/xml_wr.c.**

2.5.2   IETF-NETCONF-MONITORING.YANG

The standard NETCONF Monitoring module is used to examine the capabilities, current state, and statistics related to the NETCONF server. The entire module is supported.

This module is also used to retrieve the actual YANG or YIN files (or URLs for them) that the server is using. Clients can use the <get-schema> RPC operation to retrieve the YANG or YIN files listed in the **/netconf-state/schemas** subtree. A client will normally check the <hello> message from the server for module capabilities, and use its own local copy of a server YANG module, if it can. If not, then the <get-schema> function can be used to retrieve the YANG module.

The **agt/agt_state.c** contains the SIL callback functions for this module.

2.5.3   IETF-WITH-DEFAULTS.YANG

The standard <with-defaults> extension to some NETCONF operations is defined in this module. This parameter is added to the <get>, <get-config>, and <copy-config> operations to let the client control how 'default leafs' are returned by the server. The Yuma server can be configured to use any of the default handling styles (report-all, trim, or explicit). The filtering of default nodes is handled automatically by the server support functions in **agt/agt_util.c**, and the XML write functions in **ncx/xml_wr.c.**

2.5.4   IETF-YANG-TYPES.YANG

This module contains the standard YANG general user data types. These types are available for commonly used derived types. A YANG module author should check this module first, before creating any new data types with the YANG typedef statement.

There are no accessible objects in this module, so there are no SIL callback functions. The YANG data-types are supported within the Yuma engine core modules, such as **ncx/val.c** and **ncx/xml_wr.c.**

2.5.5   NC-NOTIFICATIONS.YANG

This module is defined in RFC 5277, the NETCONF Notifications specification. It contains the <replayComplete> and <notificationComplete> notification event definitions.

The file **agt/agt_not.c** contains the SIL support code for this module.

2.5.6   NOTIFICATIONS.YANG

This module is defined in RFC 5277, the NETCONF Notifications specification. All of this RFC is supported in the server. This module contains the <create-subscription> RPC operation. The

notification replay feature is controlled with the **--eventlog-size** configuration parameter. The <create-subscription> operation is fully supported, including XPath and subtree filters. The **yuma-nacm** module can be used to control what notification events a user is allowed to receive. The <create-subscription> filter allows the client to select which notification events it wants to receive.

The file **agt/agt_not.c** contains the SIL callback functions for this modules.

2.5.7   YUMA-APP-COMMON.YANG

This module contains some common groupings of CLI parameters supported by some or all Yuma programs. Each program with CLI parameters defines its own module of CLI parameters (using the ncx:cli extension). The program name is used for the YANG module name as well (e.g., yangdump.yang or netconfd.yang).

The SIL callback functions for the common groupings in this module are found in **ncx/val_util.c**, such as the **val_set_feature_parms** function.

2.5.8   YUMA-INTERFACES.YANG

This module contains the Yuma interfaces table, which is just a skeleton configuration list, plus some basic interface counters. This module is intended to provide an example for embedded developers to replace this module with their own interfaces table. The Yuma table uses information in some files found in Unix systems which support the **/proc/net/dev** system file.

The file **agt/agt_if.c** contains the SIL callback functions for this module.

2.5.9   YUMA-MYSESSION.YANG

This module provides the Yuma proprietary <get-my-session> and <set-my-session> RPC operations. These are used by the client to set some session output preferences, such as the desired line length, indentation amount, and defaults handling behavior.

The file **agt/agt_ses.c** contains the SIL callback functions for this module.

2.5.10   YUMA-NACM.YANG

This module contains the Yuma NETCONF Access Control Model implementation. It provides all user-configurable access control settings and also provides API functions to check if a specific access request should be allowed or not.

The file **agt/agt_acm.c** contains the SIL callback functions for this module.

2.5.11   YUMA-NCX.YANG

This module provides the YANG language extension statements that are used by Yuma programs to automate certain parts of the NETCONF protocol, document generation, code generation, etc.

There are no SIL callback functions for this module. There are support functions within the **src/ncx** directory that include the **obj_set_ncx_flags** function in **ncx/obj.c**

2.5.12   YUMA-NETCONF.YANG

The NETCONF protocol operations, message structures, and error information are all data-driven, based on the YANG statements in the **yuma-netconf.yang** module. The ietf-netconf.yang module is not used at this time because it does not contain the complete set of YANG statements needed. The yuma-

netconf.yang version is a super-set of the IETF version. Only one YANG module can be associated with an XML namespace in Yuma. In a future version, the extra data structures will be moved to an annotation module.

The file **agt/agt_ncx.c** contains the SIL callback functions for this module.

This module is not advertised in the server capabilities. It is only used internally within the server.

2.5.13   YUMA-PROC.YANG

This module provides some Unix **/proc** file-system data, in nested XML format. This module will not load if the files **/proc/meminfo** and **/proc/cpuinfo** are not found.

The file **agt/agt_proc.c** contains the SIL callback functions for this module.

2.5.14   YUMA-SYSTEM.YANG

This module contains the Yums /system data structure, providing basic server information, unix 'uname' data, and all the Yuma proprietary notification event definitions.

The file **agt/agt_sys.c** contains the SIL callback functions for this module.

2.5.15   YUMA-TYPES.YANG

This module provides some common data types that are used by other Yuma YANG modules.

There are no SIL callback functions for this module.

3   YANG Objects and Data Nodes

This section describes the basic design of the YANG object tree and the corresponding data tree that represents instances of various object nodes that the client or the server can create.

3.1   Object Definition Tree

The object tree is a tree representation of all the YANG module rpc, data definition, and notification statements. It starts with a 'root' container. This is defined with a YANG container statement which has an **ncx:root** extension statement within it. The <config> parameter within the <edit-config> operation is an example of an object node which is treated as a root container. Each configuration database maintained by the server (e.g., <candidate> and <running>) has a root container value node as its top-level object.

A root container does not have any child nodes defined in it within the YANG file. However, the Yuma tools will treat this special container as if any top-level YANG data node is allowed to be a child node of the 'root' container type.

3.1.1   OBJECT NODE TYPES

There are 14 different YANG object node types, and a discriminated union of sub-data structures contains fields common to each sub-type. Object templates are defined in ncx/obj.h.

YANG Object Types

object type	description
OBJ_TYP_ANYXML	This object represents a YANG **anyxml** data node.
OBJ_TYP_CONTAINER	This object represents a YANG presence or non-presence **container.**
OBJ_TYP_CONTAINER + ncx:root	If the ncx:root extension is present within a container definition, then the object represents a NETCONF database **root**. No child nodes
OBJ_TYP_LEAF	This object represents a YANG **leaf** data node.
OBJ_TYP_LEAF_LIST	This object represents a YANG **leaf-list** data node.
OBJ_TYP_LIST	This object represents a YANG **list** data node.
OBJ_TYP_CHOICE	This object represents a YANG **choice** schema node. The only children allowed are case objects. This object does not have instances in the data tree.
OBJ_TYP_CASE	This object represents a YANG **case** schema node. This object does not have instances in the data tree.
OBJ_TYP_USES	This object represents a YANG **uses** schema node. The contents of the grouping it represents will be expanded into object tree. It is saved in the object tree even during operation, in order for the expanded objects to share common data. This object does not have instances in the data tree.
OBJ_TYP_REFINE	This object represents a YANG **refine** statement. It is used to alter the grouping contents during the expansion of a uses statement. This object is only allowed to be a child of a uses statement. It does not have instances in the data tree.
OBJ_TYP_AUGMENT	This object represents a YANG **augment** statement. It is used to add additional objects to an existing data structure. This object is only allowed to be a child of a uses statement or a child of a 'root' container. It does not have instances in the data tree, however any children of the augment node will generate object nodes that have instances in the data tree.
OBJ_TYP_RPC	This object represents a YANG **rpc** statement. It is used to define new <rpc> operations. This object will only appear as a child of a 'root' container. It does not have instances in the data tree. Only 'rpcio' nodes are allowed to be children of an RPC node.
OBJ_TYP_RPCIO	This object represents a YANG **input** or **output** statement. It is used to define new <rpc> operations. This object will only appear as a child of an RPC node. It does not have instances in the data tree.
OBJ_TYP_NOTIF	This object represents a YANG **notification**

	statement. It is used to define new <notification> event types. This object will only appear as a child of a 'root' container. It does not have instances in the data tree.

3.1.2   OBJECT NODE TEMPLATE (OBJ_TEMPLATE_T)

The following typedef is used to represent an object tree node:

```
/* One YANG data-def-stmt */
typedef struct obj_template_t_ {
    dlq_hdr_t      qhdr;
    obj_type_t     objtype;
    uint32         flags;                /* see OBJ_FL_* definitions */
    ncx_error_t    tkerr;
    grp_template_t *grp;          /* non-NULL == in a grp.datadefQ */


    /* 4 back pointers */
    struct obj_template_t_ *parent;
    struct obj_template_t_ *usesobj;
    struct obj_template_t_ *augobj;
    struct xpath_pcb_t_    *when;         /* optional when clause */
    dlq_hdr_t              metadataQ;      /* Q of obj_metadata_t */
    dlq_hdr_t              appinfoQ;        /* Q of ncx_appinfo_t */
    dlq_hdr_t              iffeatureQ;    /* Q of ncx_iffeature_t */


    /* cbset is agt_rpc_cbset_t for RPC or agt_cb_fnset_t for OBJ */
    void                *cbset;


    /* object namespace ID assigned at runtime
     * this can be changed over and over as a
     * uses statement is expanded.  The final
     * expansion into a real object will leave
     * the correct value in place
     */
    xmlns_id_t            nsid;


    union def_ {
     obj_container_t   *container;
     obj_leaf_t        *leaf;
     obj_leaflist_t    *leaflist;
     obj_list_t        *list;
```

```
      obj_choice_t      *choic;

      obj_case_t        *cas;

      obj_uses_t        *uses;

      obj_refine_t      *refine;

      obj_augment_t     *augment;

      obj_rpc_t         *rpc;

      obj_rpcio_t       *rpcio;

      obj_notif_t       *notif;
      } def;


} obj_template_t;
```

The following table highlights the fields within the obj_template_t data structure:

obj_template_t Fields

Field	Description
qhdr	Queue header to allow the object template to be stored in a child queue
objtype	enumeration to identify which variant of the 'def' union is present
flags	Internal state and properties
tkerr	Error message information
grp	back-pointer to parent group if this is a top-level data node within a grouping
parent	Parent node if any
usesobj	Back pointer to uses object if this is a top-level data node within an expanded grouping
augobj	Back pointer to augment object if this is a top-level data node within an expanded augment
when	XPath structure for YANG when statement
metadataQ	Queue of obj_template_t for any XML attributes (ncx:metadata) defined for this object node
appinfoQ	Queue of ncx_appinfo_t for any YANG extensions found defined within the object, that were not collected within a deeper appinfoQ (e.g., within a type statement)
iffeatureQ	Queue of ncx_iffeature_t for any if-feature statements found within this object node
cbset	Set of server callback functions for this object node.
nsid	Object node namespace ID assigned by

Field	Description
	xmlns.c
def	Union of object type specific nodes containing the rest of the YANG statements. Note that the server discards all descriptive statements such as description, reference, contact,.

3.1.3 OBJ_TEMPLATE_T ACCESS FUNCTIONS

The file **ncx/obj.h** contains many API functions so that object properties do not have to be accessed directly. The following table highlights the most commonly used functions. Refer to the H file for a complete definition of each API function.

obj_template_t Access Functions

Function	Description
obj_find_template	Find a top-level object template within a module
obj_find_child	Find the specified child node within a complex object template . Skips over any nodes without names (augment, uses, etc.)
obj_first_child	Get the first child node within a complex object template . Skips over any nodes without names.
obj_next_child	Get the next child node after the current specified child. Skips over any nodes without names.
obj_first_child_deep	Get the first child node within a complex object template . Skips over any nodes without names, and also any choice and case nodes.
obj_next_child_deep	Get the next child node after the current specified child. Skips over any nodes without names, and also any choice and case nodes.
obj_find_case	Find the specified case object child node within the specific complex object node.
obj_find_type	Check if a typ_template_t in the obj typedefQ hierarchy.
obj_find_grouping	Check if a grp_template_t in the obj typedefQ hierarchy.
obj_find_key	Find a specific key component by key leaf identifier name
obj_first_key	Get the first obj_key_t struct for the specified list object type
obj_next_key	Get the next obj_key_t struct for the specified list object type
obj_gen_object_id	Allocate and generate the YANG object ID for an object node
obj_get_name	Get the object name string

Function	Description
obj_has_name	Return TRUE if the object has a name field
obj_has_text_content	Return TRUE if the object has text content
obj_get_status	Get the YANG status for the object
obj_get_description	Get the YANG description statement for an object. Note that the server will always return a NULL pointer.
obj_get_reference	Get the YANG reference statement for an object. Note that the server will always return a NULL pointer.
obj_get_config_flag	Get the YANG config statement value for an object
obj_get_typestr	Get the name string for the type of an object
obj_get_default	Get the YANG default value for an object
obj_get_default_case	Get the name of the default case for a choice object
obj_get_typdef	Get the internal type definition for the leaf or leaf-list object
obj_get_basetype	Get the internal base type enumeration for an object
obj_get_mod_prefix	Get the module prefix for an object
obj_get_mod_name	Get the module name containing an object
obj_get_mod_version	Get the module revision date for the module containing an object.
obj_get_nsid	Get the internal XML namespace ID for an object
obj_get_min_elements	Get the YANG min-elements value for a list or leaf-list object
obj_get_max_elements	Get the YANG max-elements value for a list or leaf-list object
obj_get_units	Get the YANG units field for a leaf or leaf-list object
obj_get_parent	Get the parent object node for an object
obj_get_presence_string	Get the YANG presence statement for a container object
obj_get_child_count	Get the number of child nodes for a complex object.
obj_get_fraction_digits	Get the YANG fraction-digits statement for a decimal64 leaf or leaf-list object
obj_is_leafy	Return TRUE if the object is a leaf or leaf-list type
obj_is_mandatory	Return TRUE if the object is YANG mandatory
obj_is_mandatory_when	Return TRUE if the object is YANG mandatory, but first check if any when statements are FALSE first
obj_is_cloned	Return TRUE if the object is expanded from a grouping or augment statement
obj_is_data_db	Return TRUE if the object is defined within a YANG

Function	Description
	database definition
obj_in_rpc	Return TRUE if the object is defined within an RPC statement
obj_in_notif	Return TRUE if the object is defined within a notification statement
obj_is_hidden	Return TRUE if object contains the ncx:hidden extension
obj_is_root	Return TRUE if object contains the ncx:root extension
obj_is_password	Return TRUE if object contains the ncx:password extension
obj_is_cli	Return TRUE if object contains the ncx:cli extension
obj_is_abstract	Return TRUE if object contains the ncx:abstract extension
obj_is_xpath_string	Return TRUE if the object is a leaf or leaf-list containing an XPath string
obj_is_schema_instance_ string	Return TRUE if the object is a leaf or leaf-list containing a schema instance identifier string
obj_is_secure	Return TRUE if object contains the nacm:secure extension
obj_is_very_secure	Return TRUE if object contains the nacm:very-secure extension
obj_is_system_ordered	Return TRUE if the list or leaf-list object is system ordered; FALSE if it is user ordered
obj_is_np_container	Return TRUE if the object is a YANG non presence container
obj_is_enabled	Return TRUE if the object is enabled; FALSE if any if-feature, when-stmt, or deviation-stmt has removed the object from the system.
obj_sort_children	Rearrange any child nodes in YANG schema order

3.2   Data Tree

A Yuma data tree is a representation of some subset of all possible object instances that a server is maintaining within a configuration database or other structure.

Each data tree starts with a 'root' container, and any child nodes represent top-level YANG module data nodes that exist within the server.

Each configuration database maintains its own copy (and version) of the data tree. There is only one object tree, however, and all data trees use the same object tree for reference.

Not all object types have a corresponding node within a data tree. Only 'real' data nodes are present. Object nodes that are used as meta-data to organize the object tree (e.g., choice, augment) are not present. The following table lists the object types and whether each one is found in a data tree.

Object Types in the Data Tree

Object Type	Found In Data Tree?
OBJ_TYP_ANYXML	Yes
OBJ_TYP_CONTAINER	Yes
OBJ_TYP_CONTAINER (ncx:root)	Yes
OBJ_TYP_LEAF	Yes
OBJ_TYP_LEAF_LIST	Yes
OBJ_TYP_LIST	Yes
OBJ_TYP_CHOICE	No
OBJ_TYP_CASE	No
OBJ_TYP_USES	No
OBJ_TYP_REFINE	No
OBJ_TYP_AUGMENT	No
OBJ_TYP_RPC	No
OBJ_TYP_RPCIO	No
OBJ_TYP_NOTIF	No

3.2.1 DATA NODE TYPES

The **ncx_btype_t** enumeration in **ncx/ncxtypes.h** is used within each **val_value_t** to quickly identify which variant of the data node structure is being used.

The following table describes the different enumeration values:

Yuma Data Types (ncx_btype_t)

Data Type	Description
NCX_BT_NONE	No type has been set yet. The val_new_value() function has been called but no specific init function has been called to set the base type.
NCX_BT_ANY	The node is a YANG 'anyxml' node. When the client or server parses an 'anyxml' object, it will be converted to containers and strings. This type should not be used directly.
NCX_BT_BITS	YANG 'bits' data type
NCX_BT_ENUM	YANG 'enumeration' data type
NCX_BT_EMPTY	YANG 'empty' data type
NCX_BT_BOOLEAN	YANG 'boolean' data type
NCX_BT_INT8	YANG 'int8' data type
NCX_BT_INT16	YANG 'int16' data type
NCX_BT_INT32	YANG 'int32' data type

Data Type	Description
NCX_BT_INT64	YANG 'int64' data type
NCX_BT_UINT8	YANG 'uint8' data type
NCX_BT_UINT16	YANG 'uint16' data type
NCX_BT_UINT32	YANG 'uint32' data type
NCX_BT_UINT64	YANG 'uint64' data type
NCX_BT_DECIMAL64	YANG 'decimal64' data type
NCX_BT_FLOAT64	Hidden double type, used just for XPath. If the HAS_FLOAT #define is false, then this type will be implemented as a string, not a double.
NCX_BT_STRING	YANG 'string' type. There are also some Yuma extensions that are used with this data type for special strings. The server needs to know if a string contains XML prefixes or not, and there are several flavors to automatate processing of each one correctly.
NCX_BT_BINARY	YANG 'binary' data type
NCX_BT_INSTANCE_ID	YANG 'instance-identifier' data type
NCX_BT_UNION	YANG 'union' data type. This is a meta-type. When the client or server parses a value, it will resolve the union to one of the data types defined within the union.
NCX_BT_LEAFREF	YANG 'leafref' data type. This is a meta-type. The client or server will resolve this data type to the type of the actual 'pointed-at' leaf that is being referenced.
NCX_BT_IDREF	YANG 'identityref' data type
NCX_BT_SLIST	XSD list data type (ncx:xsdlist extension)
NCX_BT_CONTAINER	YANG container
NCX_BT_CHOICE	YANG choice. This is a meta-type and placeholder. It does not appear in the data tree.
NCX_BT_CASE	YANG case. This is a meta-type and placeholder. It does not appear in the data tree.
NCX_BT_LIST	YANG list
NCX_BT_EXTERN	Internal 'external' data type, used in yangcli. It indicates that the content is actually in an external file.
NCX_BT_INTERN	Internal 'buffer' data type, used in yangcli. The content is actually stored verbatim in an internal buffer.

3.2.2   YUMA DATA NODE EDIT VARIABLES (VAL_EDITVARS_T)

There is a temporary data structure which is attached to a data node while editing operations are in progress, called **val_editvars_t**. This structure is used by the functions in agt/agt_val.c to manipulate the value tree nodes during an <edit-config>, <copy-config>, <load-config>, or <commit> operation.

The SIL callback functions may wish to refer to the fields in this data structure. There is also a SIL cookie field to allow data to be transferred from one callback stage to the later stages. For example, if an edit operation caused the device instrumentation to reserve some memory, then this cookie could store that pointer.

The following typedef is used to define the val_editvars_t structure:

```
/* one set of edit-in-progress variables for one value node */

typedef struct val_editvars_t_ {

    /* these fields are only used in modified values before they are

     * actually added to the config database (TBD: move into struct)

     * curparent == parent of curnode for merge

     */

    struct val_value_t_  *curparent;

    op_editop_t    editop;              /* effective edit operation */

    op_insertop_t  insertop;              /* YANG insert operation */

    xmlChar        *insertstr;          /* saved value or key attr */

    struct xpath_pcb_t_ *insertxpcb;      /* key attr for insert */

    struct val_value_t_ *insertval;              /* back-ptr */

    boolean        iskey;                     /* T: key, F: value */

    boolean        operset;                /* nc:operation here */

    void           *pcookie;               /* user pointer cookie */

    int            icookie;                /* user integer cookie */
} val_editvars_t;
```

The following fields within the val_editvars_t are highlighted:

val_editvars_t Fields

Field	Description
curparent	A 'new' node will use this field to remember the parent of the 'current' value. This is needed to support the YANG insert operation.
editop	The effective edit operation for this node.
insertop	The YANG insert operation, if any.
insertstr	The YANG 'value' or 'key' attribute value string, used to support the YANG insert operation.
insertxpcb	XPath parser control block for the insert 'key' expression, if needed. Used to support the YANG insert operation.
insertval	Back pointer to the value node to insert ahead of, or behind, if needed. Used to support the 'before' and 'after' modes of the YANG insert operation.
iskey	TRUE if this is a key leaf. FALSE otherwise.
operset	TRUE if there was an nc:operation attribute found in this node; FALSE if the 'editop' is derived from its parent.

Field	Description
pcookie	SIL user pointer cookie. Not used by the server. Reserved for SIL callback code.
icookie	SIL user integer cookie. Not used by the server. Reserved for SIL callback code.

3.2.3   YUMA DATA NODES (VAL_VALUE_T)

The **val_value_t** data structure is used to maintain the internal representation of all NETCONF databases, non-configuration data available with the <get> operation, all RPC operation input and output parameters, and all notification contents.

The following typedef is used to define a value node:

```
/* one value to match one type */
typedef struct val_value_t_ {

    dlq_hdr_t       qhdr;


    /* common fields */
    struct obj_template_t_ *obj;        /* bptr to object def */

    typ_def_t *typdef;                 /* bptr to typdef if leaf */

    const xmlChar   *name;                 /* back pointer to elname */

    xmlChar         *dname;             /* AND malloced name if needed */

    struct val_value_t_ *parent;        /* back-ptr to parent if any */

    xmlns_id_t      nsid;              /* namespace ID for this node */

    ncx_btype_t     btyp;                 /* base type of this value */


    uint32          flags;                 /* internal status flags */

    ncx_data_class_t dataclass;            /* config or state data */


    /* YANG does not support user-defined meta-data but NCX does.
     * The <edit-config>, <get> and <get-config> operations
     * use attributes in the RPC parameters, the metaQ is still used
     *
     * The ncx:metadata extension allows optional attributes
     * to be added to object nodes for anyxml, leaf, leaf-list,
     * list, and container nodes.  The config property will
     * be inherited from the object that contains the metadata
     *
     * This is used mostly for RPC input parameters
     * and is strongly discouraged.  Full edit-config
     * support is not provided for metdata
     */
```

```
    dlq_hdr_t        metaQ;                       /* Q of val_value_t */


    /* value editing variables */
    val_editvars_t  *editvars;              /* edit-in-progress vars */
    status_t         res;                      /* validationt result */


    /* Used by Agent only:
     * if this field is non-NULL, then the entire value node
     * is actually a placeholder for a dynamic read-only object
     * and all read access is done via this callback function;
     * the real data type is getcb_fn_t *
     */
    void *getcb;


    /* if this field is non-NULL, then a malloced value struct
     * representing the real value retrieved by
     * val_get_virtual_value, is cached here for XPath filtering
     * TBD: add timestamp to reuse cached entries for some time
     * period
     */
    struct val_value_t_ *virtualval;


    /* these fields are used for NCX_BT_LIST */
    struct val_index_t_ *index;    /* back-ptr/flag in use as index */
    dlq_hdr_t        indexQ;      /* Q of val_index_t or ncx_filptr_t */


    /* this field is used for NCX_BT_CHOICE
     * If set, the object path for this node is really:
     *    $this --> casobj --> casobj.parent --> $this.parent
     * the OBJ_TYP_CASE and OBJ_TYP_CHOICE nodes are skipped
     * inside an XML instance document
     */
    struct obj_template_t_   *casobj;


    /* these fields are for NCX_BT_LEAFREF
     * NCX_BT_INSTANCE_ID, or tagged ncx:xpath
     * value stored in v union as a string
     */
    struct xpath_pcb_t_            *xpathpcb;


    /* union of all the NCX-specific sub-types
     * note that the following invisible constructs should
```

```
       * never show up in this struct:
       *      NCX_BT_CHOICE
       *      NCX_BT_CASE
       *      NCX_BT_UNION
       */
     union v_ {
     /* complex types have a Q of val_value_t representing
      * the child nodes with values
      *    NCX_BT_CONTAINER
      *    NCX_BT_LIST
      */
        dlq_hdr_t    childQ;


        /* Numeric data types:
      *    NCX_BT_INT8, NCX_BT_INT16,
      *    NCX_BT_INT32, NCX_BT_INT64
      *    NCX_BT_UINT8, NCX_BT_UINT16
      *    NCX_BT_UINT32, NCX_BT_UINT64
      *    NCX_BT_DECIMAL64, NCX_BT_FLOAT64
      */
     ncx_num_t    num;


     /* String data types:
      *    NCX_BT_STRING
      *    NCX_BT_INSTANCE_ID
      */
     ncx_str_t   str;


     val_idref_t idref;


     ncx_binary_t binary;                /* NCX_BT_BINARY */
     ncx_list_t list;       /* NCX_BT_BITS, NCX_BT_SLIST */
     boolean    boo;     /* NCX_BT_EMPTY, NCX_BT_BOOLEAN */
     ncx_enum_t enu;        /* NCX_BT_UNION, NCX_BT_ENUM */
     xmlChar    *fname;                  /* NCX_BT_EXTERN */
     xmlChar    *intbuff;                /* NCX_BT_INTERN */
     } v;
} val_value_t;
```

The following table highlights the fields in this data structure:

val_value_t Fields

Field	Description
qhdr	Internal queue header to allow a value node to be stored in a queue. A complex node maintains a child queue of val_value_t nodes.
obj	Back pointer to the object template for this data node
typdef	Back pointer to the typedef structure if this is a leaf or leaf-list node.
name	Back pointer to the name string for this node
dname	Malloced name string if the client or server changed the name of this node, so the object node name is not being used. This is used for anyxml processing (and other things) to allow generic objects (container, string, empty, etc.) to be used to represent the contents of an 'anyxml' node.
parent	Back pointer to the parent of this node, if any
nsid	Namespace ID for this node. This may not be the same as the object node namespace ID, e.g., anyxml child node contents will override the generic object namespace.
btyp	The ncx_btype_t base type enumeration for this node. This is the final resolved value, in the event the object type is not a final resolved base type.
flags	Internal flags field. Do not access directly.
dataclass	Internal config or non-config enumeration
metaQ	Queue of val_value_t structures that represent any meta-variables (XML attributes) found for this data node. For example, the NETCONF filter 'type' and 'select' attributes are defined for the <filter> element in yuma-netconf.yang.
editvars	Pointer to the malloced edit variables structure for this data node. This node will be freed (and NULL value) when the edit variables are not in use.
res	Internal validation result status for this node during editing or parsing.
getcb	Internal server callback function pointer. Used only if this is a 'virtual' node, and the actual value node contents are generated by a SIL callback function instead of being stored in the node itself.
virtualval	The temporary cached virtual node value, if the getcb pointer is non-NULL.
indexQ	Queue of internal data structures used during parsing and filtering streamed output.
casobj	Back pointer to the OBJ_TYP_CASE object node for this data node, if this node is a top-level child of a YANG case statement.
xpathpcb	XPath parser control block, used if this value contains some sort of XPath string or instance-identifier. For example, the XML namespace ID mappings are stored,

Field	Description
	so the XML prefix map generated for the <rpc-reply> will contain and reuse the proper namespace attributes, as needed.
v	Union of different internal fields, depending on the 'btyp' field value.
v.childQ	Queue of val_value_t child nodes, if this is a complex node.
v.num	ncx_num_t for all numeric data types
v.str	Malloced string value for the string data type
v.idref	Internal data structure for the YANG identityref data type
v.binary	Internal data structure for the YANG binary data type
v.list	Internal data structure for YANG bits and NCX xsdlist data types
v.boo	YANG boolean data type
v.enu	Internal data structure for YANG enumeration data type
v.fname	File name for NCX 'external' data type
v.intbuff	Malloced buffer for 'internal' data type

3.2.4   VAL_VALUE_T ACCESS MACROS

There are a set of macros defined to access the fields within a val_value_t structure.

These should be used instead of accessing the fields directly. There are also functions defined as well. These macros are provided in addition the the access functions for quick access to the actual node value. These macros must only be used when the base type ('btyp') field has been properly set and known by the SIL code. Some auto-generated SIL code uses these macros.

The following table summarized the val_value_t macros that are defined in **ncx/val.h**:

Macro	Description
VAL_BOOL(V)	Access value for NCX_BT_BOOLEAN
VAL_EMPTY(V)	Access value for NCX_BT_EMPTY
VAL_DOUBLE(V)	Access value for NCX_BT_FLOAT64
VAL_STRING(V)	Access value for NCX_BT_STRING
VAL_BINARY(V)	Access value for NCX_BT_BINARY
VAL_ENU(V)	Access entire **ncx_enum_t** structure for NCX_BT_ENUM
VAL_ENUM(V)	Access enumeration integer value for NCX_BT_ENUM
VAL_ENUM_NAME(V)	Access enumeration name string for NCX_BT_ENUM
VAL_FLAG(V)	Deprecated: use VAL_BOOL instead

Macro	Description
VAL_LONG(V)	Access NCX_BT_INT64 value
VAL_INT(V)	Access NCX_BT_INT32 value
VAL_INT8(V)	Access NCX_BT_INT8 value
VAL_INT16(V)	Access NCX_BT_INT16 value
VAL_STR(V)	Deprecated: use VAL_STRING instead
VAL_INSTANCE_ID(V)	Access NCX_BT_INSTANCE_ID value
VAL_IDREF(V)	Access entire **val_idref_t** structure for NCX_BT_IDREF
VAL_IDREF_NSID(V)	Access the identityref namespace ID for NCX_BT_IDREF
VAL_IDREF_NAME(V)	Access the identityref name string for NCX_BT_IDREF
VAL_UINT(V)	Access the NCX_BT_UINT32 value
VAL_UINT8(V)	Access the NCX_BT_UINT8 value
VAL_UINT16(V)	Access the NCX_BT_UINT16 value
VAL_ULONG(V)	Access the NCX_BT_UINT64 value
VAL_DEC64(V)	Access the **ncx_dec64**structure for NCX_BT_DEC64
VAL_LIST(V)	Access the **ncx_list_t** structure for NCX_BT_LIST
VAL_BITS	Access the **ncx_list_t** structure for NCX_BT_BITS. (Same as VAL_LIST)

3.2.5   VAL_VALUE_T ACCESS FUNCTIONS

The file **ncx/val.h** contains many API functions so that object properties do not have to be accessed directly. In addition, the file ncx/val_util.h contains more (high-level) utility functions. The following table highlights the most commonly used functions. Refer to the H files for a complete definition of each API function.

val_value_t Access Functions

Function	Description
val_new_value	Malloc a new value node with type NCX_BT_NONE.
val_init_complex	Initialize a malloced value node as one of the complex data types.
val_init_virtual	Initialize a malloced value node as a virtual node (provide a 'get' callback function).
val_init_from_template	Initialize a malloced value node using an object template. This is the most common form of the init function used by SIL callback functions.
vaL-free_value	Clean and free a malloced value node.

Function	Description
val_set_name	Set or replace the value node name.
val_set_qname	Set or replace the value node namespace ID and name.
val_string_ok	Check if the string value is valid for the value node object type.
val_string_ok_errinfo	Check if the string value is valid for the value node object type, and provide the error information to use if it is not OK.
val_list_ok	Check if the list value is valid for the value node object type.
val_list_ok_errinfo	Check if the list value is valid for the value node object type, and provide the error information to use if it is not OK.
val_enum_ok	Check if the enumeration value is valid for the value node object type.
val_enum_ok_errinfo	Check if the enumeration value is valid for the value node object type, and provide the error information to use if it is not OK.
val_bit_ok	Check if the bits value is valid for the value node object type.
val_idref_ok	Check if the identityref value is valid for the value node object type.
val_parse_idref	Convert a string to an internal QName string into its various parts and find the identity struct that is being referenced (if available).
val_simval_ok	Check if the smple value is valid for the value node object type.
val_simval_ok_errinfo	Check if the simple value is valid for the value node object type, and provide the error information to use if it is not OK.
val_get_first_meta	Get the first meta-variable (XML attribute value) for a value node.
val_get_next_meta	Get the next meta-variable (XML attribute value) for a value node.
val_find_meta	Find the specified meta-variable in a value node.
val_dump_value	Debug function to print the contents of any value node.
val_dump_value_ex	Debug function to print the contents of any value node, with extended parameters to control the output.
val_dump_value_max	Debug function to print the contents of any value node, with full control of the output parameters.
val_set_string	Set a malloced value node as a generic string value. Used instead of val_init_from_template.

Function	Description
val_set_string2	Set a malloced value node as a specified string type. Used instead of val_init_from_template.
val_set_simval	Set a malloced value node as a specified simple type. Used instead of val_init_from_template.
val_set_simval_str	Set a malloced value node as a specified simple type. Used instead of val_init_from_template. Use a counted string value instead of a zero-terminated string value.
val_make_string	Create a complete malloced generic string value node.
val_clone	Clone a value node
val_clone_test	Clone a value node with a 'test' callback function to prune certain descendant nodes during the clone procedure.
val_clone_config_data	Clone a value node but skip all the non-configuration descendant nodes.
val_add_child	Add a child value node to a parent value node.
val_insert_child	Insert a child value node into a specific spot into a parent value node.
val_remove_child	Remove a child value node from its parent.
val_swap_child	Replace a child node within its parent with a different value node.
val_first_child_match	Match a child node name; Used for partial command completion in yangcli.
val_next_child_match	Match the next child node name; Used for partial command completion in yangcli.
val_get_first_child	Get the first child value node.
val_get_next_child	Get the next child value node.
val_find_child	Find a specific child value node.
val_find_next_child	Find the next occurrence of a specified child node.
val_match_child	Match a potential partial node name against the child node names, and return the first match found, if any.
val_child_cnt	Get the number of child nodes within a parent node.
val_liststr_count	Get the number of strings within an NCX_BT_LIST value node.
val_index_match	Check if 2 value list nodes have the same set of key leaf values.
val_compare	Compare 2 value nodes
val_compare_ex	Compare 2 value nodes with extra parameters.
val_compare_to_string	Compare a value node to a string value instead of another value node.

Function	Description
val_sprintf_simval_nc	Output the value node as a string into the specified buffer.
val_make_sprintf_string	Malloc a buffer and fill it with a zero-terminated string representation of the value node.
val_resolve_scoped_name	Find a descendant node within a value node, from a relative path expression.
val_has_content	Return TRUE if the value node has any content; FALSE if an empty XML element could represent its value.
val_has_index	Return TRUE if the value node is a list with a key statement.
val_get_first_index	Get the first index node for the specified list value node.
val_get_next_index	Get the next index node for the specified list value node.
val_set_extern	Set a malloced value node as an NCX_BT_EXTERN internal data type.
val_set_intern	Set a malloced value node as an NCX_BT_INTERN internal data type.
val_fit_oneline	Return TRUE if the value node should fit on 1 display line; Sometimes a guess is made instead of determining the exact value. XML namespace declarations generated during XML output can cause this function value to sometimes be wrong.
val_create_allowed	Return TRUE if the NETCONF create operation is allowed for the specified value node.
val_delete_allowed	Return TRUE if the NETCONF delete operation is allowed for the specified value node.
val_is_config_data	Return TRUE if the value node represents configuration data.
val_get_virtual_value	Get the value for a virtual node from its 'get' callback function.
val_is_default	Return TRUE if the value node is set to its YANG default value.
val_is_real	Check if a value node is a real node or one of the abstract node types.
val_get_parent_nsid	Get the namespace ID for the parent value node of a specified child node.
val_instance_count	Get the number of occurrences of the specified child value node within a parent value node.
val_need_quotes	Return TRUE if the printed string representation of a value node needs quotes (because it contains some whitespace or special characters).

Function	Description
val_get_dirty_flag	Check if a value node has been altered by an RPC operation, but this edit has not been finalized yet.
val_get_nest_level	Get the current numeric nest level of the specified value node.
val_get_mod_name	Get the module name for the specified value node.
val_get_mod_prefix	Get the module prefix string for the specified value node.
val_get_nsid	Get the namespace ID for the specified value node.
val_change_nsid	Change the namespace ID for the specified value node and all of its descendents.
val_set_pcookie	Set the SIL pointer cookie in the value node editvars structure.
val_set_icookie	Set the SIL integer cookie in the value node editvars structure.
val_get_pcookie	Get the SIL pointer cookie in the value node editvars structure.
val_get_icookie	Get the SIL integer cookie in the value node editvars structure.
val_get_typdef	Get the typedef structure for a leaf or leaf-list value node.
val_move_children	Move all the child nodes of one complex value node to another complex value node.
val_set_canonical_order	Re-order the descendant nodes of a value node so they are in YANG order. Does not change the relative order of system-ordered lists and leaf-lists.
val_gen_index_chain	Generate the internal key leaf lookup chain for a list value node..
val_add_defaults	Generate the leafs that have default values.
val_instance_check	Check a value node against its template to see if the correct number of descendant nodes are present.
val_get_choice_first_set	Get the first real node that is present for a conceptual choice statement.
val_get_choice_next_set	Get the next real node that is present for a conceptual choice statement.
val_choice_is_set	Return TRUE if some real data node is present for a conceptual choice statement.
val_new_child_val	Create a child node during an edit operation. Used by the server. SIL code does not need to maintain the value tree.
val_gen_instance_id	Malloc and generate the YANG instance-

Function	Description
	identifier string for the value node.
val_check_obj_when	Check if an object node has any 'when' statements, and if so, evaluate the XPath condition(s) against the value tree to determine if the object should be considered present or not.
val_check_child_conditional	Check if a child object node has any FALSE 'if-feature' or 'when' statements.
val_is_mandatory	Check if the child object node is currently mandatory or optional.
val_get_xpathpcb	Access the XPath parser control block for this value node, if any.
val_make_simval_obj	Malloc and fill in a value node from an object template and a value string.
val_set_simval_obj	Fill in a value node from an object template and a value string.

3.2.6 SIL UTILITY FUNCTIONS

There are some high-level SIL callback utilities in **agt/agt_util.h**. These functions access the lower-level functions in libncx to provide simpler functions for common SIL tasks.

The following table highlights the functions available in this module:

agt/agt_util Functions

Function	Description
agt_get_cfg_from_parm	For value nodes that represent a NETCONF configuration database name (e.g., empty element named 'running'). The configuration control block for the referenced database is retrieved.
agt_get_inline_cfg_from_parm	For value nodes that represent inline NETCONF configuration data. The value node for the inline config node is retrieved.
agt_get_parmval	Get the specified parameter name within the RPC input section, from an RPC message control block.
agt_record_error	Generate a complete RPC error record to be used when the <rpc-reply> is sent.
agt_record_error_errinfo	Generate a complete RPC error record to be used when the <rpc-reply> is sent, using the YANG specified error information, not the default error information.
agt_record_attr_error	Generate a complete RPC error record to be used when the <rpc-reply> is sent for an

Function	Description
	XML attribute error.
agt_record_insert_error	Generate a complete RPC error record to be used when the <rpc-reply> is sent for a YANG insert operation error.
agt_record_unique_error	Generate a complete RPC error record to be used when the <rpc-reply> is sent for a YANG unique statement contraint error.
agt_check_default	val_nodetest_fn_t Node Test Callback function to filter out default data from streamed replies, according to the server's definition of a default node.
agt_check_save	val_nodetest_fn_t Node Test Callback function to filter out data nodes that should not be saved to NV-storage.
agt_enable_feature	Enable the specified YANG feature
agt_disable_feature	Disable the specified YANG feature
agt_make_leaf	Create a child value node.
agt_make_virtual_leaf	Create a virtual value child node. Most device monitoring leafs use this function because the value is retrieved with a device-specific API, not stored in the value tree.
agt_init_cache	Initialize a cached pointer to a node in a data tree.
agt_check_cache	Check if a cached pointer to a node in a data tree needs to be updated or set to NULL.

4   SIL External Interface

Each SIL has 2 initialization functions and 1 cleanup function that must be present.

- The first initialization callback function is used to set up the configuration related objects.
- The second initialization callback is used to setup up non-configuration objects, after the running configuration has been loaded from the startup file.
- The cleanup callback is used to remove all SIL data structures and unregister all callback functions.

These are the only SIL functions that the server will invoke directly. They are generated by **yangdump** with the **--format**=c parameter, and usually do not require any editing by the developer.

Most of the work done by SIL code is through callback functions for specific RPC operations and database objects. These callback functions are registered during the initialization functions.

4.1   Stage 1 Initialization

The stage 1 initialization function is the first function called in the library by the server.

If the **netconfd** configuration parameters include a 'load' command for the module, then this function will be called during server initialization. It can also be called if the <load> operation is invoked during server operation.

This function MUST NOT attempt to access any database. There will not be any configuration databases if this function is called during server initialization. Use the 'init2' function to adjust the running configuration.

This callback function is expected to perform the following functions:

- initialize any module static data

- make sure the requested module name and optional revision date parameters are correct

- load the requested module name and revision with **ncxmod_load_module**

- setup top-level object cache pointers (if needed)

- register any RPC method callbacks with **agt_rpc_register_method**

- register any database object callbacks with **agt_cb_register_callback**

- perform any device-specific and/or module-specific initialization

Name Format:

> y_<modname>_init

Input:

- modname == string containing module name to load

- revision == string containing revision date to use
        == NULL if the operator did not specify a revision.

Returns:

- operation status (0 if success)

Example function generated by **yangdump**:

```
/********************************************************************
* FUNCTION y_toaster_init
*
* initialize the toaster server instrumentation library
*
* INPUTS:
*    modname == requested module name
*    revision == requested version (NULL for any)
*
* RETURNS:
*     error status
********************************************************************/
status_t
    y_toaster_init (
        const xmlChar *modname,
```

```
            const xmlChar *revision)
{
    agt_profile_t *agt_profile;
    status_t res;


    y_toaster_init_static_vars();


    /* change if custom handling done */
    if (xml_strcmp(modname, y_toaster_M_toaster)) {
        return ERR_NCX_UNKNOWN_MODULE;
    }


    if (revision && xml_strcmp(revision, y_toaster_R_toaster)) {
        return ERR_NCX_WRONG_VERSION;
    }


    agt_profile = agt_get_profile();


    res = ncxmod_load_module(
        y_toaster_M_toaster,
        y_toaster_R_toaster,
        &agt_profile->agt_savedevQ,
        &toaster_mod);
    if (res != NO_ERR) {
        return res;
    }


    toaster_obj = ncx_find_object(
        toaster_mod,
        y_toaster_N_toaster);
    if (toaster_mod == NULL) {
        return SET_ERROR(ERR_NCX_DEF_NOT_FOUND);
    }


    make_toast_obj = ncx_find_object(
        toaster_mod,
        y_toaster_N_make_toast);
    if (toaster_mod == NULL) {
        return SET_ERROR(ERR_NCX_DEF_NOT_FOUND);
    }


    cancel_toast_obj = ncx_find_object(
```

```
            toaster_mod,
            y_toaster_N_cancel_toast);
    if (toaster_mod == NULL) {
            return SET_ERROR(ERR_NCX_DEF_NOT_FOUND);
    }


    toastDone_obj = ncx_find_object(
            toaster_mod,
            y_toaster_N_toastDone);
    if (toaster_mod == NULL) {
            return SET_ERROR(ERR_NCX_DEF_NOT_FOUND);
    }


    res = agt_rpc_register_method(
            y_toaster_M_toaster,
            y_toaster_N_make_toast,
            AGT_RPC_PH_VALIDATE,
            y_toaster_make_toast_validate);
    if (res != NO_ERR) {
            return res;
    }


    res = agt_rpc_register_method(
            y_toaster_M_toaster,
            y_toaster_N_make_toast,
            AGT_RPC_PH_INVOKE,
            y_toaster_make_toast_invoke);
    if (res != NO_ERR) {
            return res;
    }


    res = agt_rpc_register_method(
            y_toaster_M_toaster,
            y_toaster_N_cancel_toast,
            AGT_RPC_PH_VALIDATE,
            y_toaster_cancel_toast_validate);
    if (res != NO_ERR) {
            return res;
    }


    res = agt_rpc_register_method(
            y_toaster_M_toaster,
```

```
            y_toaster_N_cancel_toast,

            AGT_RPC_PH_INVOKE,

            y_toaster_cancel_toast_invoke);
        if (res != NO_ERR) {

            return res;

        }


        res = agt_cb_register_callback(
            y_toaster_M_toaster,

            (const xmlChar *)"/toaster",

            (const xmlChar *)"2009-11-20",

            y_toaster_toaster_edit);
        if (res != NO_ERR) {

            return res;

        }


        /* put your module initialization code here */


        return res;
    } /* y_toaster_init */
```

4.2   Stage 2 Initialization

The stage 2 initialization function is the second function called in the library by the server:

- It will only be called if the stage 1 initialization is called first, and it returns 0 (NO_ERR status).
- This function is used to initialize any needed data structures in the running configuration, such as factory default configuration, read-only counters and status objects.
- It is called after the startup configuration has been loaded into the server.
- If the <load> operation is used during server operation, then this function will be called immediately after the state 1 initialization function.

Note that configuration data structures that are loaded during server initialization **(load_running_config**) will be handled by the database callback functions registered during phase 1 initialization.

Any server-created configuration nodes should be created during phase 2 initialization (this function), after examining the explicitly-provided configuration data. For example, the top-level /nacm container will be created (by agt_acm.c) if it is not provided in the startup configuration.

This callback function is expected to perform the following functions:

- load non-configuration data structures into the server (if needed)
- initialize top-level data node cache pointers (if needed)
- load factory-default configuration data structures into the server (if needed)

- optionally save a cached pointer to a data tree node (such as the root node for the module). The **agt_create_cache** function in agt/agt_util.h is used to initialize such a module-static variable.

Name Format:

> y_<modname>_init2

Returns:

- operation status (0 if success)

Example function generated by **yangdump**:

```
/*****************************************************************
 * FUNCTION y_toaster_init2
 *
 * SIL init phase 2: non-config data structures
 * Called after running config is loaded
 *
 * RETURNS:
 *     error status
 *****************************************************************/
status_t
    y_toaster_init2 (void)
{
    status_t res;

    res = NO_ERR;

    toaster_val = agt_init_cache(
        y_toaster_M_toaster,
        y_toaster_N_toaster,
        &res);
    if (res != NO_ERR) {
        return res;
    }

    /* put your init2 code here */

    return res;
} /* y_toaster_init2 */
```

4.3   Cleanup

The cleanup function is called during server shutdown. It is only called if the stage 1 initialization function is called. It will be called right away if either the stage 1 or stage 2 initialization functions return a non-zero error status.

This callback function is expected to perform the following functions:

- cleanup any module static data

- free any top-level object cache pointers (if needed)

- unregister any RPC method callbacks with **agt_rpc_unregister_method**

- unregister any database object callbacks with **agt_cb_unregister_callbacks**

- perform any device-specific and/or module-specific cleanup

Name Format:

   y_<modname>_cleanup

Example function generated by **yangdump**:

```
/*****************************************************************
 * FUNCTION y_toaster_cleanup
 *    cleanup the server instrumentation library
 *
 *****************************************************************/
void
    y_toaster_cleanup (void)
{
    agt_rpc_unregister_method(
        y_toaster_M_toaster,
        y_toaster_N_make_toast);


    agt_rpc_unregister_method(
        y_toaster_M_toaster,
        y_toaster_N_cancel_toast);


    agt_cb_unregister_callbacks(
        y_toaster_M_toaster,
        (const xmlChar *)"/toaster");


    /* put your cleanup code here */


} /* y_toaster_cleanup */
```

5 SIL Callback Interface

This section briefly describes the SIL code that a developer will need to create to handle the data-model specific details. SIL functions access internal server data structures, either directly or through utility functions. Database mechanics and XML processing are done by the server engine, not the SIL code. A more complete reference can be found in section 5.

When a <rpc> request is received, the NETCONF server engine will perform the following tasks before calling any SIL:

- parse the RPC operation element, and find its associated YANG rpc template

- if found, check if the session is allowed to invoke this RPC operation

- if the RPC is allowed, parse the rest of the XML message, using the **rpc_template_t** for the RPC operation to determine if the basic structure is valid.

- if the basic structure is valid, construct an **rpc_msg_t** data structure for the incoming message.

- check all YANG machine-readable constraints, such as must, when, if-feature, min-elements, etc.

- if the incoming message is completely 'YANG valid', then the server will check for an RPC validate function, and call it if found. This SIL code is only needed if there are additional system constraints to check. For example:

    ◦ need to check if a configuration name such as <candidate/> is supported

    ◦ need to check if a configuration database is locked by another session

    ◦ need to check description statement constraints not covered by machine-readable constraints

    ◦ need to check if a specific capability or feature is enabled

- If the validate function returns a NO_ERR status value, then the server will call the SIL invoke callback, if it is present. This SIL code should always be present, otherwise the RPC operation will have no real affect on the system.

- At this point, an <rpc-reply> is generated, based on the data in the rpc_msg_t.

    ◦ Errors are recorded in a queue when they are detected.

    ◦ The server will handle the error reply generation for all errors it detects.

    ◦ For SIL detected errors, the **agt_record_error** function in agt/agt_util.h is usually used to save the error details.

    ◦ Reply data can be generated by the SIL invoke callback function and stored in the rpc_msg_t structure.

    ◦ Replay data can be streamed by the SIL code via reply callback functions. For example, the <get> and <get-config> operations use callback functions to deal with filters, and stream the reply by walking the target data tree.

- After the <rpc-reply> is sent, the server will check for an RPC post reply callback function. This is only needed if the SIL code allocated some per-message data structures. For example, the **rpc_msg_t** contains 2 SIL controlled pointers (**rpc_user1** and **rpc_user2**). The post reply callback is used by the SIL code to free these pointers, if needed.

The database edit SIL callbacks are only used for database operations that alter the database. The validate and invoke callback functions for these operations will in turn invoke the data-model specific SIL callback functions, depending on the success or failure of the edit request.

5.1   RPC Operation Interface

All RPC operations are data-driven within the server, using the YANG rpc statement for the operation and SIL callback functions.

Any new protocol operation can be added by defining a new YANG rpc statement in a module, and providing the proper SIL code.

5.1.1   RPC CALLBACK INITIALIZATION

The **agt_rpc_register_method** function in agt/agt_rpc.h is used to provide a callback function for a specific callback phase. The same function can be used for multiple phases if desired.

```
/* Template for RPC server callbacks
 * The same template is used for all RPC callback phases
 */
typedef status_t
    (*agt_rpc_method_t) (ses_cb_t *scb,
                    rpc_msg_t *msg,
                    xml_node_t *methnode);


extern status_t
    agt_rpc_register_method (const xmlChar *module,
                             const xmlChar *method_name,
                             agt_rpc_phase_t  phase,
                             agt_rpc_method_t method);
```

agt_rpc_register_method

Parameter	Description
module	The name of the module that contains the rpc statement
method_name	The identifier for the rpc statement
phase	AGT_PH_VALIDATE(0): validate phase AGT_PH_INVOKE(1): invoke phase AGT_PH_POST_REPLY(2): post-reply phase
method	The address of the callback function to register

5.1.2   RPC MESSAGE HEADER

The NETCONF server will parse the incoming XML message and construct an RPC message header, which is used to maintain state and any other message-specific data during the processing of an incoming <rpc> request.

The **rpc_msg_t** data structure in ncx/rpc.h is used for this purpose. The following table summarizes the fields:

rpc_msg_t

Field	Type	User Mode	Description
qhdr	dlq_hdr_t	none	Queue header to store RPC messages in a queue (within the session header)
mhdr	xml_msg_hdr_t	none	XML message prefix map and other data used to parse the request and generate the reply.
rpc_in_attrs	xml_attrs_t *	read write	Queue of xml_attr_t representing any XML attributes that were present in the <rpc> element. A callback function may add xml_attr_t structs to this queue to send in the reply.
rpc_method	obj_template_t *	read	Back-pointer to the object template for this RPC operation.
rpc_agt_state	int	read	Enum value (0, 1, 2) for the current RPC callback phase.
rpc_err_option	op_errop_t	read	Enum value for the <error-option> parameter. This is only set if the <edit-config> operation is in progress.
rpc_top_editop	op_editop_t	read	Enum value for the <default-operation> parameter. This is only set if the <edit-config> operation is in progress.
rpc_input	val_value_t *	read	Value tree representing the container of 'input' parameters for this RPC operation.
rpc_user1	void *	read write	Void pointer that can be used by the SIL functions to store their own message-specific data.
rpc_user2	void *	read write	Void pointer that can be used by the SIL functions to store their own message-specific data.
rpc_returncode	uint32	none	Internal return code used to control nested callbacks.
rpc_data_type	rpc_data_t	write	For RPC operations that return data, this enumeration is set to indicate which type of data is desired. **RPC_DATA_STD**: A <data> container will be used to encapsulate any returned data, within the <rpc-reply> element. **RPC_DATA_YANG**: The <rpc-reply> element will be the only container encapsulated any returned data.
rpc_datacb	void *	write	For operations that return streamed data, this pointer is set to the desired callback function to use for generated the data portion of the <rpc-reply> XML response. The template for this callback is **agt_rpc_data_cb_t,** found in agt_rpc.h
rpc_dataQ	dlq_hdr_t	write	For operations that return stored data, this queue of val_value_t structures can be used to provide the response data.

Field	Type	User Mode	Description
			Each val_value_t structure will be encoded as one of the corresponding RPC output parameters.
rpc_filter	op_filter_t	none	Internal structure for optimizing subtree and XPath retrieval operations.
rpc_need_undo	boolean	none	Internal flag to indicate if rollback-on-error is in effect dusing an <edit-config> operation.
rpc_undoQ	dlq_hdr_t	none	Queue of **rpc_undo_rec_t** structures, used to undo edits if rollback-on-error is in affect during an <edit-config> operation.
rpc_auditQ	dlq_hdr_t	none	Queue of rpc_audit_rec_t structures used internally to generate database alteration notifications and audit log entries.

The following C code represents the **rpc_msg_t** data structure:

```
/* NETCONF Server and Client RPC Request/Reply Message Header */
typedef struct rpc_msg_t_ {
    dlq_hdr_t          qhdr;


    /* generic XML message header */
    xml_msg_hdr_t      mhdr;


    /* incoming: top-level rpc element data */
    xml_attrs_t     *rpc_in_attrs;      /* borrowed from <rpc> elem */


    /* incoming:
     * 2nd-level method name element data, used in agt_output_filter
     * to check get or get-config
     */
    struct obj_template_t_ *rpc_method;


    /* incoming: SERVER RPC processing state */
    int                rpc_agt_state;        /* agt_rpc_phase_t */
    op_errop_t         rpc_err_option;
    op_editop_t        rpc_top_editop;
    val_value_t     *rpc_input;


    /* incoming:
     * hooks for method routines to save context or whatever
```

```
     */
    void           *rpc_user1;
    void           *rpc_user2;
    uint32          rpc_returncode;    /* for nested callbacks */


    /* incoming: get method reply handling builtin
     * If the rpc_datacb is non-NULL then it will be used as a
     * callback to generate the rpc-reply inline, instead of
     * buffering the output.
     * The rpc_data and rpc_filter parameters are optionally used
     * by the rpc_datacb function to generate a reply.
     */
    rpc_data_t      rpc_data_type;          /* type of data reply */
    void           *rpc_datacb;             /* agt_rpc_data_cb_t */
    dlq_hdr_t       rpc_dataQ;       /* data reply: Q of val_value_t */
    op_filter_t     rpc_filter;         /* backptrs for get* methods */


    /* incoming: rollback or commit phase support builtin
     * As an edit-config (or other RPC) is processed, a
     * queue of 'undo records' is collected.
     * If the apply phase succeeds then it is discarded,
     * otherwise if rollback-on-error is requested,
     * it is used to undo each change that did succeed (if any)
     * before an error occured in the apply phase.
     */
    boolean         rpc_need_undo;   /* set by edit_config_validate */
    dlq_hdr_t       rpc_undoQ;       /* Q of rpc_undo_rec_t */
    dlq_hdr_t       rpc_auditQ;       /* Q of rpc_audit_rec_t */

} rpc_msg_t;
```

5.1.3 SIL Support Functions For RPC Operations

The file agt/agt_rpc.c contains some functions that are used by SIL callback functions.

The following table highlights the functions that may be useful to SIL developers:

agt/agt_rpc.c Functions

Function	Description
agt_rpc_register_method	Register a SIL RPC operation callback function for 1 callback phase.

Function	Description
agt_rpc_support_method	Tell the server that an RPC operation is supported by the system..
agt_rpc_unsupport_method	Tell the server that an RPC operation is not supported by the system..
agt_rpc_unregister_method	Remove all the SIL RPC operation callback functions for 1 RPC operation.

5.1.4   RPC VALIDATE CALLBACK FUNCTION
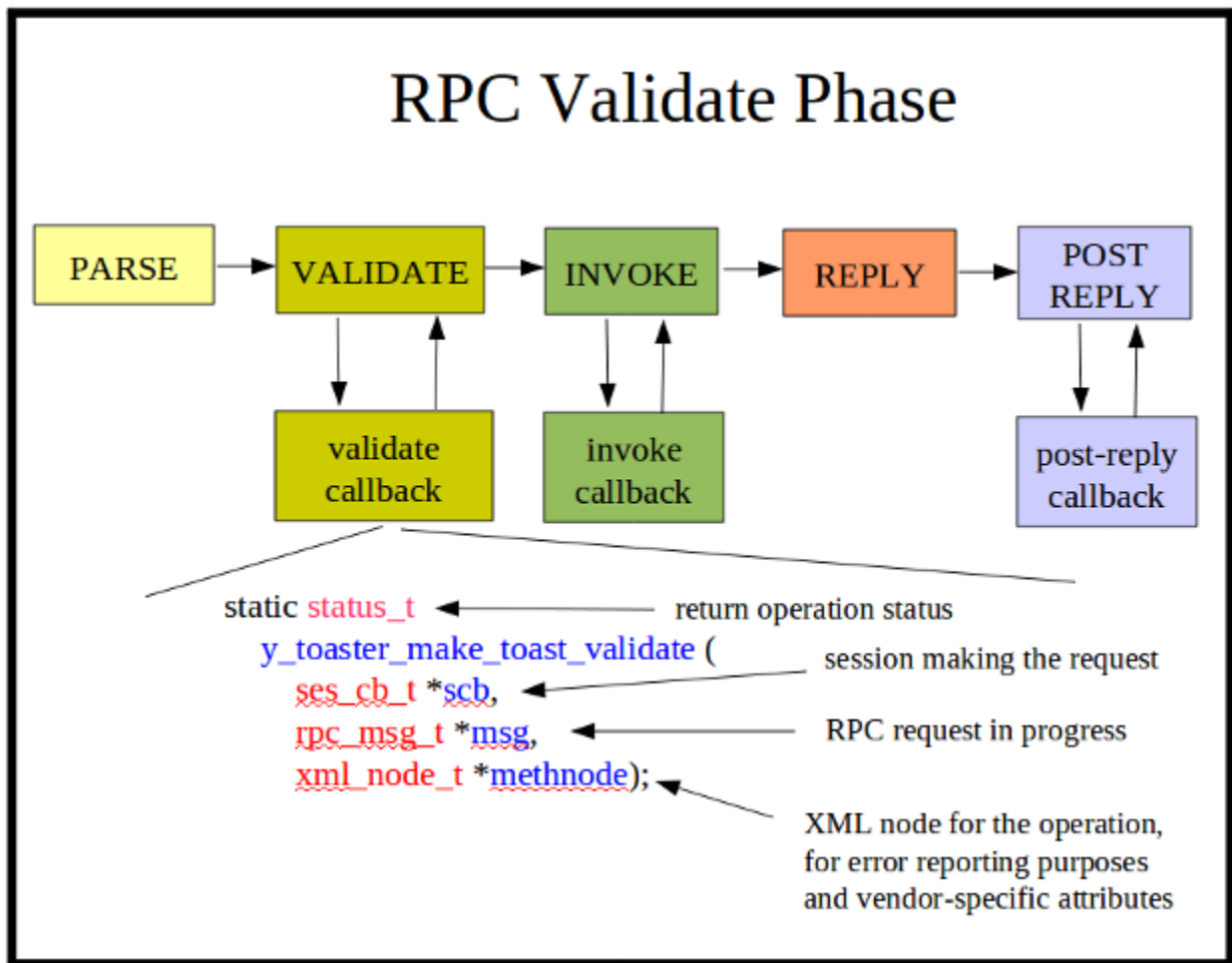
The RPC validate callback function is optional to use. Its purpose is to validate any aspects of an RPC operation, beyond the constraints checked by the server engine. Only 1 function can register for each YANG rpc statement. The standard NETCONF operations are reserved by the server engine. There is usually zero or one of these callback functions for every 'rpc' statement in the YANG module associated with the SIL code.

It is enabled with the **agt_rpc_register_method** function, within the phase 1 initialization callback function.

The **yangdump** code generator will create this SIL callback function by default. There will C comments in the code to indicate where your additional C code should be added.

The **val_find_child** function is commonly used to find particular parameters within the RPC input section, which is encoded as a **val_value_t** tree.

The **agt_record_error** function is commonly used to record any parameter or other errors. In the **libtoaster** example, there are internal state variables (**toaster_enabled** and **toaster_toasting**), maintained by the SIL code, which are checked in addition to any provided parameters.

Example SIL Function Registration

```
    res = agt_rpc_register_method(

        y_toaster_M_toaster,

        y_toaster_N_make_toast,

        AGT_RPC_PH_VALIDATE,

        y_toaster_make_toast_validate);

    if (res != NO_ERR) {

        return res;

    }
```

Example SIL Function:

```
    /*******************************************************************

    * FUNCTION y_toaster_make_toast_validate

    *

    * RPC validation phase

    * All YANG constraints have passed at this point.

    * Add description-stmt checks in this function.

    *

    * INPUTS:

    *     see agt/agt_rpc.h for details

    *

    * RETURNS:

    *     error status

    *******************************************************************/

    static status_t

        y_toaster_make_toast_validate (

            ses_cb_t *scb,

            rpc_msg_t *msg,

            xml_node_t *methnode)
```

```
{
    status_t res;

    val_value_t *errorval;

    const xmlChar *errorstr;

    val_value_t *toasterDoneness_val;

    val_value_t *toasterToastType_val;

    uint32 toasterDoneness;

    val_idref_t *toasterToastType;


    res = NO_ERR;

    errorval = NULL;

    errorstr = NULL;


    toasterDoneness_val = val_find_child(
        msg->rpc_input,
        y_toaster_M_toaster,
        y_toaster_N_toasterDoneness);
    if (toasterDoneness_val != NULL && toasterDoneness_val->res == NO_ERR) {
        toasterDoneness = VAL_UINT(toasterDoneness_val);
    }


    toasterToastType_val = val_find_child(
        msg->rpc_input,
        y_toaster_M_toaster,
        y_toaster_N_toasterToastType);
    if (toasterToastType_val != NULL && toasterToastType_val->res == NO_ERR) {
        toasterToastType = VAL_IDREF(toasterToastType_val);
    }


    /* added code starts here */
    if (toaster_enabled) {
        /* toaster service enabled, check if in use */
        if (toaster_toasting) {
            res = ERR_NCX_IN_USE;
        } else {
            /* this is where a check on bread inventory would go */


            /* this is where a check on toaster HW ready would go */
        }
    } else {
        /* toaster service disabled */
        res = ERR_NCX_RESOURCE_DENIED;
```

```
        }
        /* added code ends here */


        /* if error: set the res, errorstr, and errorval parms */
        if (res != NO_ERR) {
            agt_record_error(
                    scb,
                    &msg->mhdr,
                    NCX_LAYER_OPERATION,
                    res,
                    methnode,
                    NCX_NT_STRING,
                    errorstr,
                    NCX_NT_VAL,
                    errorval);
        }


    return res;


} /* y_toaster_make_toast_validate */
```

5.1.5   RPC Invoke Callback Function
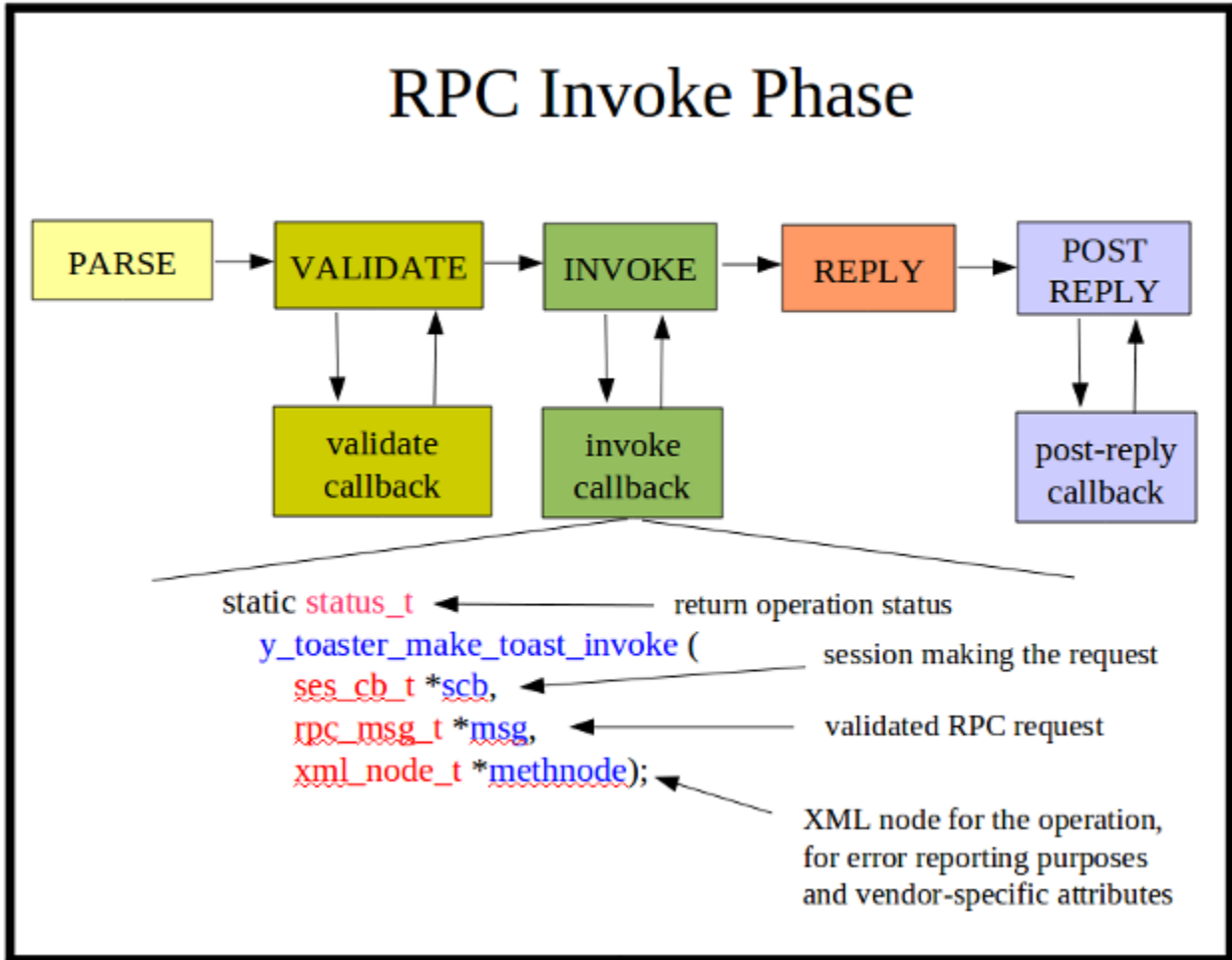
The RPC invoke callback function is used to perform the operation requested by the client session. Only 1 function can register for each YANG rpc statement. The standard NETCONF operations are reserved by the server engine. There is usually one of these callback functions for every 'rpc' statement in the YANG module associated with the SIL code.

The RPC invoke callback function is optional to use, although if no invoke callback is provided, then the operation will have no affect. Normally, this is only the case if the module is be tested by an application developer, using **netconfd** as a server simulator.

It is enabled with the **agt_rpc_register_method** function, within the phase 1 initialization callback function.

The **yangdump** code generator will create this SIL callback function by default. There will C comments in the code to indicate where your additional C code should be added.

The **val_find_child** function is commonly used to retrieve particular parameters within the RPC input section, which is encoded as a **val_value_t** tree. The **rpc_user1** and **rpc_user2** cache pointers in the **rpc_msg_t** structure can also be used to store data in the validation phase, so it can be immediately available in the invoke phase.

The **agt_record_error** function is commonly used to record any internal or platform-specific errors. In the **libtoaster** example, if the request to create a timer callback control block fails, then an error is recorded.

For RPC operations that return either an <ok> or <rpc-error> response, there is nothing more required of the RPC invoke callback function.

For operations which return some data or <rpc-error>, the SIL code must do 1 of 2 additional tasks:

- add a **val_value_t** structure to the **rpc_dataQ** queue in the **rpc_msg_t** for each parameter listed in the YANG rpc 'output' section.

- set the rpc_datacb pointer in the rpc_msg_t structure to the address of your data reply callback function. See the **agt_rpc_data_cb_t** definition in agt/agt_rpc.h for more details.

Example SIL Function Registration

```
    res = agt_rpc_register_method(

        y_toaster_M_toaster,

        y_toaster_N_make_toast,

        AGT_RPC_PH_INVOKE,

        y_toaster_make_toast_invoke);

    if (res != NO_ERR) {

        return res;

    }
```

Example SIL Function:

```
    /********************************************************************

    * FUNCTION y_toaster_make_toast_invoke

    *

    * RPC invocation phase

    * All constraints have passed at this point.

    * Call device instrumentation code in this function.

    *

    * INPUTS:

    *     see agt/agt_rpc.h for details

    *

    * RETURNS:

    *     error status

    ********************************************************************/

    static status_t

        y_toaster_make_toast_invoke (

            ses_cb_t *scb,

            rpc_msg_t *msg,

            xml_node_t *methnode)

    {
```

```
    status_t res;

    val_value_t *toasterDoneness_val;

    val_value_t *toasterToastType_val;

    uint32 toasterDoneness;

    val_idref_t *toasterToastType;


    res = NO_ERR;

    toasterDoneness = 0;


    toasterDoneness_val = val_find_child(

        msg->rpc_input,

        y_toaster_M_toaster,

        y_toaster_N_toasterDoneness);

    if (toasterDoneness_val != NULL && toasterDoneness_val->res == NO_ERR) {

        toasterDoneness = VAL_UINT(toasterDoneness_val);

    }


    toasterToastType_val = val_find_child(

        msg->rpc_input,

        y_toaster_M_toaster,

        y_toaster_N_toasterToastType);

    if (toasterToastType_val != NULL && toasterToastType_val->res == NO_ERR) {

        toasterToastType = VAL_IDREF(toasterToastType_val);

    }


    /* invoke your device instrumentation code here */


    /* make sure the toasterDoneness value is set */

    if (toasterDoneness_val == NULL) {

        toasterDoneness = 5;   /* set the default */

    }



    /* arbitrary formula to convert toaster doneness to the
     * number of seconds the toaster should be on
     */

    toaster_duration = toasterDoneness * 12;


    /* this is where the code would go to adjust the duration
     * based on the bread type
     */
```

```
    if (LOGDEBUG) {

        log_debug("\ntoaster: starting toaster for %u seconds",

                    toaster_duration);

    }


    /* this is where the code would go to start the toaster
     * heater element
     */


    /* start a timer to toast for the specified time interval */
    res = agt_timer_create(toaster_duration,

                            FALSE,

                            toaster_timer_fn,

                            NULL,

                            &toaster_timer_id);

    if (res == NO_ERR) {

        toaster_toasting = TRUE;

    } else {

        agt_record_error(

            scb,

            &msg->mhdr,

            NCX_LAYER_OPERATION,

            res,

            methnode,

            NCX_NT_NONE,

            NULL,

            NCX_NT_NONE,

            NULL);

    }
    /* added code ends here */


    return res;


} /* y_toaster_make_toast_invoke */
```

5.1.6   RPC POST REPLY CALLBACK FUNCTION

RPC Post Reply Phase
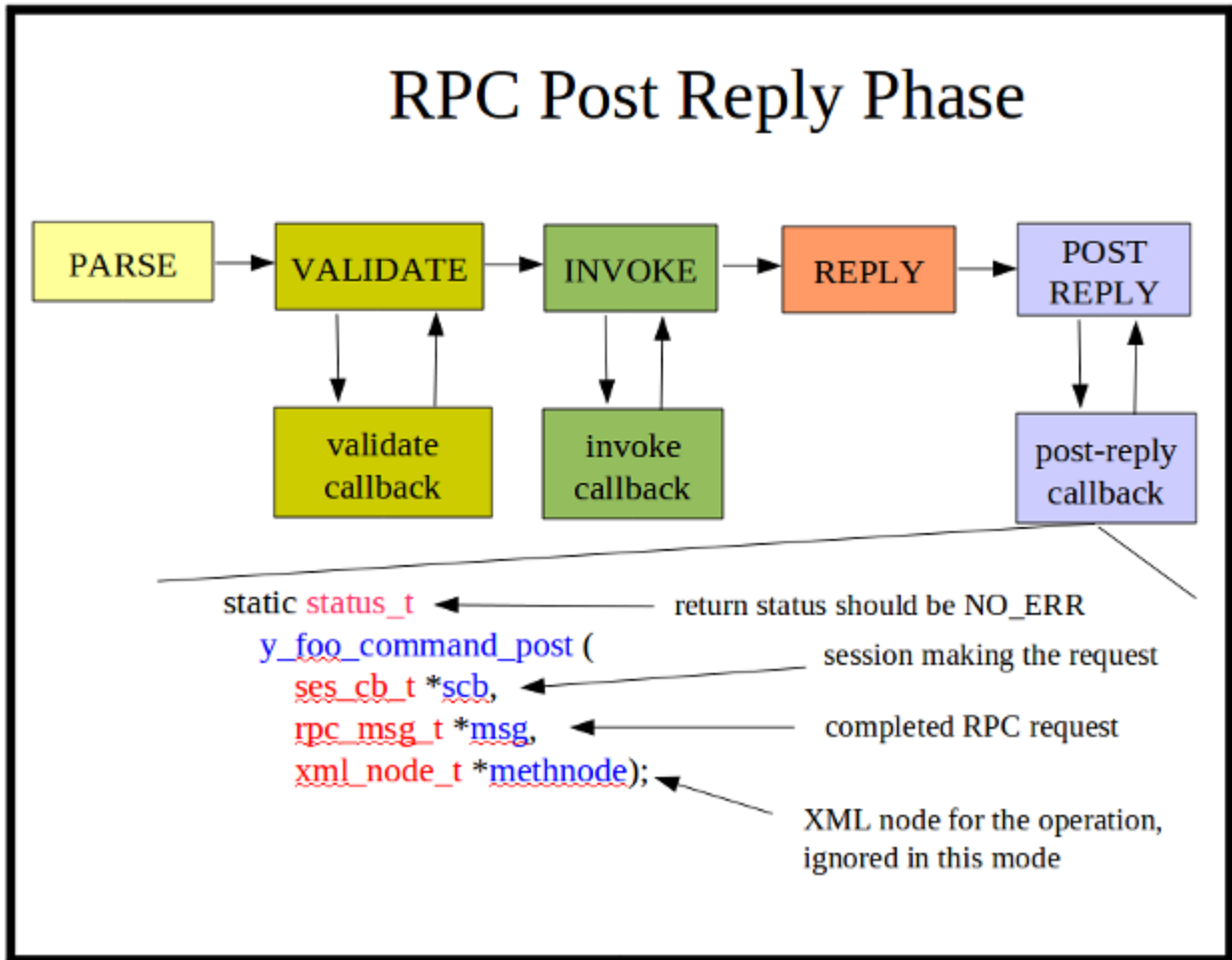
The RPC post-reply callback function is used to clean up after a message has been processed. Only 1 function can register for each YANG rpc statement. The standard NETCONF operations are reserved by the server engine. This callback is not needed unless the SIL validate or invoke callback allocated some memory that needs to deleted after the <rpc-reply> is sent.

The RPC post reply callback function is optional to use. It is enabled with the **agt_rpc_register_method** function, within the phase 1 initialization callback function.

The **yangdump** code generator will not create this SIL callback function by default.

Example SIL Function Registration

```
res = agt_rpc_register_method(

    y_foo_M_foo,

    y_foo_N_command,

    AGT_RPC_PH_POST_REPLY,

    y_foo_command_post);

if (res != NO_ERR) {

    return res;
```

```
    }
```

Example SIL Function:

```
/******************************************************************

* FUNCTION y_foo_command_post

*

* RPC post reply phase

*

* INPUTS:

*     see agt/agt_rpc.h for details

*

* RETURNS:

*     error status

******************************************************************/

static status_t

    y_foo_command_post (

        ses_cb_t *scb,

        rpc_msg_t *msg,

        xml_node_t *methnode)

{

    (void)scb;

    (void)methnode;

    if (msg->rpc_user1 != NULL) {

        m__free(msg->rpc_user1);

        msg->rpc_user1 = NULL;

    }

    return NO_ERR;

}   /* y_foo_command_post */
```

5.2 Database Operations

The server database is designed so that the SIL callback functions do not need to really know which database model is being used by the server (e.g., target is candidate vs. running configuration).

There are three SIL database edit callback phases:

1. **Validate**: Check the parameters no matter what database is the target

2. **Apply**: The server will manipulate the database nodes as needed. The SIL usually has nothing to do in this phase unless internal resources need to be reserved.

3. **Commit or Rollback**: Depending on the result of the previous phases, either the commit or the rollback callback phase will be invoked, if and when the changes are going to be finalized in the running configuration.

The SIL code is not responsible for maintaining the value tree for any database. This is done by the server.

The SIL database edit callback code is responsible for the following tasks:

- Perform any data-model specific validation that is not already covered by a machine-readable statement, during the validation phase.

- Reserve any data-model specific resources for the proposed new configuration content, during the apply phase.

- Activate any data-model behavior changes based on the new configuration content, during the commit phase.

- Release any reserved resources that were previously allocated in the apply phase, during the rollback phase.

5.2.1 DATABASE TEMPLATE (CFG_TEMPLATE_T)

Every NETCONF database has a common template control block, and common set of access functions.

NETCONF databases are not separate entities like separate SQL databases. Each NETCONF database is conceptually the same database, but in different states:

- **candidate**: A complete configuration that may contain changes that have not been applied yet. This is only available if the :candidate capability is advertised by the server. The value tree for this database contains only configuration data nodes.

- **running**: The complete current server configuration. This is available on every NETCONF server.   The value tree for this database contains configuration data nodes and non-configuration nodes created and maintained by the server. The server will maintain read-only nodes when <edit-config>, <copy-config>, or <commit> operations are performed on the running configuration. SIL code should not alter the data nodes within a configuration directly. This work is handled by the server. SIL callback code should only alter its own data structures, if needed.

- **startup**: A complete configuration that will be used upon the next reboot of the device. This is only available if the :startup capability is advertised by the server.   The value tree for this database contains only configuration data nodes.

NETCONF also recognized external files via the <url> parameter, if the :url capability is advertised by the server. These databases will be supported in a future release of the server. The NETCONF standard does not require that these external databases support the same set of protocol operations as the standard databases, listed above. A client application can reliably copy from and to an external database, but editing and filtered retrieval may not be supported.

The following typedef is used to define a NETCONF database template:

```
/* struct representing 1 configuration database */

typedef struct cfg_template_t_ {

    dlq_hdr_t       qhdr;

    ncx_cfg_t       cfg_id;

    cfg_location_t  cfg_loc;

    cfg_state_t     cfg_state;
```

```
    xmlChar        *name;

    xmlChar        *src_url;

    xmlChar        *load_time;

    xmlChar        *lock_time;

    xmlChar        *last_ch_time;

    uint32          flags;

    ses_id_t        locked_by;

    cfg_source_t   lock_src;

    dlq_hdr_t       load_errQ;    /* Q of rpc_err_rec_t */

    val_value_t   *root;           /* btyp == container */
} cfg_template_t;
```

The following table highlights the fields in the cfg_template_t data structure:

cfg_template_t Fields

Field	Description
qhdr	Queue header to allow the object template to be stored in a queue.
cfg_id	Internal configuration ID assigned to this configuration.
cfg_loc	Enumeration identifying the configuration source location.
cfg_state	Current internal configuration state.
name	Name string for this configuration.
src_url	URL for use with 'cfg_loc' to identify the configuration source.
load_time	Date and time string when the configuration was loaded.
lock_time	Date and time string when the configuration was last locked.
last_ch_time	Date and time string when the configuration was last changed.
flags	Internal configuration flags. Do not use directly.
locked_by	Session ID that owns the global configuration lock, if the database is currently locked.
lock_src	If the database is locked, identifies the protocol or other source that currently caused the database to be locked.
load_errQ	Queue of rpc_err_rec_t structures that represent any <rpc-error> records that were generated when the configuration was loaded, if any.
root	The root of the value tree representing the entire

Field	Description
	database.

5.2.2   DATABASE ACCESS FUNCTIONS

The file **ncx/cfg.h** contains some high-level database access functions that may be of interest to SIL callback functions for custom RPC operations. All database access details are handled by the server if the database edit callback functions are used (associated with a particular object node supported by the server)..The following table highlights the most commonly used functions. Refer to the H file for a complete definition of each API function.

cfg_template_t Access Functions

Function	Description
cfg_new_template	Create a new configuration database.
cfg_free_template	Free a configuration database.
cfg_get_state	Get the current internal database state.
cfg_get_config	Get a configuration database template pointer, from a configuration name string.
cfg_get_config_id	Get a configuration database template pointer, from a configuration ID.
cfg_fill_candidate_from_inline	Fill the candidate database from an internal value tree data structure.
cfg_get_dirty_flag	Returns TRUE if the database has changes in it that have not been saved yet. This applies to the candidate and running databases at this time.
cfg_ok_to_lock	Check if the database could be successfully locked by a specific session.
cfg_ok_to_unlock	Check if the database could be successfully unlocked by a specific session.
cfg_ok_to_read	Check if the database is in a state where read operations are allowed.
cfg_ok_to_write	Check if the database could be successfully written by a specific session. Checks the global configuration lock, if any is set.
cfg_is_global_locked	Returns TRUE if the database is locked right now with a global lock.
cfg_get_global_lock_info	Get some information about the current global lock on the database.
cfg_lock	Get a global lock on the database.
cfg_unlock	Release the global lock on the database.
cfg_release_locks	Release all locks on all databases

5.2.3   DATABASE CALLBACK INITIALIZATION AND CLEANUP

The file **agt/agt_cb.h** contains functions that a SIL developer needs to register and unregister database edit callback functions. The same callback function can be used for different phases, if desired.

The following function template definition is used for all SIL database edit callback functions:

```
/* Callback function for agent object handler
 * Used to provide a callback sub-mode for
 * a specific named object
 *
 * INPUTS:
 *    scb == session control block making the request
 *    msg == incoming rpc_msg_t in progress
 *    cbtyp == reason for the callback
 *    editop == the parent edit-config operation type, which
 *              is also used for all other callbacks
 *              that operate on objects
 *    newval == container object holding the proposed changes to
 *              apply to the current config, depending on
 *              the editop value. Will not be NULL.
 *    curval == current container values from the <running>
 *              or <candidate> configuration, if any. Could be NULL
 *              for create and other operations.
 *
 * RETURNS:
 *     status:
 */
typedef status_t
    (*agt_cb_fn_t) (ses_cb_t   *scb,
                rpc_msg_t *msg,
                agt_cbtyp_t cbtyp,
                op_editop_t  editop,
                val_value_t  *newval,
                val_value_t  *curval);
```

SIL Database Callback Template

Parameter	Description
scb	The session control block making the edit request. The SIL callback code should not

Parameter	Description
	need this parameter except to pass to functions that need the SCB.

Parameter	Description
msg	Incoming RPC message in progress. The server uses some fields in this structure, and there are 2 SIL fields for the RPC callback functions. The SIL callback code should not need this parameter except to pass to functions that need the message header.
cbtyp	Enumeration for the callback phase in progress..
editop	The edit operation in effect as this node is being processed.
newval	Value node containing the new value for a create, merge, replace, or insert operation. The 'newval' parm may be NULL and should be ignored for a delete operation. In that case, the 'curval' pointer contains the node being deleted.
curval	Value node containing the current database node that corresponds to the 'newval' node, if any is available.

A SIL database edit callback function is hooked into the server with the **agt_cb_register_callback** or **agt_cb_register_callbacks** functions, described below.   The SIL code generated by yangdump uses the first function to register a single callback function for all callback phases.

```
extern status_t

    agt_cb_register_callback (const xmlChar *modname,

                      const xmlChar *defpath,

                      const xmlChar *version,

                      const agt_cb_fn_t cbfn);
```

agt_cb_register_callback

Parameter	Description
modname	Module name string that defines this object node.
defpath	Absolute path expression string indicating which node the callback function is for.
version	If non-NULL, indicates the exact module version expected.
cbfn	The callback function address. This function will be used for all callback phases.

```
extern status_t

    agt_cb_register_callbacks (const xmlChar *modname,

                        const xmlChar *defpath,

                        const xmlChar *version,

                        const agt_cb_fnset_t *cbfnset);
```

agt_cb_register_callbacks

Parameter	Description
modname	Module name string that defines this object node.
defpath	Absolute path expression string indicating which node the callback function is for.
version	If non-NULL, indicates the exact module version expected.
cbfnset	The callback function set structure, fillied in with the addesses of all desired callback phases. Any NULL slots will cause that phase to be skipped.

The **agt_cb_unregister_callbacks** function is called during the module cleanup. It is generated by **yangdump** automatically for all RPC operations.

```
extern void

    agt_cb_unregister_callbacks (const xmlChar *modname,

                        const xmlChar *defpath);
```

agt_cb_unregister_callbacks

Parameter	Description
modname	Module name string that defines this object node.
defpath	Absolute path expression string indicating which node the callback function is for.

5.2.4   EXAMPLE SIL DATABASE EDIT CALLBACK FUNCTION

The following example code is from the **libtoaster** source code, available in **yuma-dev** and **yuma-source** packages.

```
    /*****************************************************************
    * FUNCTION y_toaster_toaster_edit
```

```
 *
 * Edit database object callback
 * Path: /toaster
 * Add object instrumentation in COMMIT phase.
 *
 * INPUTS:
 *     see agt/agt_cb.h for details
 *
 * RETURNS:
 *     error status
 ********************************************************************/
static status_t
    y_toaster_toaster_edit (
        ses_cb_t *scb,
        rpc_msg_t *msg,
        agt_cbtyp_t cbtyp,
        op_editop_t editop,
        val_value_t *newval,
        val_value_t *curval)
{
    status_t res;
    val_value_t *errorval;
    const xmlChar *errorstr;

    res = NO_ERR;
    errorval = NULL;
    errorstr = NULL;

    switch (cbtyp) {
    case AGT_CB_VALIDATE:
        /* description-stmt validation here */
        break;
    case AGT_CB_APPLY:
        /* database manipulation done here */
        break;
    case AGT_CB_COMMIT:
        /* device instrumentation done here */
        switch (editop) {
        case OP_EDITOP_LOAD:
            toaster_enabled = TRUE;
            toaster_toasting = FALSE;
            break;
```

```
        case OP_EDITOP_MERGE:
            break;
        case OP_EDITOP_REPLACE:
            break;
        case OP_EDITOP_CREATE:
            toaster_enabled = TRUE;
            toaster_toasting = FALSE;
            break;
        case OP_EDITOP_DELETE:
            toaster_enabled = FALSE;
            if (toaster_toasting) {
                agt_timer_delete(toaster_timer_id);
                toaster_timer_id = 0;
                toaster_toasting = FALSE;
                y_toaster_toastDone_send((const xmlChar *)"error");
            }
            break;
        default:
            res = SET_ERROR(ERR_INTERNAL_VAL);
        }


        if (res == NO_ERR) {
            res = agt_check_cache(
                &toaster_val,
                newval,
                curval,
                editop);
        }


        if (res == NO_ERR &&
            (editop == OP_EDITOP_LOAD || editop == OP_EDITOP_CREATE)) {
            res = y_toaster_toaster_mro(newval);
        }
        break;
    case AGT_CB_ROLLBACK:
        /* undo device instrumentation here */
        break;
    default:
        res = SET_ERROR(ERR_INTERNAL_VAL);
    }


    /* if error: set the res, errorstr, and errorval parms */
```

```
    if (res != NO_ERR) {
        agt_record_error(
            scb,
            &msg->mhdr,
            NCX_LAYER_CONTENT,
            res,
            NULL,
            NCX_NT_STRING,
            errorstr,
            NCX_NT_VAL,
            errorval);
    }


    return res;


} /* y_toaster_toaster_edit */
```

5.2.5 Database Edit Validate Callback Phase

Database Edit Validate Phase
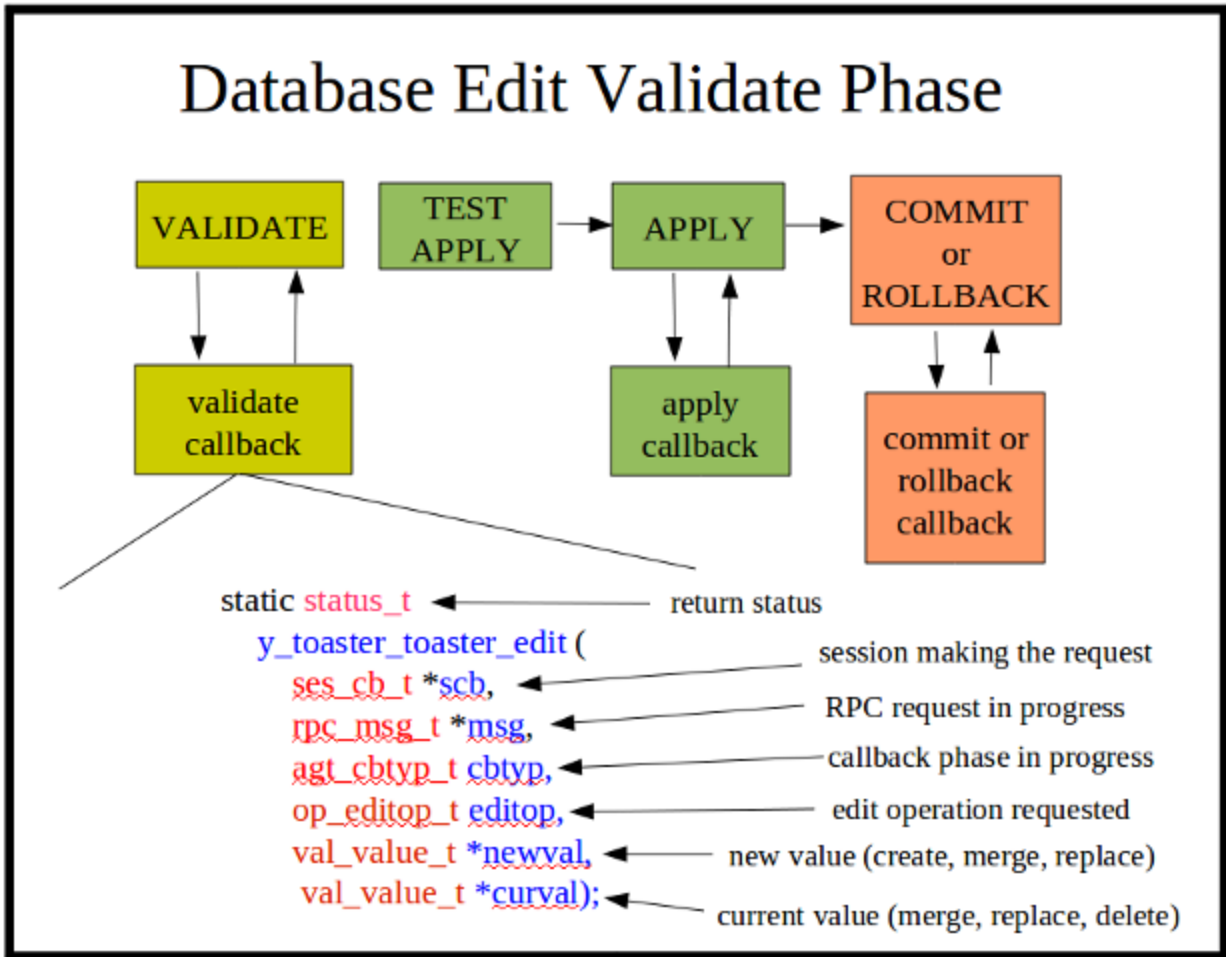
A SIL database validation phase callback function is responsible for checking all the 'description statement' sort of data model requirements that are not covered by any of the YANG machine-readable statements.

For example, if a 'user name' parameter needed to match an existing user name in **/etc/passwd,** then the SIL validation callback would call the system APIs needed to check if the 'newval' string value matched a valid user name. The server will make sure the user name is well-formed and could be a valid user name.

5.2.6   DATABASE EDIT APPLY CALLBACK PHASE

The callback function for this phase is called when database edits are being applied to the running configuration. The resources needed for the requested operation may be reserved at this time, if needed.

5.2.7   DATABASE EDIT COMMIT CALLBACK PHASE

This callback function for this phase is called when database edits are being committed to the running configuration. The SIL callback function is expected to finalize and apply any data-model dependent system behavior at this time.

5.2.8 DATABASE EDIT ROLLBACK CALLBACK PHASE

This callback function for this phase is called when database edits are being undone, after some apply phase or commit phase callback function returned an error, or a confirmed commit operation timed out.

The SIL callback function is expected to release any resources it allocated during the apply or commit phases. Usually only the commit or the rollback function will be called for a given SIL callback, but it is possible for both to be called. For example, if the 'rollback-on-error' option is in effect, and some SIL commit callback fails after your SIL commit callback succeeds, then your SIL rollback callback may be called as well.

5.2.9 DATABASE VIRTUAL NODE GET CALLBACK FUNCTION

A common SIL callback function to use is a virtual node 'get' function. A virtual node can be either a configuration or non-configuration node, but is more likely to be a non-configuration node, such as a counter or hardware status object.

The function **agt_make_virtual_leaf** in **agt/agt_util.h** is a common API used for creating a virtual leaf within an existing parent container.

The following typedef defines the **getcb_fn_t** template, used by all virtual callback functions. This function is responsible for filling in a value node with the current instance value. The status NO_ERR is returned if this is done successfully.

```
/* getcb_fn_t
 *
 * Callback function for agent node get handler
 *
 * INPUTS:
 *    scb    == session that issued the get (may be NULL)
 *              can be used for access control purposes
 *    cbmode == reason for the callback
 *    virval == place-holder node in the data model for
 *                this virtual value node
 *    dstval == pointer to value output struct
 *
 * OUTPUTS:
 *    *fil may be adjusted depending on callback reason
 *    *dstval should be filled in, depending on the callback reason
 *
 * RETURNS:
 *     status:
 */
typedef status_t
    (*getcb_fn_t) (ses_cb_t *scb,
              getcb_mode_t cbmode,
              const val_value_t *virval,
```

```
                    val_value_t *dstval);
```

The following table describes the parameters.

getcb_fn_t Parameters

Parameter	Description
scb	This is the session control block making the request, if available. This pointer may be NULL, so in the rare event this parameter is needed by the SIL callback function, it should be checked first.
cbmode	The callback type. The only supported enumeration at this time is GETCB_GET_VALUE. Other values may be used in the future for different retrieval modes.
virval	The virtual value node in the data tree that is being referenced, The **val_get_virtual_value** function was called for this value node.
dstval	The destination value node that needs to be filled in. This is just an empty value node that has been malloced and then initialized with the **val_init_from_template** function. The SIL callback function needs to set the value properly. The **val_set_simval** and **val_set_simval_obj** functions are two API functions that can be used for this purpose.

The following example from **agt/agt_ses.c** shows a SIL get callback function for the ietf-netconf-monitoring data model, which returns the 'in-sessions' counter value:

```
/*******************************************************************
* FUNCTION agt_ses_get_inSessions
*
* <get> operation handler for the inSessions counter
*
* INPUTS:
*    see ncx/getcb.h getcb_fn_t for details
*
* RETURNS:
*    status
*******************************************************************/
status_t
    agt_ses_get_inSessions (ses_cb_t *scb,
```

```
                              getcb_mode_t cbmode,

                              const val_value_t *virval,

                              val_value_t  *dstval)
{

    (void)scb;

    (void)virval;


    if (cbmode == GETCB_GET_VALUE) {

        VAL_UINT(dstval) = agttotals->inSessions;

        return NO_ERR;

    } else {

        return ERR_NCX_OPERATION_NOT_SUPPORTED;

    }


} /* agt_ses_get_inSessions */
```

The following example from **agt/agt_state.c** shows a complex get callback function for the same data model, which returns the entire <capabilities> element when it is requested. This is done by simply cloning the 'official copy' of the server capabilities that is maintained by the agt/agt_caps.c module.

```
/*****************************************************************

 * FUNCTION get_caps

 *

 * <get> operation handler for the capabilities NP container

 *

 * INPUTS:

 *    see ncx/getcb.h getcb_fn_t for details

 *

 * RETURNS:

 *    status

 *****************************************************************/

static status_t

    get_caps (ses_cb_t *scb,

              getcb_mode_t cbmode,

              val_value_t *virval,

              val_value_t  *dstval)
{

    val_value_t        *capsval;

    status_t            res;
```

```
     (void)scb;

     (void)virval;

     res = NO_ERR;


     if (cbmode == GETCB_GET_VALUE) {

         capsval = val_clone(agt_cap_get_capsval());

         if (!capsval) {

             return ERR_INTERNAL_MEM;

         } else {

             /* change the namespace to this module,

              * and get rid of the netconf NSID

              */

             val_change_nsid(capsval, statemod->nsid);

             val_move_children(capsval, dstval);

             val_free_value(capsval);

         }

     } else {

         res = ERR_NCX_OPERATION_NOT_SUPPORTED;

     }

     return res;


}   /* get_caps */
```

5.3   Notifications

The **yangdump** program will automatically generate functions to queue a specific notification type for processing. It is up to the SIL callback code to invoke this function when the notification event needs to be generated. The SIL code is expected to provide the value nodes that are needed for any notification payload objects.

5.3.1   NOTIFICATION SEND FUNCTION

The function to generate a notification control block and queue it for notification replay and delivery is generated by the **yangdump** program. A function parameter will exist for each top-level data node defined in the YANG notification definition.

In the example below, the 'toastDone' notification event contains just one leaf, called the 'toastStatus'. There is SIL timer callback code which calls this function, and provides the final toast status, after the <make-toast> operation has been completed or canceled.

```
/****************************************************************
```

```
 * FUNCTION y_toaster_toastDone_send
 *
 * Send a y_toaster_toastDone notification
 * Called by your code when notification event occurs
 *
 ********************************************************************/
void
    y_toaster_toastDone_send (
        const xmlChar *toastStatus)
{
    agt_not_msg_t *notif;
    val_value_t *parmval;
    status_t res;

    res = NO_ERR;

    if (LOGDEBUG) {
        log_debug("\nGenerating <toastDone> notification");
    }

    notif = agt_not_new_notification(toastDone_obj);
    if (notif == NULL) {
        log_error("\nError: malloc failed, cannot send <toastDone> notification");
        return;
    }

    /* add toastStatus to payload */
    parmval = agt_make_leaf(
        toastDone_obj,
        y_toaster_N_toastStatus,
        toastStatus,
        &res);
    if (parmval == NULL) {
        log_error(
            "\nError: make leaf failed (%s), cannot send <toastDone> notification",
            get_error_string(res));
    } else {
        agt_not_add_to_payload(notif, parmval);
    }

    agt_not_queue_notification(notif);
```

```
} /* y_toaster_toastDone_send */
```

5.4   Periodic Timer Service

Some SIL code may need to be called at periodic intervals to check system status, update counters, and/or perhaps send notifications.

The file **agt/agt_timer.h** contains the timer access function declarations.

This section provides a brief overview of the SIL timer service.

5.4.1   TIMER CALLBACK FUNCTION

The timer callback function is expected to do a short amount of work, and not block the running process. The function returns zero for a normal exit, and -1 if there was an error and the timer should be destroyed.

The **agt_timer_fn_t** template in **agt/agt_timer.h** is used to define the SIL timer callback function prototype. This typedef defines the callback function template expected by the server for use with the timer service:

```
/* timer callback function
 *
 * Process the timer expired event
 *
 * INPUTS:
 *    timer_id == timer identifier
 *    cookie == context pointer, such as a session control block,
 *           passed to agt_timer_set function (may be NULL)
 *
 * RETURNS:
 *     0 == normal exit
 *    -1 == error exit, delete timer upon return
 */
typedef int (*agt_timer_fn_t) (uint32  timer_id,
                        void *cookie);
```

The following table describes the parameters for this callback function:

SIL Timer Callback Function Parameters

Parameter	Description
timer_id	The timer ID that was returned when the **agt_timer_create** function was called.

Parameter	Description
cookie	The cookie parameter value that was passed to the server when the **agt_timer_create** function was called.

5.4.2   TIMER ACCESS FUNCTIONS

A SIL timer can be set up as a periodic timer or a one-time event.

The timer interval (in seconds) and the SIL timer callback function are provided when the timer is created. A timer can also be restarted if it is running, and the time interval can be changed as well.

The following table highlights the SIL timer access functions in **agt/agt_timer.h:**

SIL Timer Access Functions

Function	Description
agt_timer_create	Create a SIL timer.
agt_timer_restart	Restart a SIL timer
agt_timer_delete	Delete a SIL timer

5.4.3   EXAMPLE TIMER CALLBACK FUNCTION

The following example from toaster.c simply completes the toast when the timer expires and calls the auto-generated 'toastDone' send notification function:

```
    /*****************************************************************
    * FUNCTION toaster_timer_fn
    *
    * Added timeout function for toaster function
    *
    * INPUTS:
    *     see agt/agt_timer.h
    *
    * RETURNS:
    *     0 for OK; -1 to kill periodic timer
    *****************************************************************/
    static int
        toaster_timer_fn (uint32 timer_id,
                          void *cookie)
    {
        (void)timer_id;
        (void)cookie;
```

```
    /* toast is finished */

    toaster_toasting = FALSE;

    toaster_timer_id = 0;

    if (LOGDEBUG2) {

        log_debug2("\ntoast is finished");

    }

    y_toaster_toastDone_send((const xmlChar *)"done");

    return 0;


} /* toaster_timer_fn */
```

6   Development Environment

This section describes the Yuma Tools development environment used to produce the Linux binaries.

6.1   Programs and Libraries Needed

There are several components used in the Yuma software development environment:

- gcc compiler and linker
- ldconfig and install programs
- GNU make program
- shell program, such as bash
- Yuma development tree: the source tree containing Yuma Tools code, specified with the **$$YUMA_HOME** environment variable.
- SIL development tree: the source tree containing server instrumentation code

The following external program is used by Yuma, and needs to be pre-installed:

- **opensshd (needed by netconfd)**
  - The SSH2 server code does not link with **netconfd**. Instead, the **netconf-subsystem** program is invoked, and local connections are made to the **netconfd** server from this SSH2 subsystem.

The following program is part of Yuma Tools, and needs to be installed:

- **netconf-subsystem (needed by netconfd)**
  - The thin client sub-program that is called by **sshd** when a new SSH2 connection to the 'netconf' sub-system is attempted.
    - This program will use an AF_LOCAL socket, using a proprietary **<ncxconnect>** message, to communicate with the **netconfd** server..

- After establishing a connection with the **netconfd** server, this program simply transfers SSH2 NETCONF channel data between **sshd** and **netconfd**.

The following program is part of Yuma Tools, and usually found within the Yuma development tree:

- **netconfd**
  - The NETCONF server that processes all protocol operations.
    - The **agt_ncxserver** component will listen for **<ncxconnect>** messages on the designated socket (/tmp/ncxserver.sock). If an invalid message is received, the connection will be dropped. Otherwise, the **netconf-subsystem** will begin passing NETCONF channel data to the netconfd server. The first message is expected to be a valid NETCONF <hello> PDU. If

The following external libraries are used by Yuma, and need to be pre-installed:

- **ncurses (needed by yangcli)**
  - character processing library needed by **libtecla**; used within **yangcli**
- **libc (or glibc, needed by all applications)**
  - unix system library
- **libssh2 (needed by yangcli)**
  - SSH2 client library, used by yangcli
- **libxml2 (needed by all applications)**
  - xmlTextReader XML parser
  - pattern support

6.2   SIL Makefile

The SIL Makefile is based on the Makefile for Yuma sources. The automake program is not used at this time. There is no ./configure script to run. There are 2 basic build steps:

1. make
2. [sudo] make install

The installation step may require root access via the **sudo** program, depending on the system.

6.2.1   TARGET PLATFORMS

The following target platforms are supported:

- **Fedora and Ubuntu**: these are the default targets and no special command line options are needed.
- **MacOSX**: the command "MAC=1" is added to the command line.
- **IWL Mini-Maxwell**: the command "MAXWELL=1" is added to the command line.

6.2.2   BUILD TARGETS

The following table describes the build targets that are supported:

Yuma Build Targets

Make Target	Description
all	Make everything. This is the default if not target is specified.
depend	Make the dependencies files.
clean	Remove the target files.
superclean	Remove all the dependencies and the target files.
distclean	Remove all the distribution files that make be installed, all the dependencies and the target files.
test	Make any test code.
lint	Run a lint program on the source files.
install	Install the components into the system.
uninstall	Remove the components from the system.

6.2.3 COMMAND LINE BUILD OPTIONS

There are several command line options that can be used when making the Yuma source code. These may have an impact on building and linking the SIL source code. The following table describes these options:

Yuma Build Parameters for 'make'

Make Parameter	Description
DEBUG=1	Generate debug symbols for **gdb** debugging, and do not generate optimized binary code. The default is not to generate symbols, and instead use -O2 optimization.
STATIC=1	Build static libraries instead of dynamic libraries, where needed. This sometimes makes debugging simpler and faster. Embedded systems without dynamic library support need to use this make option.
MEMTRACE=1	Enabled 'mtrace' memory leak debugging to identify the exact nodes which were malloced but never freed.
MAC=1	Build the software for the MacOSX platform.
MAXWELL=1	Build the software for the IWL Mini-Maxwell platform.
RELEASE=n	Builds release 'n' (version-release) for Debian or RPM package distribution

Make Parameter	Description
STATIC_SERVER	Defining this make flag forces the server not to require or use the 'dylib' functions to dynamically load SIL libraries at run-time, without requiring ldconfig. Instead, the server will assume the server code has been modified so these SIL libraries are initialized and cleaned up statically.
NOFLOAT	Defining this make flag will force the server to implement XPath numbers with a string representation instead of using the 'double' data type from the floating point library. This will impact the correctness of XPath expression evaluation, so this should not be done unless the <math.h> library support is not available.

6.2.4   EXAMPLE SIL MAKEFILE

The script **/usr/bin/make_sil_dir.sh** is used to automatically create a SIL work sub-directory tree. Support files are located in the **/usr/share/yuma/util** directory.

The following Makefile is for the **libtoaster** library, supporting the **toaster.yang** module:

```
# SIL Makefile for Yuma Server Instrumentation Library

#


############### SOURCE PROFILE #############################


SUBDIR_NM=toaster


SUBDIR_CPP=


############### TARGET PROFILE #############################


TARGET=../bin


LIB_INST=../lib


WORK_INST=$(YUMA_HOME)/target/lib


REAL_INST=$(DESTDIR)/usr/lib/yuma



#################### MAKE RULES #########################
```

```
all: sil_dummy sil_lib


#################### PLATFORM DEFINITIONS ############


ifdef YUMA_HOME
CINC = -I. \
     -I$(YUMA_HOME)/src/platform \
     -I$(YUMA_HOME)/src/ncx \
     -I$(YUMA_HOME)/src/agt \
     -I/usr/include \
     -I/usr/include/libxml2 \
     -I/usr/include/libxml2/libxml
else
ifdef YUMA_INSTALL
CINC = -I. \
     -I$(YUMA_INSTALL)/src/platform \
     -I$(YUMA_INSTALL)/src/ncx \
     -I$(YUMA_INSTALL)/src/agt \
     -I/usr/include \
     -I/usr/include/libxml2 \
     -I/usr/include/libxml2/libxml
else
CINC = -I. \
     -I/usr/share/yuma/src/netconf/src/platform \
     -I/usr/share/yuma/src/netconf/src/ncx \
     -I/usr/share/yuma/src/netconf/src/agt \
     -I/usr/include \
     -I/usr/include/libxml2 \
     -I/usr/include/libxml2/libxml
endif
endif


# added /sw/include for MacOSX
ifdef MAC
# MACOSX version
  CINC +=-I/sw/include
  CFLAGS += -DMACOSX=1
endif


LBASE=../lib
```

```
ifdef DESTDIR
OWNER=
else
ifdef MAC
OWNER=-oroot
else
OWNER= --owner=root
endif
endif


### GCC + [LINUX or MACOSX]

CWARN=-Wall -Wno-long-long -Wformat-y2k -Winit-self \
     -Wmissing-include-dirs -Wswitch-default -Wunused-parameter \
     -Wextra -Wundef -Wshadow -Wpointer-arith \
     -Wwrite-strings -Wbad-function-cast -Wcast-qual -Wcast-align \
     -Waggregate-return -Wstrict-prototypes -Wold-style-definition \
     -Wmissing-prototypes -Wmissing-declarations \
     -Wpacked -Winvalid-pch \
     -Wredundant-decls -Wnested-externs -Winline -std=gnu99 -Werror



# -Wunreachable-code removed due to -O3
# -O3 changed to -O2 due to code bloat from inline functions


CDEFS=-DDEBUG -DLINUX -DGCC


ifndef NOFLOAT
  CDEFS += -DHAS_FLOAT
endif


CFLAGS=$(CDEFS) $(CWARN) -fPIC


# production (0) or debug (1) build
ifdef DEBUG
  CFLAGS += -ggdb3
else
  CFLAGS += -O2
endif


# memory leak debugging mode
ifdef MEMTRACE
```

```
   CFLAGS += -DMEMORY_DEBUG=1
endif


# free or SDK version
ifdef FREE
   CFLAGS += -DFREE_VERSION
endif


ifdef RELEASE
   CFLAGS += -DRELEASE=$(RELEASE)
endif


ifdef MAC
    GRP=
else
ifdef DESTDIR
    GRP=
else
    GRP=--group=root
endif
endif


ifdef STATIC
LIBSUFFIX=a
else
ifdef MAC
LIBSUFFIX=dylib
else
LIBSUFFIX=so
endif
endif


CC=gcc
LINK=gcc
LINT=splint
LINTFLAGS= '-weak -macrovarprefix "m_"'
##LIBFLAGS=-lsocket


LIBTOOL=ar
#LFLAGS=-v --no-as-needed
LFLAGS=-lm
LPATH=-L$(LBASE)
```

```
    CEES = $(wildcard *.c)


    HEES = $(wildcard *.h)


    ############### OBJS RULE #############
    OBJS = $(patsubst %.c,$(TARGET)/%.o,$(CEES))


    ############### DEPS RULE #############
    DEPS = $(patsubst %.c,%.D,$(wildcard *.c))


    ####################### PLATFORM DEFINITIONS #############
    PLATFORM_CPP=


    .PHONY: all superclean clean test install uninstall \
            distclean depend lint


    ######################### MAKE DEPENDENCIES ##############
    COMPILE.c= $(CC) $(CFLAGS) $(CPPFLAGS) $(PLATFORM_CPP) \
               $(CINC) $(SUBDIR_CPP) $(TARGET_ARCH) -c



    $(TARGET)/%.o: %.c
        $(CC) $(CFLAGS) $(CPPFLAGS) $(PLATFORM_CPP) \
            $(CINC) $(SUBDIR_CPP) $(TARGET_ARCH) -c -o $@ $<



    # Common library rule

    $(LBASE)/lib%.a: $(OBJS)
        $(LIBTOOL) cr $@ $(OBJS)
        ranlib $@


    # dependency rule to make temp .D files from .c sources
    # all the .D files are collected and appended to the
    # appropriate Makefile when 'make depend' is run
    # this rule is kept here to make sure it matches COMPILE.c
    %.D: %.c
        $(CC) -MM -MG -MT $(TARGET)/$(patsubst %.c,%.o,$<) \
        -Wall -Wcomment $(CPPFLAGS) $(PLATFORM_CPP) $(CINC) \
        $(SUBDIR_CPP) $(TARGET_ARCH) -c $< > $@
```

```
################# MAKE DEPENDENCIES ####################
# following depend rule is the GNU version! Other versions TBD
# this rule cannot be included by the top level makefile
# so platform.profile.cmn was created to allow that Makefile
# to define a different 'depend' rule.
depend: dependencies


dependencies: $(DEPS)
     @if [ ! -f Makefile ]; then \
         echo "Error: Makefile missing!"; \
         exit 1; \
     fi
     @rm -f dependencies
     @for i in $(DEPS); do \
       if [ -f $$i ] ; then \
         (cat $$i >> dependencies; echo "" >> dependencies) ; \
       else \
         (echo "*** Warning: Dependency file $i.D is missing! (Skipping...) ***"; \
          echo "# Warning: Missing file $$i !!!") ; \
       fi; \
     done
     @echo "" >> dependencies
# delete the .D files to force make depend to rebuild them each time
# that target is built
#    @rm -f $(DEPS)



################ DEPENDENCIES #########################
# depend rule must be included after the 'all' make rule


include ../../netconf/src/platform/platform.profile.depend



test:


install:
     mkdir -p $(REAL_INST)
     $(SUDO) install $(LIB_INST)/lib$(SUBDIR_NM).$(LIBSUFFIX) $(REAL_INST)


work:
     install $(LIB_INST)/lib$(SUBDIR_NM).$(LIBSUFFIX) $(WORK_INST)
```

```
    sil_lib: $(LIB_INST)/lib$(SUBDIR_NM).$(LIBSUFFIX)


    # this dummy rule keeps make from deleting the $(OBJS) as
    # intermediate files
    sil_dummy: dependencies $(OBJS)


    clean:
        rm -f $(OBJS) $(LIB_INST)/lib$(SUBDIR_NM).*


    superclean:
        rm -f *~  $(DEPS) dependencies $(OBJS) \
        $(LIB_INST)/lib$(SUBDIR_NM).*


    $(LIB_INST)/lib$(SUBDIR_NM).so: $(OBJS)
        gcc -shared -rdynamic -Wl,-soname,lib$(SUBDIR_NM).so -o $@ $(OBJS) -ldl


    #    gcc -shared -Wl,-soname,libtoaster.so -o $@ $(OBJS) -lc


    $(LBASE)/lib$(SUBDIR_NM).dylib: $(OBJS)
        gcc -shared -dynamiclib -std=gnu99 -current_version 1.0 \
        -undefined dynamic_lookup \
        -o $@ -install_name lib$(SUBDIR_NM).dylib $(OBJS) -lxml2


    code:
        yangdump format=h indent=4 module=$(SUBDIR_NM) output=$(SUBDIR_NM).h
        yangdump format=c indent=4 module=$(SUBDIR_NM) output=$(SUBDIR_NM).c



    # prevent the make program from choking on all the symbols
    # that get generated from autogenerated make rules
    .NOEXPORT:


    include dependencies
```

6.3   Automation Control

The YANG language includes many ways to specify conditions for database validity, which traditionally are only documented in DESCRIPTION clauses. The YANG language allows vendors and even data modelers to add new statements to the standard syntax, in a way that allows all tools to skip extension statements that they do not understand.

The **yangdump** YANG compiler sames all the non-standard language statements it finds, even those it does not recognize. These are stores in the **ncx_appinfo_t** data structure in **ncx/ncxtypes.h**..

There are also SIL access functions defined in **ncx/ncx_appinfo.h** that allow these language statements to be accessed. If an argument string was provided, it is saved along with the command name.

Several data structures contains an 'appinfoQ' field to contain all the ncx_appinfo_t stuctures that were generated within the same YANG syntax block (e.g., within a typedef, type, leaf, import statement).

6.3.1   BUILT-IN YANG LANGUAGE EXTENSIONS

There are several YANG extensions that are supported by Yuma. They are all defined in the YANG file named **ncx.yang**. They are used to 'tag' YANG definitions for some sort of automatic processing by Yuma programs. Extensions are position-sensitive, and if not used in the proper context, they will be ignored. A YANG extension statement must be defined (somewhere) for every extension used in a YANG file, or an error will be occur.

Most of these extensions apply to **netconfd** server behavior, but not all of them. For example, the **ncx:hidden** extension will prevent **yangcli** from displaying help for an object containing this extension. Also, **yangdump** will skip this object in HTML output mode.

The following table describes the supported YANG language extensions. All other YANG extension statements will be ignored by Yuma, if encountered in a YANG file:

YANG Language Extensions

extension	description
ncx:hidden;	Declares that the object definition should be hidden from all automatic documentation generation. Help will not be available for the object in **yangcli**.
ncx:metadata "*attr-type attr-name*";	Defines a qualified XML attribute in the module namespace. Allowed within an RPC input parameter. **attr-type** is a valid type name with optional YANG prefix. **attr-name** is the name of the XML attribute.
ncx:no-duplicates;	Declares that the **ncx:xsdlist** data type is not allowed to contain duplicate values. The default is to allow duplicate token strings within an **ncx:xsdlist** value.
ncx:password;	Declares that a string data type is really a password, and will not be displayed or matched by any filter.
ncx:qname;	Declares that a string data type is really an XML qualified name. XML prefixes will be properly generated by **yangcli** and **netconfd**.
ncx:root;	Declares that the container parameter is really a NETCONF database root, like <config> in the <edit-config> operations. The child nodes of this container are not specified in the YANG file. Instead, they are allowed to contain any top-

	level object from any YANG file supported by the server.
ncx:schema-instance;	Declares that a string data type is really an special schema instance identifier string. It is the same as an instance-identifier built-in type except the key leaf predicates are optional. For example, missing key values indicate wild cards that will match all values in **nacm** <dataRule> expressions.
ncx:secure;	Declares that the database object is a secure object. If the object is an **rpc** statement, then only the **netconfd** 'superuser' will be allowed to invoke this operation by default. Otherwise, only read access will be allowed to this object by default, Write access will only be allowed by the 'superuser', by default.
ncx:very-secure;	Declares that the database object is a very secure object. Only the 'superuser' will be allowed to access the object, by default.
ncx:xsdlist *"list-type"*;	Declares that a string data type is really an XSD style list. **list-type** is a valid type name with optional YANG prefix. List processing within <edit-config> will be automatically handled by **netconfd**.
ncx:xpath;	Declares that a string data type is really an XPath expression. XML prefixes and all XPath processing will be done automatically by **yangcli** and **netconfd**.

6.3.2   SIL LANGUAGE EXTENSION ACCESS FUNCTIONS

The following table highlights the SIL functions in ncx/ncx_appinfo.h that allow SIL code to examine any of the non-standard language statements that were found in the YANG module:

Language Extension Access Functions

Function	Description
ncx_find_appinfo	Find an **ncx_appinfo_t** structure by its prefix and name, in a queue of these entries.
ncx_find_next_appinfo	Find the next occurrence of the specified **ncx_appinfo_t** data structure.
ncx_clone_appinfo	Clone the specified **ncx_appinfo_t** data structure.