Yuma® Developer Manual

YANG-Based Unified Modular Automation

Table Of Contents

Yuma Developer Manual

1 Preface

1.1 Legal Statements

1.2 Restricted Rights Legend

1.3 Additional Resources

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

Yuma® Installation and Quickstart Guide

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

Yuma® User Manual

To obtain additional support you may email Netconf Central at the e-mail address

support@netconfcentral.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

- **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.

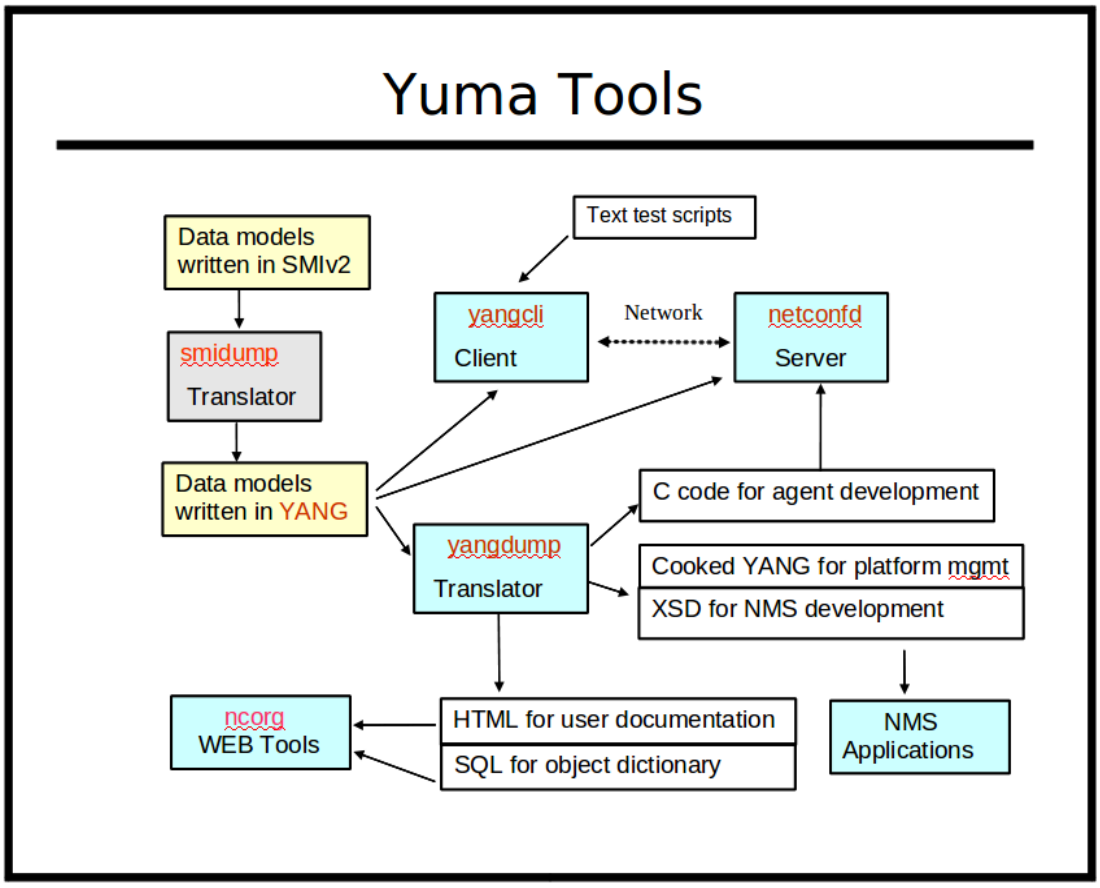
1.4 Conventions Used in this Document

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 Development 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 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.

2.1.4 Basic Development Steps

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

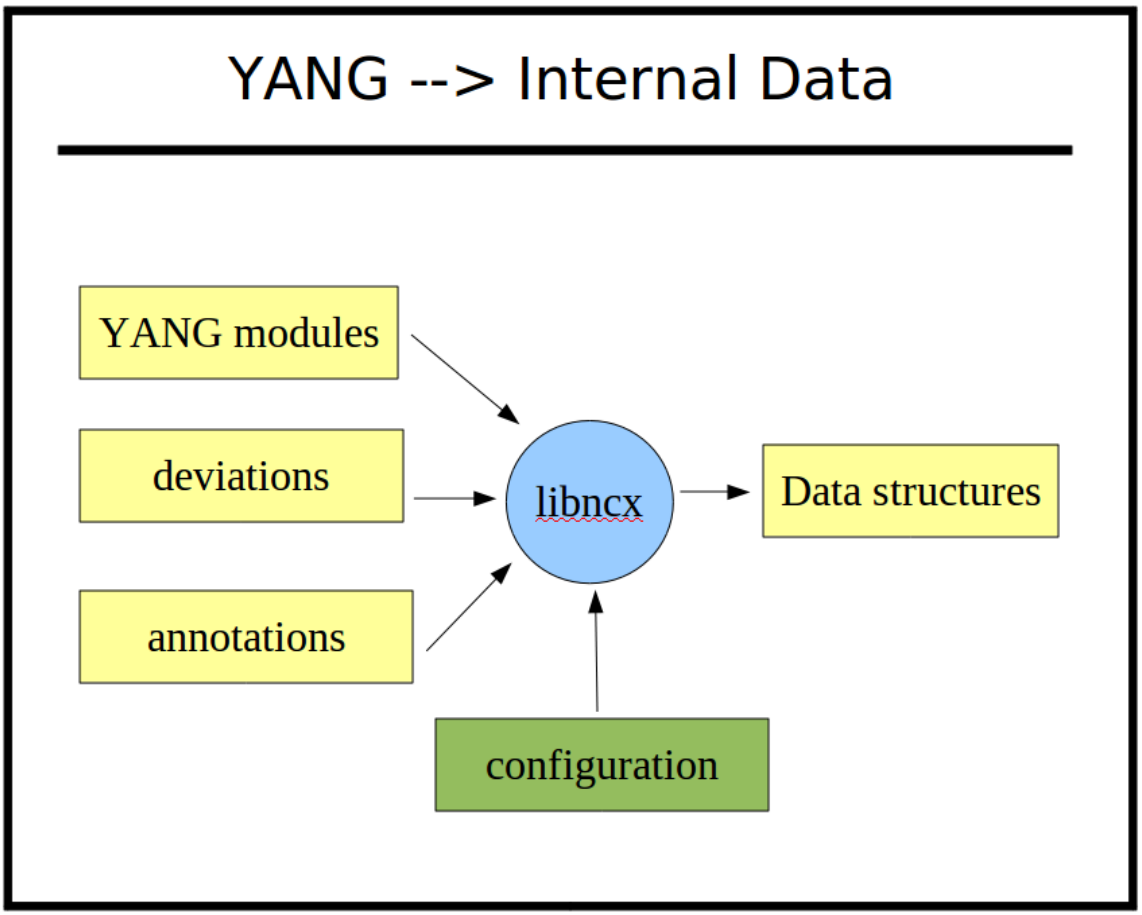
- **Create the YANG module data model** definition, or use an existing YANG module.
  - Validate the YANG module with the **yangdump** program and make sure it does not contain any errors. All warnings should also be examined to determine if they indicate data modeling bugs, or not.
  - `Example toaster.yang`
- 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 **yuma_make_sil** script, found in the **netconf/bin** directory in the Yuma development tree.
  - `Example: mydir> $YUMA_HOME/bin/yuma_make_sil toaster`
- **Create the C source code files** for the YANG module callback functions
  - Generate the H and C files with the 'make code' command, in the SIL 'src' directory

    - `Example: mydir/toaster/src> make code`
- **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.
- **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 Server Design

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

2.2.1 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.

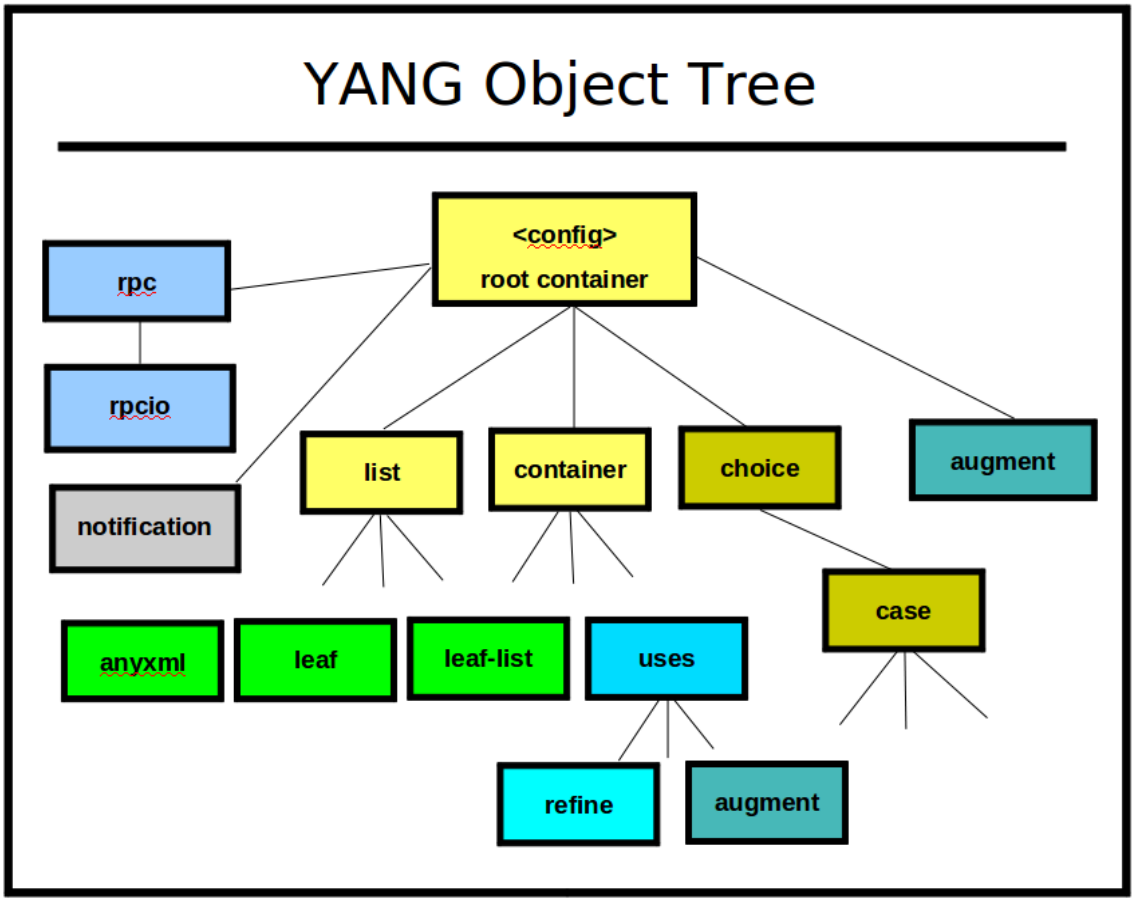
In addition, range statements, patterns, XPath expressions, and all other machine-readable statements are all processed automatically, so the YANG statements themselves are like server source code.

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.

In addition, 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.

2.2.2 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.

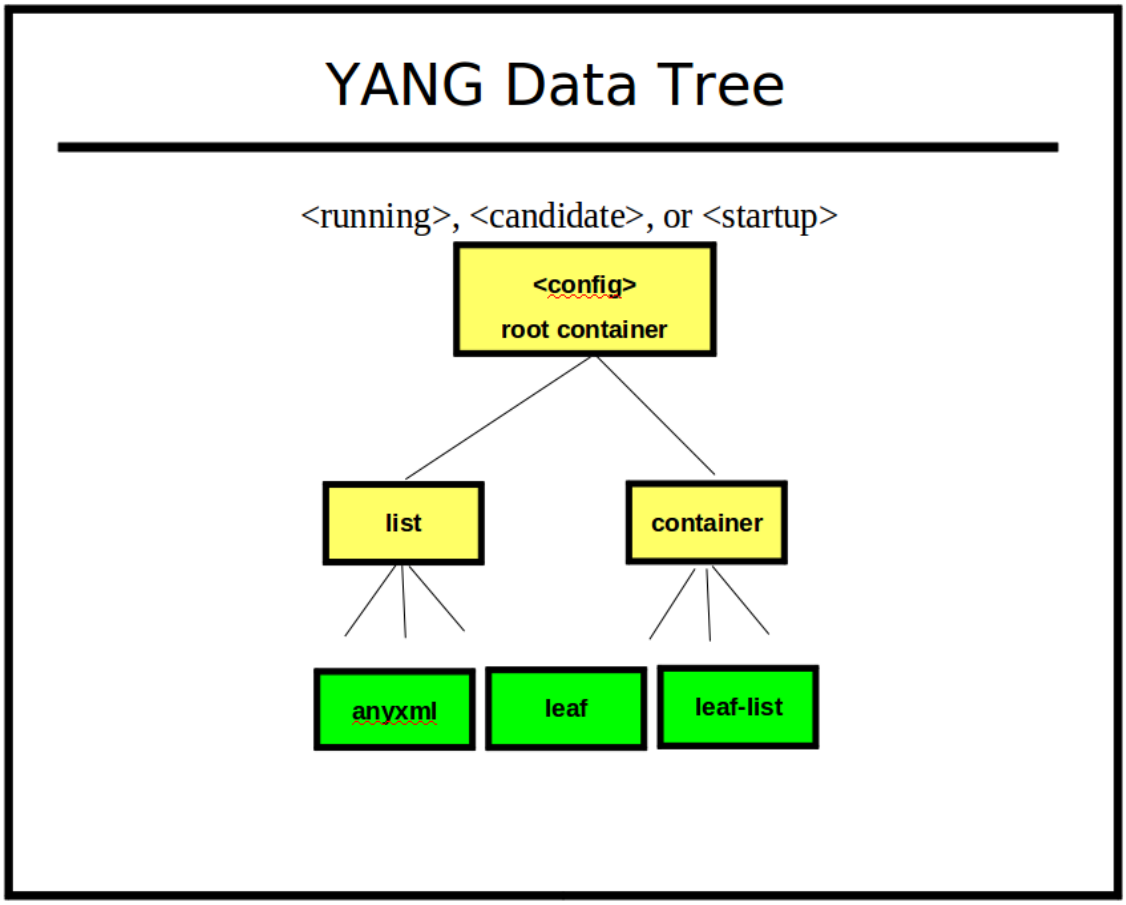
There are 14 different variants of this data structure, and the discriminated union of sub-data structures contains fields common to each sub-type. Object templates are defined in ncx/obj.h.

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.

2.2.3 Data Tree



A YANG data tree represents the instances 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.

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.

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

2.2.4 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.

- **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.
- **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 Server Operation

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

2.3.1 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
- 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.
  - Core YANG modules are loaded
- The agt_init2 function is called:
  - Initialize the core server code modules
    - agt_cb_init()
    - agt_signal_init()
    - agt_rpc_init()
    - agt_connect_init()
    - agt_hello_init()
    - agt_caps_set_caps()
    - agt_ncx_init()
    - agt_acm_init()
    - agt_ses_init()
    - agt_sys_init()
    - agt_state_init()
    - agt_not_init()
    - agt_proc_init()
    - agt_if_init()
  - Any 'load' commands found in the CLI or server configuration file are processed
  - agt_cap_set_modules()
  - load_running_config()
  - Initialize non-config (read-only) data structures in the <running>

2.3.2 Start a Session

2.3.3 Process an Incoming Request

2.3.4 Edit the Database

2.3.5 Save the Database

2.3.6 Send a Notification

2.3.7 End a Session

2.4 SIL External Interface

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

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

2.4.1 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
- 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 (may be NULL)

Returns:

- operation status (0 if success)

Example H file declaration:

```
extern status_t
    y_toaster_init (
        const xmlChar *modname,
        const xmlChar *revision);
```

2.4.2 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. 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)

Name Format:

y_<modname>_init2

Returns:

- operation status (0 if success)

Example H file declaration:

```
extern status_t
    y_toaster_init2 (void);
```

2.4.3 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

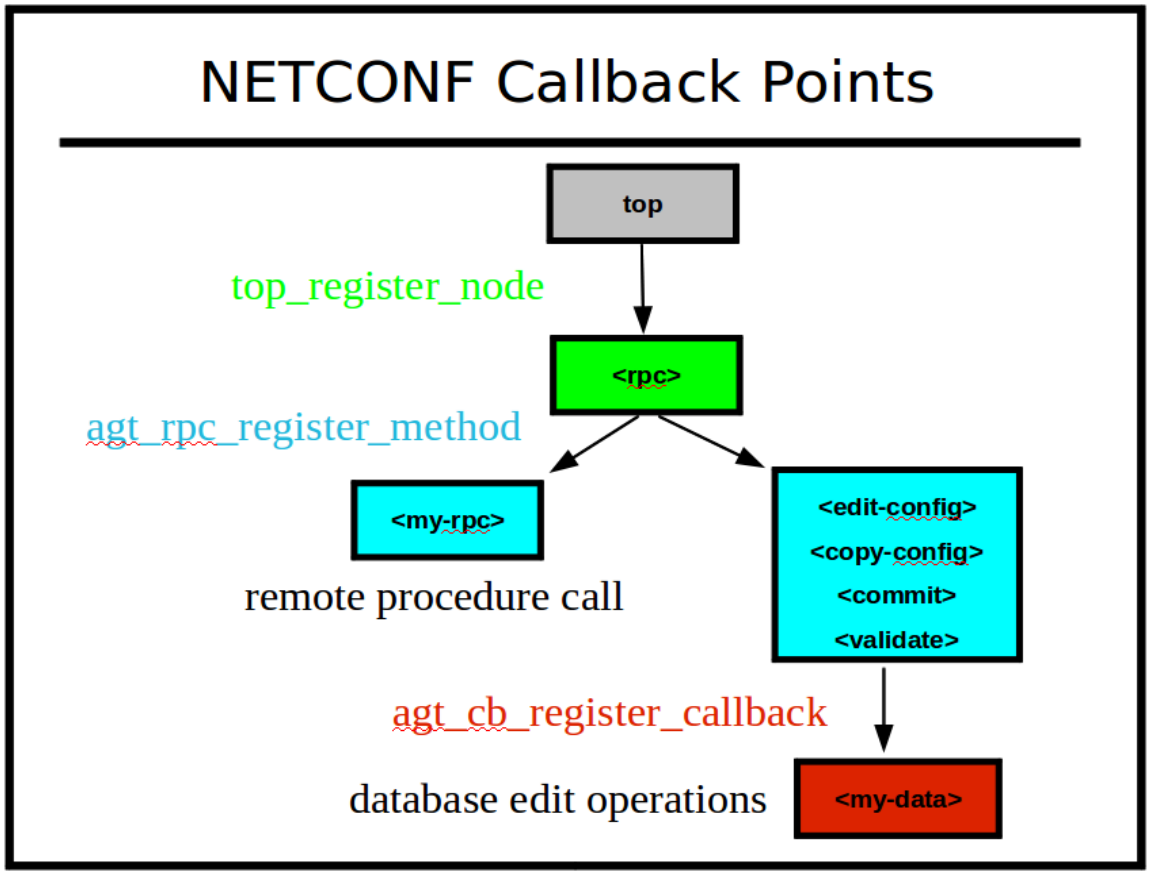
Name Format:

y_<modname>_cleanup

Example H file declaration:

```
extern void
    y_toaster_cleanup (void);
```

2.5 Server Callback Functions



2.5.1 RPC Operations

2.5.2 Database Operations

2.5.3 YANG Extension Data

2.6 Main Data Structures

2.6.1 ncx_module_t

2.6.2 obj_template_t

2.6.3 val_value_t

2.6.4 ses_cb_t

2.6.5 rpc_msg_t

2.6.6 xml_node_t

2.6.7 status_t

2.7 Toaster Tutorial

3 Development Environment

3.1 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**
  - 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**
  - 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 (usually /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 <hello> PDU. If

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

- **libtecla**
  - command line library provided with Yuma Tools; used by **yangcli**
- **ncurses**
  - character processing library needed by libtecla; used by **yangcli**
- **libc (or glibc)**
  - unix system library
- **libssh2**
  - SSH2 client library, used by yangcli
- **libxml2**
  - xmlTextReader XML parser
  - pattern support

3.2 Yuma Source Tree Layout

3.3 Platform Profile

3.4 SIL Makefile

3.4.1 Build Targets

3.4.2 Command Line Build Options

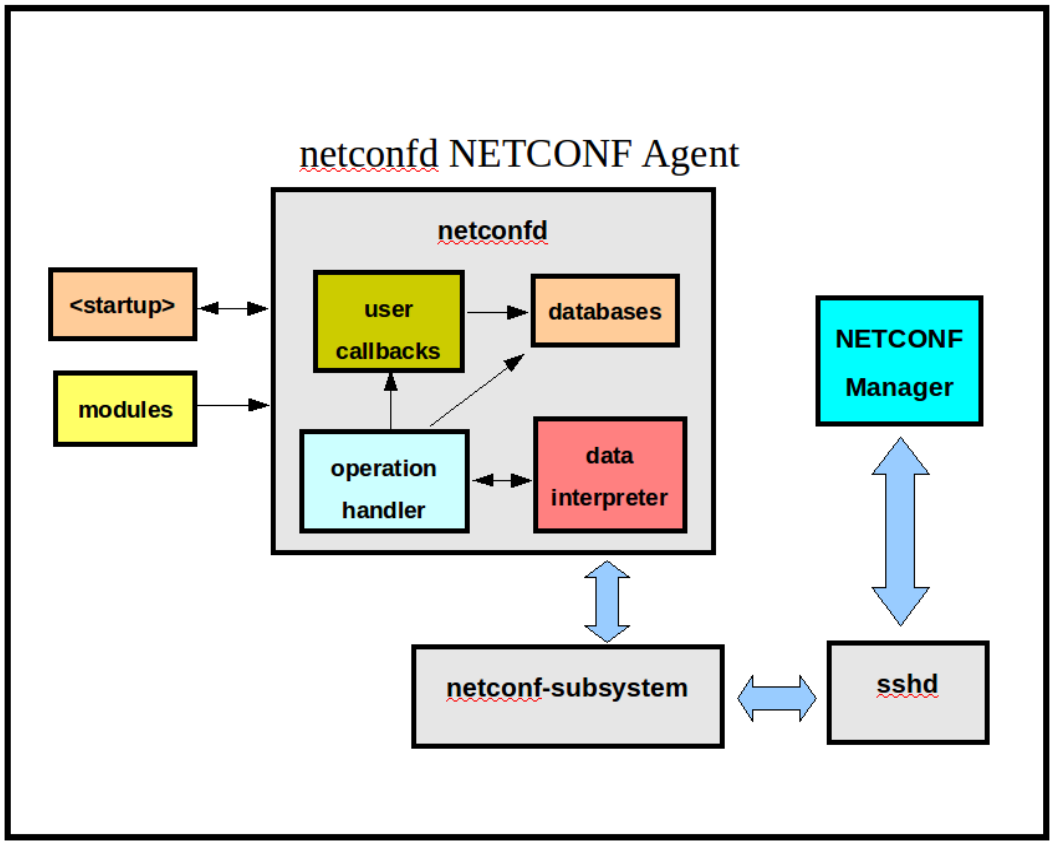
DEBUG

STATIC

MEMTRACE

MAC

4 Runtime Environment



4.1 Memory Management

4.2 Capability Management

4.3 Session Management

4.4 Database Management

4.5 Network Input and Output

4.6 Logging and Debugging

4.7 XML

4.8 XPath

4.9 YANG Data Structures

4.10 NETCONF Operations

4.11 RPC Reply Generation

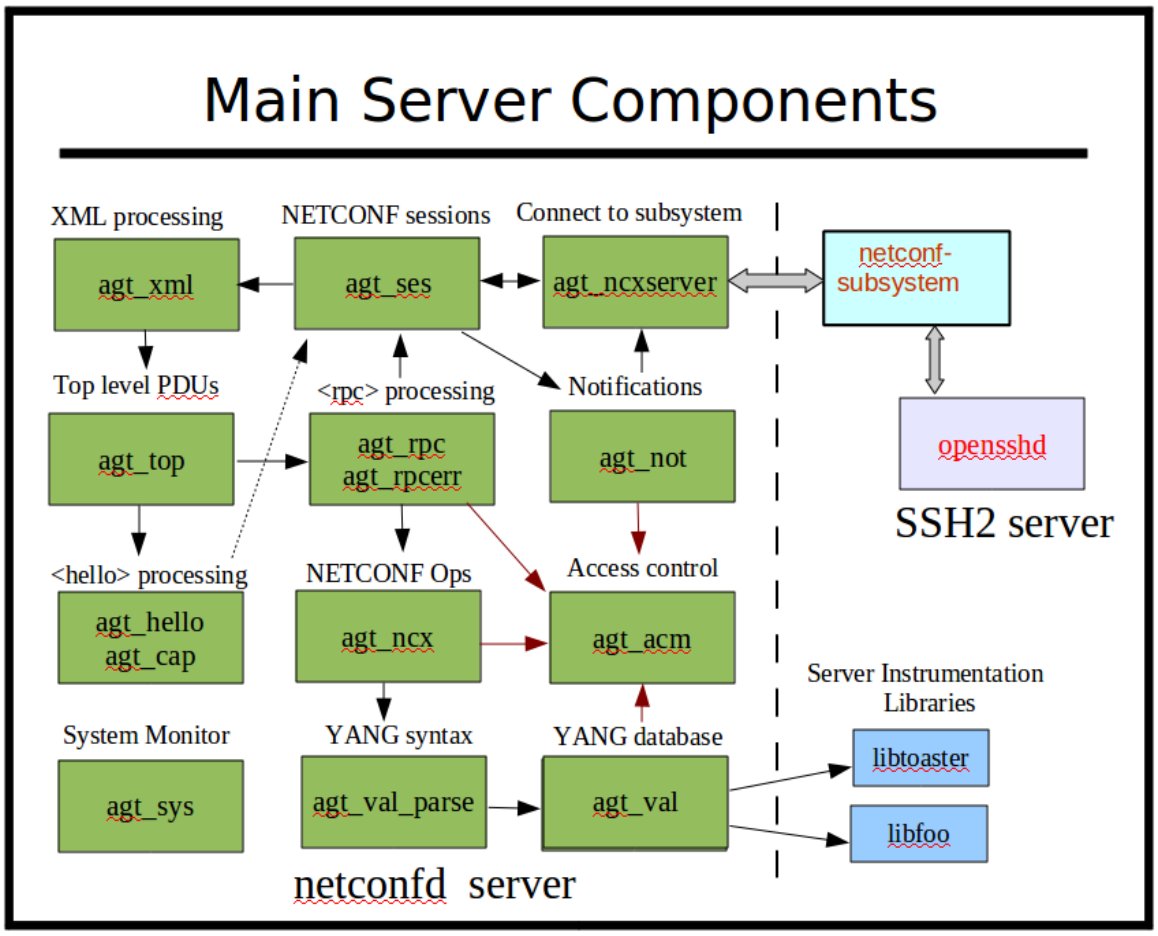
4.12 RPC Error Reporting

4.13 Notifications

4.14 Retrieval Filtering

4.15 Access Control

4.16 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.

4.17 Automation Control

The YANG language includes many ways to specify conditions for database validity, which traditionally are only documented in DESCRIPTION clauses:

4.17.1 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**.

4.17.2 NCX:CLI EXTENSION

The **ncx:cli** extension is used in in YANG container definitions, which represent the program CLI parameters, not NETCONF database parameters. It does not take any parameters, and is defined in **ncx.yang**.

```
container yangcli {
    ncx:cli;

    // all the yangcli CLI parameters
}
```

If this extension is present, then **netconfd** will ignore the container when loading the database object definitions. Only the program with the same name as the container will use the CLI parameter definition.

4.17.3 NCX:DEFAULT-PARM EXTENSION

The **ncx:default-parm** extension is used within a container with an **ncx:cli** extension, or within an 'input' section of an RPC operation definition. It is defined in **ncx.yang**.

If no parameter name is found when processing CLI parameter input, and the **ncx:default-parm** extension is present in the container or RPC input being processed, then the specified parameter name will be used instead of generating an error. The value must be valid for the parameter syntax, according to its YANG definition. This means that for the default parameter, only the <value> component of the complete parameter syntax may be used, as well as the normal forms.

```
container yangdump {
    ncx:cli;
    ncx:default-parm module;

    // all the yangdump CLI parameters
}
```

When invoking the **yangdump** program, the default CLI parameter is **--module**. These two command lines are equivalent:

```
yangdump –-module=test1 –-module=test2
yangdump test1 test2
```

A string that does not start with any dashes will still be tried as a parameter name, before trying the default parameter. If the value used for a default parameter conflicts with another parameter name, then the normal form must be used, instead of this form.

```
yangdump log-app  test1
```

Even if there was a module named 'log-app', it would not be tried as a **--module** parameter, since it also matches the **--log-append** parameter.,

5 Software Components

5.1 ncx

5.1.1 b64.c

5.1.2 blob.c

5.1.3 bobhash.c

5.1.4 cap.c

5.1.5 cfg.c

5.1.6 cli.c

5.1.7 conf.c

5.1.8 def_reg.c

5.1.9 dlq.c

5.1.10 ext.c

5.1.11 GRP.C

5.1.12 HELP.C

5.1.13 LOG.C

5.1.14 NCX.C

5.1.15 NCXMOD.C

5.1.16 OBJ.C

5.1.17 OBJ_HELP.C

5.1.18 OP.C

5.1.19 RPC.C

5.1.20 RPC_ERR.C

5.1.21 RUNSTACK.C

5.1.22 SEND_BUFF.C

5.1.23 SES.C

5.1.24 SES_MSG.C

5.1.25 STATUS.C

5.1.26 TK.C

5.1.27 TOP.C

5.1.28 TSTAMP.C

5.1.29 TYP.C

5.1.30 VAL.C

5.1.31 VAL_UTIL.C

5.1.32 VAR.C

5.1.33 XML_MSG.C

5.1.34 XMLNS.C

5.1.35 XML_UTIL.C

5.1.36 xml_val.c

5.1.37 xml_wr.c

5.1.38 xpath1.c

5.1.39 xpath.c

5.1.40 xpath_wr.c

5.1.41 xpath_yang.c

5.1.42 yang.c

5.1.43 yang_ext.c

5.1.44 yang_grp.c

5.1.45 yang_obj.c

5.1.46 yang_parse.c

5.1.47 yang_typ.c

5.2 agt

5.2.1 AGT_ACM.C

5.2.2 AGT.C

5.2.3 AGT_CAP.C

5.2.4 AGT_CB.C

5.2.5 AGT_CLI.C

5.2.6 AGT_CONNECT.C

5.2.7 AGT_EXPR.C

5.2.8 AGT_HELLO.C

5.2.9 AGT_IF.C

5.2.10 AGT_NCX.C

5.2.11 AGT_NCXSERVER.C

5.2.12 AGT_NOT.C

5.2.13 AGT_PROC.C

5.2.14 AGT_RPC.C

5.2.15 AGT_RPCERR.C

5.2.16 AGT_SES.C

5.2.17 AGT_SIGNAL.C

5.2.18 AGT_STATE.C

5.2.19 AGT_SYS.C

5.2.20 AGT_TIMER.C

5.2.21 AGT_TOP.C

5.2.22 AGT_TREE.C

5.2.23 AGT_UTIL.C

5.2.24 AGT_VAL.C

5.2.25 AGT_VAL_PARSE.C

5.2.26 AGT_XML.C

5.2.27 AGT_XPATH.C

5.3 mgr

5.3.1 MGR.C

5.3.2 MGR_CAP.C

5.3.3 MGR_HELLO.C

5.3.4 MGR_IO.C

5.3.5 MGR_NOT.C

5.3.6 MGR_RPC.C

5.3.7 MGR_SES.C

5.3.8 MGR_SIGNAL.C

5.3.9 MGR_TOP.C

5.3.10 MGR_VAL_PARSE.C

5.3.11 MGR_XML.C

5.4 platform

5.4.1 CURVERSION.H

5.4.2 PLATFORM.PROFILE

5.4.3 PLATFORM.PROFILE.DEPEND

5.4.4 PLATFORM.PROFILE.SIL

5.4.5 PROCDEFS.H

5.4.6 SETVERSION.SH

5.5 subsys

5.5.1 NETCONF-SUBSYSTEM.C

5.6 netconfd

5.6.1 NETCONFD.C

5.7 yangcli

5.7.1 YANGCLI_AUTOLOAD.C

5.7.2 YANGCLI_AUTOLOCK.C

5.7.3 YANGCLI.C

5.7.4 YANGCLI_CMD.C

5.7.5 YANGCLI_LIST.C

5.7.6 YANGCLI_SAVE.C

5.7.7 YANGCLI_SHOW.C

5.7.8 YANGCLI_TAB.C

5.7.9 YANGCLI_UTIL.C

5.8 yangdump

5.8.1 c.c

5.8.2 c_util.c

5.8.3 cyang.c

5.8.4 h.c

5.8.5 html.c

5.8.6 sql.c

5.8.7 xsd.c

5.8.8 xsd_typ.c

5.8.9 xsd_util.c

5.8.10 xsd_yang.c

5.8.11 yangdump.c

5.8.12 yangdump_util.c

5.9 yangdiff

5.9.1 YANGDIFF.C

5.9.2 YANGDIFF_GRP.C

5.9.3 YANGDIFF_OBJ.C

5.9.4 YANGDIFF_TYP.C

5.9.5 YANGDIFF_UTIL.C