User Guide

Ayla Portable Device Agent Porting Guide

Version: 1.0

Date Released: September 27, 2018

Document Number: AY006UPD3-1

Ayla Networks

Copyright Statement

Trademarks Statement

Referenced Documents

Ayla Networks does not supply all documents that are referenced in this document with the equipment. Ayla Networks reserves the right to decide which documents are supplied with products and services.

Contact Information

Ayla Networks TECHNICAL SUPPORT and SALES

Contact Technical Support: https://support.aylanetworks.com
or via email at support@aylanetworks.com

Contact Sales: https://www.aylanetworks.com/company/contact-us

Ayla Networks REGIONAL OFFICES

GREATER CHINA	HEADQUARTERS	EUROPE
Shenzhen	Silicon Valley	Munich
Room 310-311	4250 Burton Drive, Suite 100	Building 64.07
City University of Hong Kong	Santa Clara, CA 95054	Room EG.A.076 / 14b
Research Institute Building	United States	Rupert-Mayer-Str. 44
No. 8 Yuexing 1st Road	Phone: +1 408 830 9844	D-81379 München
High-Tech Industrial Park	Fax: +1 408 716 2621	Germany
Nanshan District		
Shenzhen, China	**JAPAN**	**TAIWAN**
Phone: 0755-86581520	Room #701	Taipei
	No. 2 Ueno Building 3-7-18	5F No. 250 Sec. 1
	Shin-Yokohama, Kohoku Ward	Neihu Road, Neihu District
	Yokohama City, 222-0033 Japan	Taipei 11493, Taiwan
	Phone: 045-594-8406	

For a Complete Contact List of Our Offices in the US, China, Europe, Taiwan, and Japan:
https://www.aylanetworks.com/company/contact-us

Table of Contents

1 Introduction

The Portable Device Agent (PDA) gives manufacturers the option to select any cellular or Wi-Fi module and have it connected easily to the Ayla Cloud. Ayla customers no longer have to use hardware supported by Ayla, and they have more control over development and deployment schedules.

PDA is an Ayla Integrated Agent (white box) design with all platform-dependent functionalities encapsulated in an adapter layer. Like the Ayla Integrated Agent, the PDA design is a lower cost option than the Ayla Production Agent (black box) design; however, PDA additionally removes the complexity involved in porting the Ayla Integrated Agent. Ayla customers using PDA simply port the adapter layer to the new platform. PDA offers Ayla customers the flexibility to enable Ayla connectivity on their hardware of choice and per their own schedule, while maintaining the high standards of Ayla's reliability and security.

1.1 Audience

This document is intended for Ayla partners who want to port white box device agents to previously unsupported platforms, while maintaining connectivity to the Ayla Device Service (ADS).

1.2 Related Documentation

Refer to the following documents available on support.aylanetworks.com for additional information on the Ayla Developer Portal and OEM Dashboard.

- *Ayla Linux Agent Setup* (AY006ULA6)

- *Ayla Customer Dashboard User's Guide* (AY006UDB3)

- *Ayla Developer Portal User's Guide* (AY006UDP3)

- *Ayla Embedded Agent for Marvell WMSDK* (AY006DAM6)

1.3 Document Conventions

This document uses these Ayla documentation conventions:

- Function prototypes, function names, variables, structure names and members, and other code fragments are shown in `courier new`, a fixed-width font.

- Network paths, file paths, menu paths and the like are shown in **bold** text and each point that you have to click to navigate to the next is separated by "`/`."

1.4 Abbreviations and Acronyms

The following acronyms are used in this document.

ADA	Ayla Device Agent. This is a legacy term for Ayla Embedded Agent. The ADA acronym is still used in CLI commands and sources, and descriptions of either of these may refer to the Ayla Embedded Agent as the "device agent."
ADS	Ayla Device Service (the cloud service)
ADW	Ayla Device Wi-Fi (library for embedded systems)
AL	Adapter Layer. A layer that encapsulates all the platform-dependent code.
ANS	Ayla Notification Service
DNS	Domain Name Server
EVB	Evaluation Board
MCU	Microcontroller unit
PDA	Portable Device Agent.
PWB	Portable White Box device, which we call PDA.
RTC	Real-time Clock
RTOS	Real-time operating system
SDK	Software Developer Kit
TCP	Transmission Control Protocol
TLS	Transport Layer Security

1.6 Glossary

Production Agent (formerly called Black Box)	This is a fully-managed, Ayla-enabled module intended to be used as-is by the manufacturer. Some of the primary characteristics include: - Available for embedded solutions. - Provides the fastest time to market for OEMs - No custom gateway or other forms of communication agent software, including QA required regardless of the type of end-device. - Any microcontroller-based system can easily be enabled with cloud connectivity.

Integrated Agent (formerly called White Box)

This is a type of Ayla endpoint that allows for a more complex and versatile device than the Production Agent class of devices. However, the development effort is often significantly longer for OEMs and therefore results in longer time to market than the Ayla Wi-Fi Production Modules. Some of the primary characteristics include:

- Available for embedded or LINUX solutions.
- Makes the Ayla Embedded Agent available as a library or source.
- Well-equipped for applications with existing RTOS and networking.
- Because of the modular design, allows code for additional functions to be included as needed.
- Allows for a reduced bill of material (BOM) cost in certain situations.

2 PDA Software Architecture

In the PDA design, an adapter layer (AL) is constructed and all platform-dependent functionalities are encapsulated in this layer. AL interfaces are defined in `al_xxx.h`.

IMPORTANT!	The implementation of the AL for the Linux platform is only for demo purposes. If your new platform is Linux, the demo cannot be used for production. For this platform, the [Ayla Linux Agent](#) is the best option, especially in terms of its robustness.

This section provides information on how the PDA Software Architecture works.

2.1 PDA Architecture

Following is a basic diagram of the PDA's layered architecture:
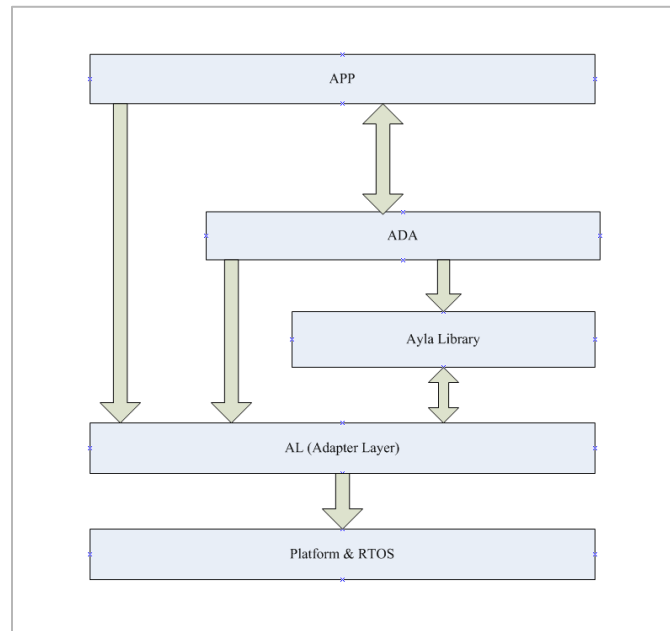
APP	A customer application.
ADA	Ayla Device Agent, which is a legacy term for Ayla Embedded Agent. It is available as a library or source, and provides secure connectivity to Ayla services.
Ayla Library	This is a library of utilities, which is platform independent.
AL	The Adapter Layer, which provides the interfaces that encapsulate platform-dependent code.
RTOS	Platform-dependent implementation, including device drivers, and Real-time operating system.

In the PDA layered architecture (as shown in the above diagram), the customer application interacts with the Ayla Embedded Agent, and may call some AL interfaces, like memory, thread, and mutex lock. The implementation of the Ayla Embedded Agent is based on the AL and Ayla library. Basically, the AL is the base of the customer application, Ayla Embedded Agent, and Ayla Library.

PDA has two main differences from the Ayla white box design, which are described in this section:

1. A new thread model
2. An adapter layer

2.2 PDA Thread Model

There are two threads in the PDA:

1. The application thread, for example, LEDEVB in sample demo.
2. The Ayla Embedded Agent (referred to as ADA) thread.

A main-loop is used in the ADA thread, and all operations are done in the main-loop. When an application sends a request to the Ayla Embedded Agent, the request is placed in a queue in the order of the request's priority, and the ADA thread is intentionally woken up. When the ADA main-loop runs, each request in the queue is checked according the request's priority. If a request is found, it is processed and then its callback is called to report the result.

In the PDA sample code, the default example of main-loop I/O processing does the following:

- Uses `select () statement` to check the socket for RX / TX / Exception signals
- If a signal is detected, then the process calls the socket RX-callback for receiving data, tx-callback for sending data, err-callback for exception or error handling.

In the ADA main-loop, the `select () statement` is used for both the socket I/O as well as for the timer handling and ADA thread wakeup. The select statement detects all of the file, socket, and pipe descriptors that are registered. When waking up the ADA thread, a special pipe descriptor is used. When writing data to the pipe in another thread, the select statement in the ADA main-loop recovers from the blocking state; that is, the ADA thread is woken up by the other thread.

For the timer, no special descriptors are required. Timer scheduling is just finding the next timer that will timeout, calculating the time span, and using the time span as the last the parameter of the `select ()` function. For events in which the socket or pipe is not signaled, the select statement will be recovered from blocking state in the specified time (automatically woken up by the ADA timer).

2.3 Adapter Layer

The Adapter Layer (also referred to as AL) is a new layer added for the PDA. All lower-level interfaces that are platform dependent are encapsulated in this layer. You may think of the AL as you would a Hardware Abstract Layer (HAL). All interfaces are listed in the AL by category. For example, the default implementation is based on Linux. Ayla's chip vendors and cooperative partners who develop a device based on PDA just basically have to port the AL of the PDA to the desired platform.

AL contains the following main components; all of which are explained in this section:

- Memory Manager

- Thread and Mutex Lock

- ADA Thread

- Netstream

- NetUDP

Memory Manager

The Memory Manager provides functionalities of memory allocation, memory free, and memory pool initialization. In the AL, there are two types of memory:

1. `al_os_mem_type_long_period`

   This is memory that is seldom free after allocation.

2. `al_os_mem_type_long_cache`

   This is memory that is frequently allocated and free.

Thread and Mutex Lock

The thread interfaces in the AL includes: thread creation, suspend, resume, sleep, get exit-flag, set exit-code, terminate synchronously, terminate asynchronously, and join (wait for thread terminated). To create a thread, the thread name, stack address and size, priority, and entry should be specified.

A running thread should call `al_os_thread_get_exit_flag()` constantly in its loop. If the exit-flag is detected, the thread should leave the loop and terminate gracefully.

A thread can be terminated by another thread or itself. This can be achieved by calling `al_os_thread_termiate()`. This function just sets the exit flag for the thread and lets the thread terminate itself asynchronously.

If thread A wants to terminate thread B, thread A needs to call `al_os_thread_terminate_with_status()` for thread B. This function sets an exit flag for thread B, joins and waits for thread B to be terminated, and returns the exit-code of thread B. This function cannot be used for the current thread to terminate itself.

If thread A wants to block itself until thread B is terminated, thread A just needs to call `al_os_thread_join()` for thread B.

After a thread is created, the default memory allocation type for the thread is `al_os_mem_type_long_period`. This can be changed by calling `al_os_mem_set_type()`.

In the AL, mutex is called lock. It is used to keep thread safe when multiple threads access the same data. The interfaces include: lock create, lock, unlock, destroy.

ADA Thread

ADA thread is an Ayla specialized thread. It contains a loop that is used for communication, timer, and wakeup (that is, wakes itself up). ADA thread interfaces are defined in `al_ada_thread.h`. The `al_ada_main_loop()` function is defined in `al_ada_thread.c`.

Interaction between the customer application (APP) thread and ADA thread can be performed by asynchronous calls or synchronous calls, as follows:

- To wakeup the ADA thread from another thread, use call `al_ada_wakeup()`.

- To send an asynchronous request to ADA: First, the APP should initialize a callback structure using `al_ada_callback_init()`, and then, use call `al_ada_call()` with the callback structure as a parameter. The request is executed in ADA thread. After execution, a callback is called to report the result. It is APP's responsibility to keep the callback being thread safe.

- To send a synchronous request to ADA: First, initialize a callback structure using `al_ada_callback_init()`, and then call `al_ada_sync_call()` with the callback structure as a parameter. The request is put into the ADA request queue, and then wakes up ADA thread and waits for a semaphore. When ADA thread is activated, the request is executed, the result is reported to APP through callback. Finally the semaphore is signaled by the ADA, so ADA thread is suspended and the APP is activated and continues to run. In this case, the callback's thread-safety is guaranteed by the synchronous calling mechanism.

- Use ADA timer as follows: To start a timer, call `al_ada_timer_set()`. To stop a timer, call `al_ada_timer_cancel()`.

NOTE   The ADA timer is for ADA internal use only. Calling `al_ada_timer_set()` in the APP thread starts a timer in ADA, not a timer in APP. Calling `al_ada_timer_set()`, `al_ada_timer_cancel()` in the APP thread is not thread safe.

Netstream

Netstream is used by the http-client in ADA. The device uses Netstream to communicate with Ayla Cloud service. There are 2 types of Netstream, stream of TCP and stream of TLS. The implementation of Netstream is based on Net TCP and Net TLS. For TCP stream, communication is reliable, but not secured. If TLS stream is used, a handshake is made, and a secure link is set up. To use Netstream, follow these steps:

1. Call `al_net_stream_new()` to create a specified type of stream. A handle is returned.

2. To set callbacks, call `al_net_stream_set_recv_cb()`, `al_net_stream_set_sent_cb()`, and `al_net_stream_set_err_cb()`.

3. To set the argument (which is used in callbacks), call `al_net_stream_set_arg()`. Generally, the argument is the caller itself.

4. To connect to remote, call `al_net_stream_connect()`. For stream TLS, a handshake is done at this stage, and the socket handle is also registered to the ADA main loop.

5. To send data, call `al_net_stream_write()`. The data is buffered in Netstream until `al_net_stream_output()` is called. After the data is actually sent, `sent-callback` is called to report how many bytes have been sent.

6. When the RX signal is detected in the ADA main loop, the Netstream raw receiving function is called to receive data. Then, the caller's `recv-callback` is called to report the received data, and the caller should call `al_net_stream_recved()` to tell Netstream how many bytes were received.

7. After communication is done, call `al_net_stream_close()` to close Netstream.

   Netstream is used for time-limited communication. After data is exchanged, Netstream closes. For TLS-stream type, if a connection is in an idle status for 60 seconds, the server also disconnects.

NetUDP

NetUDP is another communication interface used by the Ayla Embedded Agent. The Ayla Notification Service (ANS) in the Ayla cloud sends the UDP packet to devices to inquire about their online status. Devices send a UDP packet to report their status to the server. When a property of a device is changed in the Ayla Device Service (ADS), ADS lets ANS send a notification (UDP packet) to the device, and the device then performs a synchronization through Netstream. NetUDP is used to transmit events or notifications.

The process to use NetUDP is as follows:

1. ADA calls `al_net_udp_new()` to create a net-udp object.

2. To receive a UDP packet correctly, it is required to bind the socket to a local address and port. To do this, just call `al_net_udp_bind()`.

3. To set the remote address and port, call `al_net_udp_connect()`. At this stage, the socket is registered to the ADA main-loop.

4. To set recv-callback and argument, call `al_net_udp_set_recv_cb()`.

5. To send the UDP packet to remote, call `al_net_udp_send()`. The packet is buffered in NetUDP and is sent to remote when the TX signal is detected in the ADA main loop. The UDP packet is then allocated by the ADA, and freed by NetUDP when the packet is actually sent.

6. When the RX signal is detected in the ADA main loop, `al_net_udp_raw_recv()` is called, a UDP packet is received, and the recv-callback is called to report to the upper layer. The UDP packet is then allocated in NetUDP, and freed by the upper layer.

7. If ADA wants to send a multicast packet, call `al_net_igmp_joingroup()` to join a UDP multicast group before sending. If multicast is not used any more, call `al_net_igmp_leavegroup()` to leave the multicast group.

8. To close the NetUDP object, call `al_net_udp_free()`.

2.4 Adapter Layer (AL) Interface List

The table in this section provides a complete list of the interfaces in the adapter layer. For detailed information, please read the related header files and the Doxygen document.

Interface	Header File	Implementation
Memory	al_os_mem.h	al_os_mem.c
Thread	al_os_thread.h	al_os_thread.c
Lock (Mutex)	al_os_lock.h	al_os_lock.c
ADA thread	Al_ada_thread.h	Al_ada_thread.c
Clock	Al_clock.h	Al_clock.c
Netstream	al_net_stream.h	al_net_stream.c
Net TCP	al_net_tcp.h	al_net_tcp.c
Net TLS	al_net_tls.h	al_net_tls.c
Net-Interface	al_net_if.h	al_net_if.c
Net-DNS	al_net_dns.h	al_net_dns.c
Net-Address	al_net_addr.h	al_net_addr.c
Net-UDP	al_net_udp.h	al_net_udp.c
Random	al_random.h	al_random.c
AES	al_aes.h	al_aes.c
RSA	al_rsa.h	al_rsa.c
SHA1	al_hash_sha1.h	al_hash_sha1.c
SHA256	al_hash_sha256.h	Al_hash_sha256.c
CLI	al_cli.h	al_ali.c
Persist	al_persist.h	al_persist.c
OS Reboot	al_os_reboot.h	al_os_reboot.c

2.5 Source Code Organization

```
pda
├── ayla
│   ├── doc                                    Document files
│   ├── include                                Header file for App and platform
│   │   ├── ada
│   │   └── al
│   ├── make
│   │   ├── common_cc.mk
│   │   └── common_defs.mk
│   ├── src
│   │   ├── ext
│   │   │   └── jsmn
│   │   ├── include                            Ayla internal header files
│   │   │   ├── ada                            Internal header files of ADA
│   │   │   └── ayla                           Internal header files of Ayla library
│   │   ├── libada
│   │   │   ├── ada_version.mk
│   │   │   ├── certs                          Certificate files
│   │   │   └── Makefile                       Sub-makefile of ADA
│   │   └── libayla                            Internal C files of Ayla library
│   │       └── Makefile                       Sub-makefile of Ayla library
│   └── test
│       ├── altest                             C files of test-framework
│       │   ├── Makefile                       Sub-makefile of test-framework
│       │   └── testcases                      C files of test-cases
│       ├── altest-server                      Test server for test-framework
│       │   ├── altest-server.py               Python script to start test server
│       │   └── certs                          Certificate files of altest-server
│       └── apptest                            Application test to test ADA APIs
│           ├── automation_apptest_script      The automation script for apptest
│           │   ├── py                         The python scripts of automation
│           │   │   ├── ada
│           │   │   └── lib
│           │   ├── rc.sh                      Environment setup script for apptest
│           │   └── util                       Utils for automation script
│           ├── Makefile                       Sub-makefile of apptest
│           └── src                            C files of apptest
├── examples
│   ├── include
│   │   └── demo                               Header files of ledevb demo
│   └── ledevb                                 C files of ledevb demo
│       └── Makefile                           Sub-makefile of ledevb demo
├── Makefile                                   Makefile of PDA
└── platform
    ├── include
    │   └── platform
    ├── linux                                  AL C files for Linux platform
    │   ├── include                            AL Header files for Linux platform
    │   │   ├── al
    │   │   │   └── al_compiler.h               Header files for compiler portable
    │   │   └── platform
    │   ├── Makefile                           Sub-makefile of AL
    │   └── utils
    │       └── conf-gen.py
    └── stubs                                  AL API dummy files for Linux platform
        ├── main.c                             C main file for stubs
        └── Makefile                           Sub-makefile for stubs
```

3 Build and Run the Demo

1. Install the compiler on your developing host system.

2. Obtain the PDA software package (pda-demo-src-2.1-eng.tgz) from Ayla.

NOTE Once the Software License Agreement is signed, this is available on the Ayla Support Portal. Contact your Ayla Customer Success representative if you need assistance finding this.

3. Decompress the package in a directory (name it PDA, for example).

4. In the PDA directory, run the `make` command.

   - To build and run the altest program, input the following command:

     ```
     make run EXEC=altest
     ```

   - To build and run the ledevb program, input the following command:

     ```
     make run
     ```

5. After compilation, make sure you have the following output file(s):

   - pda/ayla/bin/native/ledevb

   - pda/ayla/bin/native/altest

6. Run the test program (altest).

   When the `PWB ALTest>` prompt appears, input the `altest` command to list all test cases.

7. Run `LEDEVB` demo.

IMPORTANT! Before running demo LEDEVB (a Linux program, a virtual device), you need to configure a DSN (device serial number) and a key for the device. There is a python script (`pda/platform/linux/utils/conf-gen.py`) that can be used to generate a default configure file (`.conf.pwb`) in your home directory. To do this, run the following command on Linux command line:

```
./conf-gen.py --region CN  AC000W002397415.xml  0dfc7900  ledevb \
          --oem-key xxxxxxxxxxxxxxxxxxxxxxxx
```

   - CN is the area code of China.
   - AC000W002397415 is the DSN.
   - 0dfc7900 is OEM-ID string.
   - ledevb is oem_model.

- After running demo LEVEVB for the first time, you need to register the device to the Ayla Cloud Service. After that, you can change the property of the device. For detailed information, please refer to _Ayla Embedded Agent for Marvell WMSDK_ (AY006DAM6).

When the demo ledevb runs, the `PWB ledevb>` prompt is shown in the console. In the actual device, the Ayla Embedded Agent starts to work (or enter the Up state) only when the device

becomes a node of a network. But for PDA, there is no Wi-Fi module or no connection manager. PDA is just a virtual device. You need to input the `up` command in console to start ADA file.

4 Porting the PDA

Porting programs to another platform can be challenging work. The process is related to target compiler, C library and platform drivers provided by the vendor. Some common challenges are:

- Compilers may not support 64-bit data type or operation. The enum type in C language may take a different size for a different compiler.

- Issues with the structure alignment and packing. Communication programs are sensitive to platform byte order (little endian or big endian), so using union types is not recommended.

Though it is useful to be aware of issues like these, the primary focus of this section is on PDA porting.

IMPORTANT!	If required, the file called `al_utypes.h` may need some modification. But all other header files that take form of `al_xxx.h` should not be modified. These header files are Ayla-defined interfaces for the Adapter Layer. Only `al_xxx.c` files need to be re-implemented on new platform.

4.1 Porting Tactic

When you start porting, you need to modify the `make` file. All the `al_xxx.c` files referred to in `make` file should be changed to files with the same name, but must be located in platform/stubs directory. This is because in the initial porting stage, all of the C files for the adapter layer should use stub files.

In the PDA demo packet, all `al_xxx.c` files are located in the platform/Linux directory. If, for example, your target platform is `stm32,` you need to create a new directory called stm32 in the same directory as the platform. Once this is done, follow these steps (which are based on this same example):

1. Put your c file (for example, `al_aes.c`) in the platform/stm32 directory.

2. Modify the `make` file to refer to the new file.

3. Add a reference in the make file to refer to `al_test_aes.c` in the test directory.

4. Build `altest` by running the following command:

   ```
   make run EXEC=altest
   ```

5. Test your new implementation, which in this example is an implementation of AES. For this example, let's assume that the altest program built in step 4 above is in the Ayla/bin/smt32/ directory. Run the program, and in the console of the device, type the test command:

   ```
   altest  aes
   ```

4.2 Porting Memory Manager

Following is important information regarding the Porting Memory Manager:

- Support for memory allocation is free.

- Memory interfaces are called by the customer application, Ayla Embedded Agent, and Ayla Library. These interfaces should be thread-safe.

- There are two types of memory and therefore two memory pools.

- Function `al_os_mem_set_type()` is related to the thread. Calling this function sets memory type for the current thread. After this call, all memory allocated in the thread are of this same memory-type.

4.3 Porting Thread and Lock

The thread and mutex interfaces are always related to the operating system. Operating systems have different prototypes of task or mutex. The Ayla Black Box and White Box devices have been ported to Realtek Ameba (FreeRTOS), Marvell (FreeRTOS), and QCA4010 (ThreadX) platforms. Therefore, these are the preferred platforms. Additional information to keep in mind:

- Thread creation, sleep, set/get priority, join, terminate, terminate with exit code returned should be implemented.

- Thread get exit-flag and set exit-code should be implemented.

- Thread suspend and resume are optional.

- Using the operating system's mutex to implement lock in the adapter layer is sufficient.

4.4 Porting Clock

In the PDA, local time and the software timer are based on the clock. There two types of clocks provided in the adapter layer: One is UTC time (seconds elapsed from 1970, Jan, 1st 00:00), and the another is elapsed time from boot (in milliseconds)

The clock interfaces may be used by the customer application and the Ayla Embedded Agent. They should be thread-safe, whether based on the system clock or RTC.

When calling `al_clock_set()`, the clock-source should be specified. After setting the clock with a high priority clock-source, setting the clock with a lower priority clock-source has no effect. The clock-source in `al_clock.h` is defined by Ayla and should not be changed.

4.5 Porting Netstream, Net TCP, and Net TLS

Netstream is used by the http-client. There two types of streams: stream-tcp for HTTP and stream-tls for HTTPS. The sample implementation of Netstream is based on Net TCP and Net TLS.

Netstream itself is platform-independent and not required to port, but you can implement your own Netstream and keep the prototypes as-is. If re-implementing a new Netstream, you can ignore Ayla's sample codes of Net TCP and Net TLS. Other important considerations regarding Netstream, Net TCP, and Net TLS are as follows:

- Netstream, Net TCP, and Net TLS are not thread-safe; the design is for use with the ADA thread only. Do not call Netstream, Net TCP, and Net TLS in other threads. If you want to use Netstream in both the customer application and the Ayla Embedded Agent, you must make Netstream thread-safe.

- Do not use Net TCP and Net TLS directly. Use Netstream instead.

- In porting Net TCP, low-level functions, like `socket ()`, `connect ()`, `recv()`, `send()` may need to be ported. The `select ()` function used in the ADA main-loop may need porting as well.

NOTE   Make sure that the ADA main loop works in blocking mode.

- Net TCP is implemented in `pfm_net_tcp.c`. When the upper-layer calls `pfm_net_tcp_write()`, the data is not sent immediately, and instead is buffered by Net TCP until `select()` in the ADA main-loop detects the TX signal, then the callback `pfm_net_sync_write()` is called for the actual transmission. Net TCP uses `pfm_pwb_linux_add_socket ()` to register its socket handler, `pfm_net_sync_read`, `pfm_net_sync_writ` to the main-loop.

- The Net TCP sample is based on Linux. The two functions are empty. On embedded platform, these functions are required to handle communication error:

    o `pfm_net_tcp_abort()`

    o `pfm_net_tcp_abandon()`

- The Net TLS sample code is based on OpenSSL on Linux. To set up a security link, certificates need to be verified. In OpenSSL, certificates are files stored in /etc/ssl/certs/ directory. Before running the sample PDA demo on Linux, you need to copy certificates (*.pem) from the PDA directory ayla/src/libada/certs/ to the destination of /etc/ssl/certs/. Some examples are:

```
cd /etc/ssl

sudo mv certs  certs_old

sudo mkdir certs

sudo cp ~/pda/ayla/src/libada/certs/*.pem /etc/ssl/certs/

sudo c_rehash
```

On the embedded platform, there is no filesystem or OpenSSL, so it is required to load certificate into memory in the function `pfm_net_tls_init()`.

- When a Netstream of TLS stream type is created and `al_net_stream_connect()` is called, Net TLS makes a handshake with the remoted server. The handshake time depends on net

speed. After a successful handshake, a security link is set up. The security link is disconnected when it is in an idle state for 60 seconds.

4.6 Porting Interface of Random, SHA1, SHA256, AES, RSA

The default implementation of the random generator is pseudo-random. For porting `al_random.c` to the embedded platform, it is recommended to use the true random generator supported by the hardware.

The default implementation of Advanced Encryption Standard (AES) is based on OpenSSL. On the embedded platform, we suggest that hardware acceleration is used. Cipher block chaining (CBC) operation mode and the initialization vector (IV) are used in AES encryption and decryption. The encryptions and decryptions are strictly sequential operations, so AES encryption of the first step affects the encryption of the next step. This feature makes it thread-unsafe. Do not use an AES handle in two threads to encrypt/decrypt data concurrently.

RSA is an asymmetric crypto interface used in digital signature and verification. On an embedded platform, RSA's speed is relatively slow for software implementation. Hardware acceleration should be used. However, RSA with software implementation is thread-safe. So, when using hardware acceleration, make sure it is thread safe.

SHA1 and SHA256 are hash algorithms. Using software implementation on an embedded platform is sufficient for SHA1 and SHA256. The interface should be thread-safe.

The software implementations of Random, SHA1, SHA256, and RSA are thread-safe. If acceleration is used, it is your responsibility to make the acceleration thread-safe. AES with CBC mode supported is not thread-safe.

4.7 Porting net-addr, net-if, net-udp

In the default implementation of net-addr, the internet address is stored in a structure called `struct in_addr`. The address provides a convenient way for conversion to and from a 32-bit host address. Porting is not required.

The interface net-if is used for searching net-interfaces (like ether-net, Wi-Fi) in the system. When the device is powered on, the Wi-Fi interface is selected, and Ayla Embedded Agent tries working in STATION mode and connecting to Ayla Cloud Server. If this fails, AP mode is used for configuration and device registration. The net interface implementation varies with embedded platforms, so the interface must be ported and must be thread-safe.

The default implementation of net-udp is based on the Linux socket. The interface of net-udp defined in adapter layer is for use in the Ayla Embedded Agent only, the buffered packet-list is not protected by a mutex, so it is not thread-safe. Porting net-udp is just to replace `bind ()`, `sendto ()`, and `recvfrom ()` with corresponding functions defined in the platform library or RTOS.

4.8 Porting the ADA Thread

As mentioned in Section 4, there is a main-loop in the ADA thread, and all I/O and communications are performed in the main-loop, which uses `select ()`. On embedded systems, `select ()` should keep the same functionality.

In the default implementation of `al_ada_sync_call()`, a semaphore is used. The semaphore may need to be ported.

The implementation of the PDA sample demo is based on Linux. After building a PDA target (`LEDEVB`), PDA is an executable program on Linux. After the program is started, it can restart itself without `^C` and can be started again manually. For this purpose, `al_ada_kill()` is provided to kill the ADA thread on Linux. On the embedded platform, this function is not required and can be implemented as empty.

5 Test the Framework and Auto-Test

There is a test framework in the PDA source package. The test framework is used to test all of the Adapter Layer APIs manually or automatically. This test framework is implemented in the following file:

```
ayla/test/altest/pfm_test_frame.c
```

The main entry is `pfm_altest_main()`. This entry is registered to CLI in `pfm_test_frame_init()`. The implementation of all test cases is located in the following directory:

```
ayla/test/altest/testcases/
```

To build the target of `altest`, change the current directory to PDA, and input the following command:

```
make run EXEC=altest
```

For PDA on the Linux platform, the built program `altest` is in the `ayla\bin\native` directory.

For your target platform, you can change the output directory by modifying the path in the makefile.

After compilation is completed, run the `altest` program, then input the `altest` commands on the device console. You can test all of the Adapter Layer APIs, and after a test command is executed, the statistic information of test results is shown.

5.1 Using the altest Command

The test framework supports both auto test cases and manual test cases. For an auto test case, its order number must be larger than or equal to zero. For manual test case, its order number must be negative. All auto test cases can be automatically executed by the following command:

```
altest all
```

If you want to test a manual test case, the case name must be specified on the command line.

NOTE   Test variables must be set before running `altest all`. Refer to Section 5.3, Test Variables, for more details on this.

Following is how to use the `altest` command to test cases:

1. Get help on the command line by inputting the following command:

   ```
   altest ?
   ```

2. Show all of the test case names by inputting the following command:

   ```
   altest
   ```

3. To test a specified case (for example, aes-C610096), input the following command:

   ```
   altest aes-C610096
   ```

4. To test two specified cases, input the following command:

```
altest aes-C610096 aes-C610097
```

5. To test all cases, input the following command::

```
altest all
```

6. To test all cases and repeat the test 10 times, input the following command:

```
altest all --repeat 10
```

7. To test all cases and stop the test immediately when a case fails, input the following command:

```
altest all --stop_on_fail
```

5.2 Prepare altest-server

In the PDA source packet, there is a test-server written in python, which is in the following location:

```
/ayla/test/altest-server/altest-server.py
```

This is used to help net-udp and Netstream (TCP or TLS) testing. When altest program sends data to the server, the server simply sends the data back. The server-related packages and versions are as follows:

- Python 3.6.5
- Tornado 5.0.2

In the `/ayla/test/altest-server/certs/` directory, the following files are provided:

- altest.aylanetworks.com.chain --- Certificate chain
- altest.aylanetworks.com.key --- RSA private key of server
- altest.aylanetworks.com.pub.key --- RSA public key of server (format1)
- altest.aylanetworks.com.pub2.key --- RSA public key of server (format2)
- AylaQATestRoot.crt --- Root Certificate

These files are generated by Ayla and are used for altest only. The Certificate chain and the RSA private key are used by the server (`altest-server.py`). The two public keys are used for test RSA encryption, so it is not used for altest-server.

On device side (which is where altest located), to make TLS work, the following two actions should be done:

1. The root certificate file should be installed on the computer where altest runs. Generally, the certificate is stored in this directory:

```
/etc/ssl/certs/
```

2. In the `/etc/hosts` file, add this line:

```
172.18.88.2    altest.aylanetworks.com
```

Where 172.18.88.2 is the server's IP address, and altest.aylanetworks.com is the server's host name.

Proprietary   **18**

On the server side, to start `altest-server`, input this command:

```
python altest-server.py
```

5.3 Test Variables

There are test variables in some of the altest test cases The test variables have a default value, but need have another value set according to different running environments.

Before running `altest all`, test variables need to be set. This section provides information on the testvar commands and test variables.

testvar Commands

- To show how testvar is used, input the following command to get test variable help:

  ```
  testvar ?
  ```

- To show the name and value of the test variables, input the following command:

  ```
  testvar show
  ```

- To set a new value for the test variable, input the `testvar set` command as follows:

  ```
  testvar set <name> <value>
  ```

  Where `<name>` is the name of the test variable.

Test Variables

- dns_ads_ayla_com_ip

  The `dns_ads_ayla_com_ip` test variable is a list of DNS IPs of ads-dev.aylanetworks.com. This test variable is used to verify that the DNS Adapter Layer API can perform correctly. The DNS Adapter Layer API can get a list of DNS IPs of ads-dev.aylanetworks.com, and then compare them to test variable `dns_ads_ayla_com_ip`. If the DNS IPs in the list can be found in the `dns_ads_ayla_com_ip` test variable, then the DNS Adapter Layer API performs correctly.

  To get DNS IPs of ads-dev.aylanetworks.com, use the command: `nslookup` or `dig`, and then set `dns_ads_ayla_com_ip` using the `testvar set dns_ads_ayla_com_ip <dns_ip_list>` command. For example:

  ```
  testvar set dns_ads_ayla_com_ip "34.195.40.112,52.72.209.12"
  ```

- dns_www_ayla_com_ip

  The `dns_www_ayla_com_ip` test variable is a list of DNS IPs of www.aylanetworks.com. This test variable is used to verify that the DNS Adapter Layer API can perform correctly. The DNS Adapter Layer API can get a list of DNS IPs of www.aylanetworks.com, and then compare them to the `dns_www_ayla_com_ip` test variable. If the DNS IPs in the list can be found in the `dns_www_ayla_com_ip` test variable, then the DNS Adapter Layer API performs correctly.

To get DNS IPs of www.aylanetworks.com use the command: `nslookup` or `dig`, and then set `dns_www_ayla_com_ip` using the `testvar set dns_www_ayla_com_ip <dns_ip_list>` command. For example:

```
testvar set dns_www_ayla_com_ip "52.27.42.170,34.213.252.117"
```

- if_ip

The `if_ip` test variable is the IP of the main interface that PDA uses to connect to the Ayla cloud. This test variable is used to verify that the net-if Adapter Layer API can perform correctly. The net-if Adapter Layer API can get the IP of the main interface, then the net-if test case compares the IP to the `if_ip` test variable. If they are equal, the test case is marked as "pass."

To get the IP of the main interface, use the `ifconfig <interface_name>` command, then set `if_ip` using the `testvar set if_ip <if_ip>` command. For example,

```
testvar set if_ip 192.168.50.202
```

- if_mac

The `if_mac` test variable is the MAC of the main interface that PDA uses to connect to the Ayla cloud. This test variable is used to verify that the net-if Adapter Layer API can perform correctly. The net-if Adapter Layer API can get the MAC of the main interface, then the net-if test case compares the MAC to the `if_mac` test variable. If they are equal, the test case is marked as "pass."

To get the MAC of the main interface, use the ifconfig <interface_name> command, then set the `if_mac` using `testvar set if_mac <if_mac>` command. For example:

```
testvar set if_mac 08:00:27:ef:4d:55
```

- if_netmask

The `if_netmask` test variable is the netmask of the main interface that PDA uses to connect to the Ayla cloud. This test variable is used to verify the net-if Adapter Layer API can perform correctly. The net-if Adapter Layer API can get the netmask of the main interface, then the net-if test case compares the netmask to the `if_netmask` test variable. If they are equal, the test case is marked as "pass."

To get the netmask of the main interface, use the ifconfig <interface_name> command, then set `if_netmask` using the testvar set if_netmask <if_netmask> command. The default value of `if_netmask` is 255.255.255.0. If `if_netmask` of the main interface is equal to the default value, the `if_netmask` test variable does not need to be set. For example:

```
testvar if_netmask 255.255.255.0
```

- stream_server_ip

The Stream Server IP is the IP of the altest-server for Netstream to connect. Altest has test cases for Netstream to connect to altest-server, the stream_server_ip test variable should be set before performing the test.

If altest-server is located in current system, set `stream_server_ip` to `127.0.0.1`. If altest-server is located in another system in the local LAN, set `stream_server_ip` to the IP of the other system. For example:

```
testvar set stream_server_ip 192.168.50.105
```

- udp_server_lip

  The UDP local IP is the IP of the local system that receives UDP packets. The IP can have the following settings:

    o If the IP is set to `0.0.0.0`, this indicates that the local system can receive UDP packets from any interfaces.

    o If the IP is set to `127.0.0.1`, this indicates that the local system can receive UDP packets from a local interface.

    o If the IP is set to the IP of one interface, this indicates that the local system can receive UDP packets from the interface.

      For example:

      ```
      testvar set udp_server_lip 192.168.50.66
      ```

- udp_server_rip

  The UDP server remote IP is the IP of altest-server for net-udp to connect. Altest has test cases for net-udp to connect to altest-server; however, the `udp_server_rip` test variable should be set before performing the test:

    o If altest-server is located in the current system, set udp_server_ip to `127.0.0.1`.

    o If altest-server is located in another system in the local LAN, set udp_server_ip to the IP of the other system.

      For example:

      ```
      testvar set udp_server_rip 192.168.50.105
      ```

5.4 Writing Your Own Test Case

All of the test cases provided by Ayla are sufficient for testing the Adapter Layer APIs. If you want to run a specific test, you need to write a non-standard test case with a negative order. In the test case function, the return-value of `null` means success and `non-null` is the address of an error message string.

If the function to be tested is a synchronous procedure, the test framework can detect assertion failure in your function or its sub-functions. But if your function starts an asynchronous procedure (i.e. using `ada_call()`), and then returns a `null` value immediately, the actual test will be performed later. Also, the asynchronously executed function may cause assertion failure. In this case, the test framework cannot detect the failure, and instead, the PDA main function detects it and the default action is to terminate the program or reboot the device.

NOTE   Sometimes, when running the `altest all` test command if the test program accidently terminates or reboots, this could be the same assertion failure issue described above.

If you want to test a function that is asynchronous procedure, follow these steps:

1. In your test case, call `pfm_test_case_async_start()` before calling `al_ada_call()`. For example:

```
static char *xxxx_test_case_proc(const struct pfm_test_desc *pcase, int argc,
char **argv)
{
        struct al_ada_callback cb;
        if (my_init()) {
        return "my_init() failure";
        }
        al_ada_callback_init(&cb, xxxx_callback_proc, NULL);
        pfm_test_case_async_start();
        al_ada_call(&cb);
        return NULL;
}
PFM_TEST_DESC_DEF(xxxx_test_case_desc, -1, "xxxx_case_name",
EXPECT_SUCCESS, xxxx_test_case_proc, NULL);
```

2. Write your asynchronous callback function (which is part of the test case). For example:

```
static void xxxx_callback_proc(void *arg)
{
        int rc;
        char *status;
        rc = pfm_try_catch_assert();
        if (rc == 0) {
        status = async_action_to_be_tested();
        } else {
        status = "async_action_to_be_tested() assert fails";
        }
        pfm_try_catch_final();
        pfm_test_case_async_finished(status);
}
```

NOTE   When `pfm_test_case_async_start()` is called, the test framework does not execute the next test case until `pfm_test_case_async_finished()` is called.

---

6 Apptest

Apptest provides utilities to test the API calls in the Adapter Layer of the PDA. Apptest includes an apptest demo application and an automation apptest script. This section describes both and provides Apptest CLI Commands and Apptest ADA layer APIs.

6.1 Apptest Demo Application

The apptest demo application is an application layer test program running on the device. The apptest demo provides test CLI commands that can set the property, show the property, and show test results. When test CLI commands are called to set the property, apptest demo calls property set ADA APIs.

When apptest connects to the Ayla Cloud, apptest calls the ADA layer APIs to send a property and get property updates from the cloud. Additionally, apptest does the following:

- Calls the schedule ADA APIs when schedules and schedule actions are set in the Ayla Customer Dashboard.

- Calls log APIs when running and logging output.

- Calls OTA APIs when an OTA job is started in Ayla Customer Dashboard.

The ADA APIs can be tested manually by inputting test CLI commands to apptest demo, setting schedule in Ayla Customer Dashboard, and starting an OTA job in the dashboard.

6.2 Automation Apptest Script

The automation apptest script is a python script that communicate with the apptest demo program using a local pipeline or serial console. The script facilitates the testing of the ADA APIs, so that the ADA APIs do not need to be tested manually.

The script sends CLI commands to the apptest demo program, parses the output buffer of the commands, and gets the test results of ADA APIs. The tests include:

- prop ada API testing

- log ada API testing

- schedule ada API testing

- client ada API testing

- ota ada API testing

6.3 Apptest CLI Commands

CLI Commands	Description
test-show sys	Shows the OEM and DSN information
test-show res	Shows the test results
test-show prop	Shows the property name and value
test-prop set <prop_name> <prop_val>	Sets the value for the property whose name is <prop_name>

6.4 Apptest ADA Layer APIs

ADA API	API Description	How to Test
ada_sprop_mgr_register	Register a table of properties to the sprop prop manager	Start apptest demo
ada_sprop_send	Send property update	Input the `test-prop set input 202` CLI command in apptest demo, and apptest demo sends output property
ada_sprop_send_by_name	Send a property update by name	Input `test-prop set decimal_in 88.5` CLI command in apptest demo, and apptest demo sends `decimal_out` property
ada_sprop_set_bool	Set an `ATLV_BOOL` property value to the value	Input `test-prop set Blue_LED 1` CLI command in apptest demo
ada_sprop_get_bool	Get an `ATLV_BOOL` type property from the sprop structure	Input `test-prop get Blue_LED` CLI command in apptest demo
ada_sprop_set_int	Set an `ATLV_INT` or `ATLV_CENTS` property value	Input `test-prop set input -101` CLI command in apptest demo
ada_sprop_get_int	Get an `ATLV_INT` or `ATLV_CENTS` type property from the sprop structure	Input `test-prop get input` CLI command in apptest demo
ada_sprop_set_uint	Set an `ATLV_UINT` property value to the value	Input `test-prop set uinput 212` CLI command in apptest demo
ada_sprop_get_uint	Get an ATLV_UINT type property from the sprop structure	Input `test-prop get uinput` CLI command in apptest demo
ada_sprop_set_string	Set an `ATLV_UTF8` property value to the value	Input `test-prop set cmd abc` CLI command in apptest demo
ada_sprop_get_string	Get an `ATLV_UTF8 type` property from the sprop structure	Input `test-prop get cmd` CLI command in apptest demo
ada_sprop_dest_mask	Mask of currently-connected destinations	Start apptest demo, which gets the mask in main loop
ada_prop_mgr_register	Property manager uses this to register itself as a handler of	Start apptest demo, and the prop manager is registered during

ADA API	API Description	How to Test
	properties	initialization
ada_prop_mgr_ready	Property manager reports that it is ready to receive data	Start apptest demo, and the prop manager is registered during initialization
ada_prop_mgr_request	Request a property value from the ADS	Start apptest demo, then connect to the ADS
ada_prop_mgr_send	Post a property to ADS/app	Apptest demo connects to the ADS, then posts property outlet_pmgr to the ADS using the property manager
send_done	Callback to report success/failure of property post to ADS/apps	Apptest demo connects to the ADS, and the send_done callback is called after the property outlet_pmgr is sent to the ADS
connect_status	ADC reports a change in its connectivity	Apptest demo connects to the ADS, connect_status callback is called
prop_recv	Receive property value from ADS or app	Apptest demo connects to the ADS, and then creates a new datapoint for property uinput_pmgr in the Ayla Customer Dashboard
client_reg_window_start	Starts the registration window	Apptest demo connects to the ADS for the first time, the client_reg button changes, and the registration window begins
ada_init	Initializes the ADA client environment	Start apptest demo, and ada_init is called during initialization
ada_client_up	Starts the ADA client agent	Input the up CLI command in apptest demo
ada_client_down	Shuts down the ADA client agent	Inputs the down CLI command in apptest demo
ada_sched_init	Initializes schedules and allocates space	Start apptest demo, and schedule init is called during initialization
ada_sched_enable	Turns on schedule handling	Start apptest demo, and schedule enable is called during initialization
ada_sched_set_name	Sets the name of the schedule	Start apptest demo, schedule set name is called when loading schedule configuration
ada_sched_get_index	Gets the name and value for the schedule	Start apptest demo, then set schedule 'sched1' in the Ayla Customer Dashboard
ada_sched_set_index	Sets the value for a schedule by index	Start apptest demo, schedule set index is called when loading schedule configuration
ada_ota_register	Registers the handler for OTA	Start apptest demo, ota register is called during initialization
ada_ota_start	Gives permission for OTA to start	Uploads the OTA image and then performs OTA in the Ayla Customer Dashboard

ADA API	API Description	How to Test
ada_ota_report	Report status of OTA update	Uploads OTA image and then perform OTA in the Ayla Customer Dashboard
log_info	Sending information to log	Start apptest demo, then connect to ADS, and the `virtual button control` info is logged
log_put	Sending log in default mode	Start apptest demo, then connects to the ADS, and the button process is logged.

7 Apptest Environment Setup

Setting up the apptest environment involves the following procedures (which are described in this section):
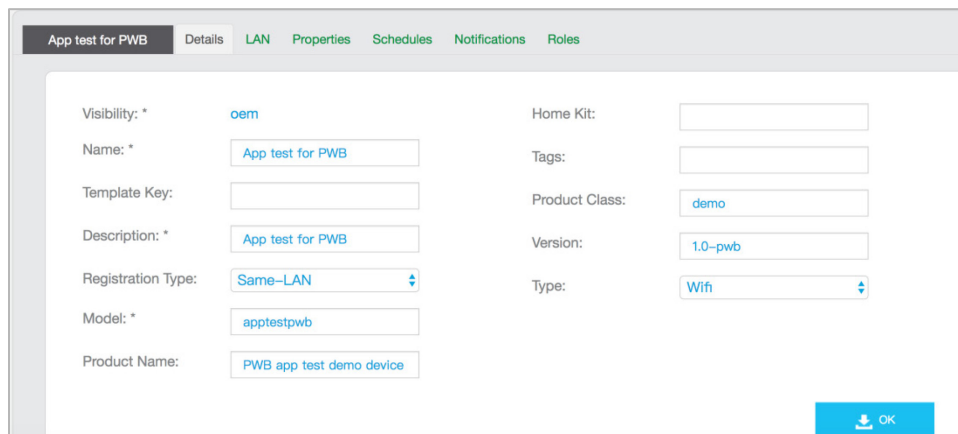
- apptest demo app setup

- automation apptest script setup

- automation apptest script run

7.1 Apptest Demo App Setup Procedure

In the PDA source packet, the apptest demo is located in ayla/test/apptest/src. Follow these steps to complete the setup:

1. Enter the OEM of apptest demo, which is `apptestpwb`.

2. Use `conf-gen.py` in platform/linux/utils/ to generate the config file of apptest demo. For example:

```
./conf-gen.py --oem-key <oem_key> --region US ACxxx.xml <oem_id> apptestpwb
```

3. In the Ayla Developer Portal, enter the details of the template for apptest demo. Refer to the following as an example:

    a. Log in to the Ayla Developer Portal at https://developer.aylanetworks.com/, and click **View My Devices**.

    b. Click **Templates**, which is a tab across the top of the portal.

    c. Click ⊕ ADD .

    d. Fill in the details requested in the New Template dialog box. Following is an example of this dialog box completed.

e. Click the **Properties** tab to show the PROPERTIES page where you can modify or add
    template properties. Refer to the following example:

f. Click **Schedules** to show the SCHEDULES page to add or modify schedules. Refer to
    the following example:

NOTE   For more information on how to use the Ayla Developer Portal, refer to the Ayla
            Developer Portal User Guide (AY006UDP3) on support.aylanetworks.com.

---

4. Enter the commands for the apptest demo. Following describes the commands:

- test-show sys

    This command shows the OEM and DSN information. Following is an example:

    ```
    PWB apptest> test-show sys
    model = AY008PWB1
    serial = AC000W001565370
    oem = 0dfc7900
    oem_model = apptestpwb
    version = ADA 2.1.1-beta demo 2018-07-27 18:49:56 5069bb8
    version_demo = 1.0-pwb
    ```

- test-show prop

  This command shows the property name and its corresponding value. Following is an example:

  ```
  PWB apptest> test-show prop
  oem_host_version = 1.0-pwb
  version = ayla_apptest_demo 1.3 Jul 27 2018 21:42:37
  Blue_button = 0
  Blue_LED = 0
  …...
  ```

- test-show res

  This command shows the test results for the ADA APIs. Following is an example:

  ```
  PWB apptest> test-show res
  C610678      ada_sprop_mgr_register ->   0 (      1) <none>
  C610679          ada_sprop_send -> -99 (      0) <not tested>
  …...
  ```

- test-prop set <prop_name> <prop_value>

  This command sets the value for the property whose name is prop_name. Following is an example:

  ```
  PWB apptest> test-prop set Blue_LED 1
  [ada] 1236453 info:  c mod: Blue_LED set to 1
  ```

5. Compile apptest demo:

   a. In the PDA source packet, input the following commands to compile apptest demo:

   ```
   make clean
   make
   ```

   b. Use the following command to run apptest demo:

   ```
   make run EXEC=apptest
   ```

6. Make sure the apptest demo can be called anywhere:

   Add the location of apptest to the PATH variable as follows:

   ```
   export $PATH=$PATH:<location_apptest_bin>
   ```

7.2 Automation Apptest Script Setup Procedure

The automation apptest script is an automation python script that communicates with the apptest demo application using a local pipe or serial port. If the automation script communicates with the device via the serial port, do the following:

1. Set `localCmd` to `''`

2. Modify the port to `serial port /dev/tty*` in the file called:
   `~/automation_pda/automation_apptest_script/py/lib/TestCaseAda.py`

Following is the setup procedure:

1. Install Ubuntu:

   ```
   ~$ lsb_release -a
   ```

   Response:

   ```
   No LSB modules are available.

   Distributor ID: Ubuntu

   Description: Ubuntu 12.04.5 LTS

   Release: 12.04

   Codename: precise
   ```

2. Install python3.6.2:

   ```
   sudo apt-get install zlib1g

   sudo apt-get install zlib1g-dev

   wget https://www.python.org/ftp/python/3.6.2/Python-3.6.2.tar.xz

   tar -xvf Python-3.6.2.tar.xz

   cd Python-3.6.2

   ./configure

   make

   sudo make install
   ```

   Install `virtualenv` and `virtualenvwrapper` as follows:

   ```
   pip install virtualenv

   pip install virtualenvwrapper
   ```

3. Create the virtual environment for the apptest automation script as follows:

   a. Modify `~/.bashrc` to use `virtualenvwrapper`. Add the following lines to `~/.bashrc`:

   ```
   export WORKON_HOME=$HOME/venv

   export VIRTUALENVWRAPPER_PYTHON=/usr/local/bin/python3

   . /usr/local/bin/virtualenvwrapper.sh
   ```

b. Use the following commands to create `virtualenv`:

```
mkdir ~/venv
cd ~/venv
mkvirtualenv venv_automation_pda
```

4. Activate the virtual environment as follows:

```
source ~/venv/venv_automation_pda/bin/activate
```

5. Create `pip.conf`, and add `config` to it:

```
cd ~/.pip/
touch pip.conf
vi ~/.pip/pip.conf

[global]
index-url = http://pypi.douban.com/simple
trusted-host = pypi.douban.com
disable-pip-version-check = true
timeout = 120
```

6. Copy the automation script to the working directory, which must be:
    `~/automation_pda/automation_apptest_scrip`.

```
mkdir ~/automation_pda
cp ayla/test/apptest/automation_apptest_script/  ~/automation_pda/ -a
cd ~/automation_pda/automation_apptest_script
```

7. Run `testenv.sh` to install the packages and init automation environment.

```
cd ~/automation_pda/automation_apptest_script/util
./testenv.sh
```

7.3 Automation Apptest Script Run Procedure

1. Open a new terminal in Ubuntu.

2. Make sure the apptest is located in `/usr/bin`, and the apptest program can be called in the automation script.

3. Activate the virtual environment as follows:

```
source ~/venv/venv_automation_pda/bin/activate
```

4. Run the `TestAdaSuilte.py` automation script:

```
TestAdaSuilte.py located in
~/automation_pda/automation_apptest_script/py/ada.
```

5. Run the `TestAdaSuilte.py` automation script using the following command:

```
python3 TestAdaSuite.py
```

6. Wait for the prompt and then perform corresponding actions in the Ayla Customer Dashboard as follows:

a. Navigate and log in to the dashboard:

   o US: https://dashboard.aylanetworks.com

   o EU: https://dashboard-field-eu.aylanetworks.com/

   o CN: https://dashboard.ayla.com.cn/

b. Create a new datapoint for the `uinput_pmgr` property in the dashboard:

   1. Click **Devices** in the Navigation Panel on the left side of the dashboard.

   2. Double-click the device that has the `uinput_pmgr` property.

   3. Click **PROPERTIES** in the Device Navigation menu (see below).

DEVICE	>
PROPERTIES	>
COMMANDS	>
SCHEDULES	>
TIMEZONES	>
NOTIFICATIONS	>
METADATA	>
AYLA IMAGES	>
HOST MCU IMAGES	>
ALERT HISTORY	>
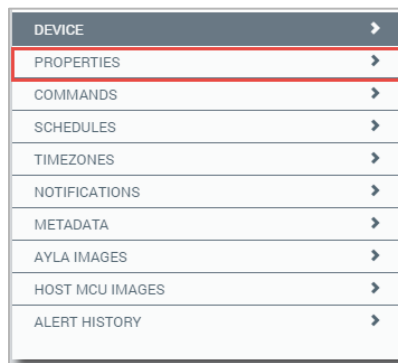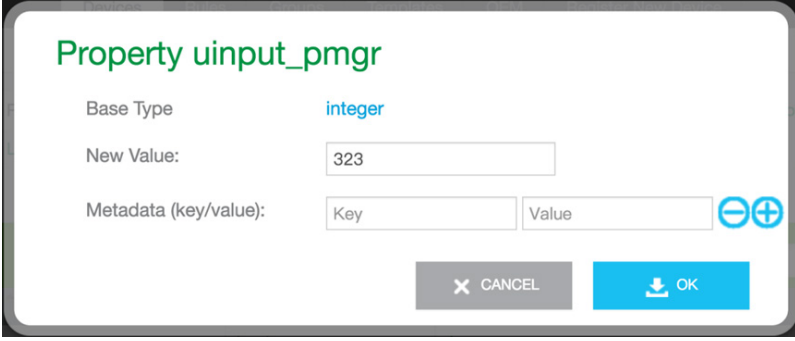
   4. Double-click the `uinput_pmgr` property on the Properties page.

   5. Click **CREATE DATAPOINT** to create the new datapoint.

NOTE For more information on how to use the Ayla Customer Dashboard, refer to the Ayla Customer Dashboard User Guide (AY006UDB3) on support.aylanetworks.com.

You can view and change the value for this datapoint in the Ayla Developer Portal as follows:

1. Log in to the Ayla Developer Portal at https://developer.aylanetworks.com/, and click the **Devices** tab across the top of the portal.

2. Click the serial number of the device to open the PROPERTIES page.

3. Click the uinput_pmgr property under DISPLAY NAME to open the Details for this property.

4. Click the **click to update** link in the Current Value field to open the dialog box to view and change the value for the datapoint, as shown below:
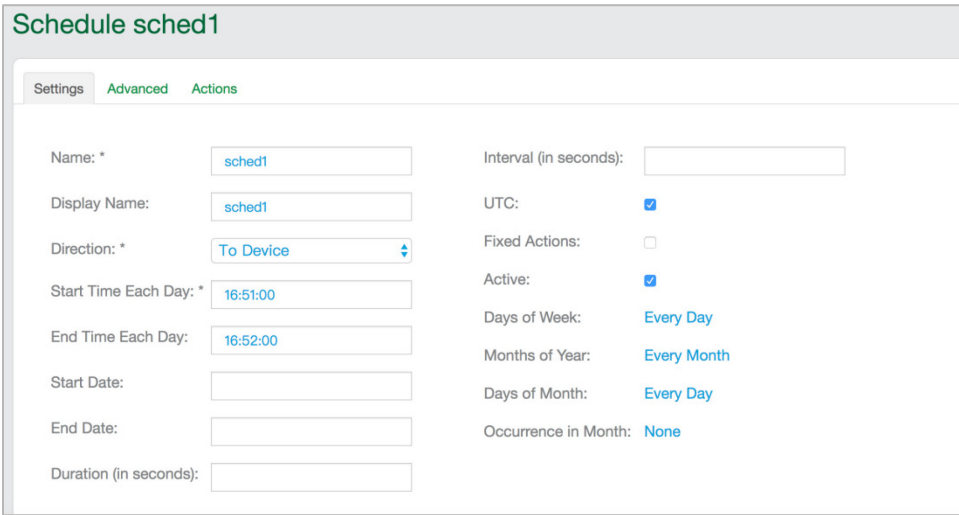
c. Configure the schedule, which in this example is sched1, in the Ayla Developer Portal:

1. Log in to the Ayla Developer Portal at https://developer.aylanetworks.com/, and click the **Devices** tab across the top of the portal.

2. Click the serial number of the device, which opens the PROPERTIES page.

3. Click the **Schedules** tab.

4. Click ⊕ ADD to open the New Schedule dialog box, fill in the details for your schedule (as shown below), and then click OK.

d. Create and start an OTA job for apptest in the Ayla Customer Dashboard:

1. Navigate and log in to the dashboard:

   o US: https://dashboard.aylanetworks.com

   o EU: https://dashboard-field-eu.aylanetworks.com/

   o CN: https://dashboard.ayla.com.cn/

2. Click **OTA** in the Navigation Panel on the left side of the dashboard.

3. Click the **Host MCU Images** tab (shown below).

Ayla Images   Host MCU Images   OTA Jobs
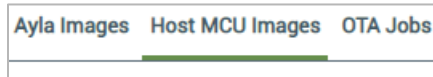
4. Click **CREATE** (either button at the top or bottom of the page).

5. Enter all of the details for the New Host Image, and then click **SAVE**, which adds it to the table listing on the Host MCU Image tab.

6. Click this new Host MCU image in the table listing to edit any of the details, as shown below:

Host MCU Image		×
OEM Model *	apptestpwb	
Release notes (URL)		
Version *	1.1	
Min version		
Size (bytes)	649407	
Source:	local	
Description	michael_pda_apptest_local	

CLOSE   EDIT

7. On the Host MCU Image tab, click the Create OTA job icon, [icon], in the same row as the new Host MCU image that you created in steps 3-6 above.

8. Enter all of the details for the OTA job in the Create Job dialog box (example shown below), and then click **CREATE**.

Create Job		×
Select a group:	michael_pda_ota_group	
Name:	michael_pda_apptest_local_job	
☐ Ignore validations ☐ Start after creation		CANCEL CREATE

9. Click the OTA Jobs tab, as shown below:

Ayla Images   Host MCU Images   OTA Jobs
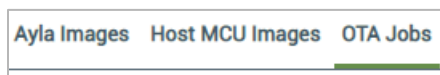
10. Locate the new OTA job, and then click the **Start OTA Job** icon.

	Name	Status	User	Creation Date (UTC)	Type	Image Version	SW Version (from)	Devices	Passed	Failed	Last Updated (UTC)	LAN OTA	Action
☑	michael_pda_apptest_local_job	⟳ initialized	32175	07/31/2018 at 9:00:45	host_mcu	1.1		1	0	0	07/31/2018 at 9:00:45	☐	↻ ⊙ 🗑

11. Click ACCEPT in the Confirmation dialog box that opens to start the OTA job.

Confirmation   ×

Are you sure you want to start the selected job?

CANCEL   ACCEPT

e. Collect the test results when the automation script is finished. The test results are located in `~/automation_pda/automation_apptest_script/logs`. Collect the following test results:

- `TestAdaProp_xxx.tr`

- `TestAdaLog_xxx.tr`

- `TestAdaSched_xxx.tr`

- `TestAdaClient_xxx.tr`

- `TestAdaOta_xxx.tr`

The log is shown in the following line:

```
{"case_id":610678, "status_id":1, "elapsed":"3s",
"comment":"Expected: <[0]>, got: <0>"}
```

Where status_id value 1 means success, and other value means failure.

Ayla Networks

4250 Burton Drive, Santa Clara, CA 95054
Phone: +1 408 830 9844
Fax: +1 408 716 2621