ARM

PAL

PAL Porting Guide

Last Updated: 3 January 2017

ARM

Internal Revision History

Rev	Date	Author	Description
1.0	26/09/16	Mohammad Abo Mokh	First draft.
1.1	10/11/16	Mohammad Abo Mokh	Refining the document chapters.
1.1.1	17/11/16	Netanel Gonen	Update chapter
1.1.2	14/12/16	Nir Sonnenschein	Networking chapter
1.1.3	23/12/16	Pekka Niskanen	Technical writer’s edits.
1.1.4	3/1/17	Alex Volinski	Text page alignment adjustments

Document Revision History

Rev	Date	Description
1.0	Not yet released	First release version
1.1	3/1/17	1.1 Public Release

Contents

List of Figures

List of Tables

1 About this document

This document explains

- The steps of porting the PAL Platform APIs to a new target platform.
- All PAL modules and layers.

To ensure correct porting, expected tests results are attached.

Notes:

- Currently, mbedTLS library is a requirement from the underlying platform to support mbed Client porting.
- To ensure mbed TLS is configured securely in the customer platform as part of porting, mbed TLS entropy should be bound to it (see https://docs.mbed.com/docs/mbed-os-handbook/en/5.1/advanced/tls_porting/) and implement seed persistence - `mbedtls_nv_seed_read/mbedtls_nv_seed_write`.

1.1 Intended Audience

1.2 Referenced Documents

Table 1: Referenced Documents

Ref	Name
[PROD_DOCNAME1]	
[PROD_DOCNAME2]	

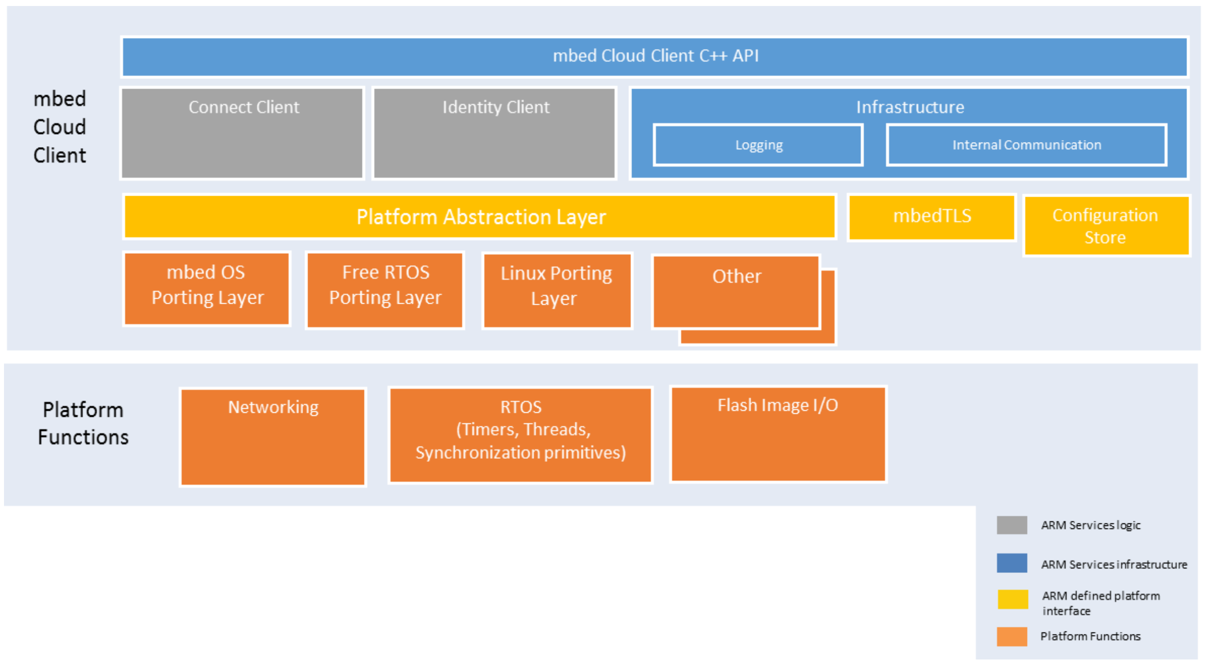
1.3 Terms and Abbreviations

Table 2: Glossary

Term	Description

2 Introduction

A Platform Abstraction Layer (PAL) connects mbed Cloud Client with the underlying platform.



The main purpose of PAL is to enable easy and fast mbed Cloud Client services portability, allowing them to operate over wide range of ARM Cortex based platforms running different operating systems with various libraries (networking for example).

3 PAL structure

3.1 Service API layer

The service APIs are exposed to the mbed Cloud Client services. The APIs are identical for all platforms and operating systems, and should not be modified.

3.2 Platform API layer

This layer provides a standard set of baseline requirements for the platform that need to be be implemented to enable mbed Cloud Client to run on the target platform.

This layer needs to be implemented differently for each target OS or library;

PAL provides reference implementations over several operating systems including mbed OS.

4 PAL types

PAL defines data types for service and platform APIs. The APIs make communication easier and safer and set coding conventions and function return values for the PAL interfaces over the different platforms.

The PAL types, macros and errors are defined in separate files:

- `pal_types.h`
- `pal_macros.h`
- `pal_errors.h`

For each PAL module, there are specific data types that are defined in the related header files. The data types are introduced and described in detail in further chapters.

4.1 palError_t

An enumeration of the error values used by PAL. All the error values are negative. The errors are divided into modules according to the PAL modules:

```
typedef enum {
    PAL_ERR_MODULE_GENERAL,
    PAL_ERR_MODULE_PAL,
    PAL_ERR_MODULE_C,
    PAL_ERR_MODULE_RTOS,
    PAL_ERR_MODULE_NET,
    PAL_ERR_MODULE_TLS,
    PAL_ERR_MODULE_CRYPTO,
    PAL_ERR_MODULE_UPDATE
} palErrorModules_t;
```

The values are used to separate the errors into different origins of values, for example:

```
typedef enum {
// generic errors
PAL_ERR_GENERAL_BASE    = (-1 << PAL_ERR_MODULE_GENERAL),
PAL_ERR_GENERIC_FAILURE = PAL_ERR_GENERAL_BASE,
…
// RTOS errors
PAL_ERR_RTOS_ERROR_BASE = (-1 << PAL_ERR_MODULE_RTOS),
PAL_ERR_RTOS_PARAMETER  = PAL_ERR_RTOS_ERROR_BASE + 0x80,
PAL_ERR_RTOS_RESOURCE   = PAL_ERR_RTOS_ERROR_BASE + 0x81,
…
} palError_t;
```

In this example:

- Generic error code values start from (`0xFFFFFFF0`) and down.
- RTOS error code values start from (`0xFFFFF000`) and down.

The enumeration is built in this way to allow scalable error addition in the future and to prevent value collisions between different PAL modules.

4.2 palStatus_t

The main use of this type is for the return values of PAL functions. This data type makes communication between Service and Platform functions much easier.

The definition is in `pal_types.h`:

```
typedef int32_t palStatus_t;
```

The expected values:

- PAL_SUCCESS in case of success → defined in `pal_types.h`:
    - `#define PAL_SUCCESS 0`
- Values from `palError_t` enumeration in case of errors.

4.3 palBuffer_t

This data type represents a mutable buffer. The definition is in `pal_types.h`:

```
typedef struct _palBuffer_t
{
    uint32_t  maxBufferLength;
    uint32_t  bufferLength;
    uint8_t *buffer;
} palBuffer_t;
```

4.4 palConstBuffer_t

This data type represents an immutable buffer. The definition is in `pal_types.h`:

```
typedef struct _palConstBuffer_t
{
    const uint32_t  maxBufferLength;
    const uint32_t  bufferLength;
    const uint8_t *buffer;
} palConstBuffer_t;
```

5 PAL configuration

PAL has a configuration header file `pal_configuration.h`. This file contains defines that can be configured in file or via compilation flags.

It provides a robust implementation for the different platforms. You can modify it, only if required, when porting to a new platform. The configuration flags are as follows:

Table 3: Configurations

Flag	Values Type	Way of definition	Configuration
`PAL_NET_TCP_AND_TLS_SUPPORT`	`true/false`	hardcoded	Add PAL support for TCP.
`PAL_NET_ASYNCHRONOUS_SOCKET_API`	`true/false`	hardcoded	Add PAL support for asynchronous sockets.
`PAL_NET_DNS_SUPPORT`	`true/false`	hardcoded	Add PAL support for DNS lookup.
`PAL_RTOS_64BIT_TICK_SUPPORTED`	`true/false`	hardcoded	Platform must support 64bit tick counter.
`PAL_UNIQUE_THREAD_PRIORITY`	`true/false`	compilation flag	If defined code skips the uniqueness priority check.
`PAL_MAX_NUMBER_OF_THREADS`	`Number`	hardcoded	Max number of threads in the system.
`PAL_MAX_SUPORTED_NET_INTERFACES`	`Number`	hardcoded	Max number of interfaces in the system.

6 PAL Modules

This chapter explains the PAL Platform Layer APIs, that you need to implement when porting to a new platform. Each platform function signature MUST have the prefix `pal_plat_` and the name of the function, for example:

```
void pal_plat_osReboot(void);
```

6.1 Real-time operating system (RTOS)

6.1.1 Overview

The RTOS module is responsible for providing the real-time operating system functionality. This includes primitives such as:

- Threads.
- Mutexes.
- Semaphores.

It provides also an API for:

- Kernel ticks.
- Timers.
- Memory pool.
- Message queue.

The following is a list of the APIs that MUST be implemented:

```
pal_plat_osReboot
pal_plat_osKernelSysTick
pal_plat_osKernelSysTick64
pal_plat_osKernelSysTickMicroSec
pal_plat_osKernelSysMilliSecTick
pal_plat_osKernelSysTickFrequency
pal_plat_osThreadCreate
pal_plat_osThreadTerminate
pal_plat_osThreadGetId
pal_plat_osThreadGetLocalStore
pal_plat_osDelay
pal_plat_osTimerCreate
pal_plat_osTimerStart
pal_plat_osTimerStop
pal_plat_osTimerDelete
pal_plat_osMutexCreate
pal_plat_osMutexWait
pal_plat_osMutexRelease
pal_plat_osMutexDelete
pal_plat_osSemaphoreCreate
pal_plat_osSemaphoreWait
pal_plat_osSemaphoreRelease
```

```
pal_plat_osSemaphoreDelete
pal_plat_osPoolCreate
pal_plat_osPoolAlloc
pal_plat_osPoolCAlloc
pal_plat_osPoolFree
pal_plat_osPoolDestroy
pal_plat_osMessageQueueCreate
pal_plat_osMessagePut
pal_plat_osMessageGet
pal_plat_osMessageQueueDestroy
pal_plat_osAtomicIncrement
```

6.1.2 RTOS types

The RTOS data types are defined in the `pal_rtos.h` and `pal_plat_rtos.h` files.

6.1.2.1 Type handles

The RTOS module gives an ID for each primitive and data type. This ID holds in a special type:

```
// Thread ID Handle
typedef uintptr_t palThreadID_t;
// Timer ID Handle
typedef uintptr_t palTimerID_t;
// Mutex ID Handle
typedef uintptr_t palMutexID_t;
// Semaphore ID Handle
typedef uintptr_t palSemaphoreID_t;
// MemoryPool ID Handle
typedef uintptr_t palMemoryPoolID_t;
// Message Queue ID Handle
typedef uintptr_t palMessageQID_t;
```

6.1.2.2 Enumerations

The RTOS module has the following enumerations:

- `palTimerType_t`: Specifies the timer type as "One Shot" or "Periodic".
- `palThreadPriority_t`: Specifies the available thread priorities.

```
//! Timers types supported in PAL
typedef enum  palTimerType {
    palOsTimerOnce = 0, /*! One shot timer*/
    palOsTimerPeriodic = 1 /*! Periodic (repeating) timer*/
} palTimerType_t;

//! Available priorities in PAL implementation.
typedef enum    pal_osPriority {
    PAL_osPriorityIdle = -3,
    PAL_osPriorityLow = -2,
    PAL_osPriorityBelowNormal = -1,
    PAL_osPriorityNormal = 0,
    PAL_osPriorityAboveNormal = +1,
    PAL_osPriorityHigh = +2,
    PAL_osPriorityRealtime = +3,
    PAL_osPriorityError = 0x84
} palThreadPriority_t;
```

6.1.2.3 Function pointer

The RTOS module uses the following function pointers:

- `palTimerFuncPtr`: A pointer to the timer function.
- `palThreadFuncPtr`: A pointer to the thread function.

```
//! PAL timer function prototype
typedef void(*palTimerFuncPtr)(void const *funcArgument);

//! PAL thread function prototype
typedef void(*palThreadFuncPtr)(void const *funcArgument);
```

6.1.2.4 Data structure

The RTOS module has the following data structures:

- `palThreadLocalStore_t`: This data structure can be attached to the threads to create global variables, or for any other data sharing and communication purposes.
- `g_palThreadPriorities`: An array to hold a counter for the available priorities. The main purpose of this array is a binary counter (priority used or not). It is used to enforce one use of each priority; you can disable the usage of the array by defining the configuration flag `PAL_UNIQUE_THREAD_PRIORITY`" as false.

NOTE! You must define and initialize this array in the porting code.

```
//! Thread Local Store struct.
//! Can be used to hold: State, configurations and etc inside the thread.
typedef struct pal_threadLocalStore{
    void* storeData;
} palThreadLocalStore_t;

//! An array of PAL thread priorities.
uint8_t g_palThreadPriorities[PAL_MAX_NUMBER_OF_THREADS];
```

6.1.2.5 Defines

The RTOS module has the following defines:

- PAL_RTOS_WAIT_FOREVER: Time related functions and primitives define, from platform point of view, if a function receives this value, it must act as waiting for ever.
- PRIORITY_INDEX_OFFSET: Priority array counter. It is used to map between priorities and their index in the `g_palThreadPriorities` array.

This define is available only if the configuration flag `PAL_UNIQUE_THREAD_PRIORITY` is set to be true.

```
#define PAL_RTOS_WAIT_FOREVER PAL_MAX_UINT32
#define PRIORITY_INDEX_OFFSET 3
```

6.1.3 APIs

This section defines the requirements for each RTOS API. It also includes tips and warnings to save time when porting. The requirements cannot replace the documentation found in the `pal_plat_rtos.h`.

For each API, there is information about:

- The exact declaration.
- Requirements.
- Special notes.
- Related tests in PAL test code.

6.1.3.1 Reboot

Function declaration:

`void pal_plat_osReboot(void);`

Requirements:

This function reboots the device.

Related test:

CustomisedTest.

6.1.3.2 RTOS initialization

Function declaration:

`palStatus_t pal_plat_RTOSInitialize(void* opaqueContext);`

Requirements:

- This function serves the platform in case of needed initializations, for example, global data structures.
- Any allocated MUST be deallocated in the destroy function.

  NOTE! If not necessary, please ignore the `opaqueContext` argument. If some external data is required for initialization you can pass it in through this parameter.

Related test:

`pal_init_test`.

6.1.3.3 RTOS destroy

Function declaration:

`void pal_plat_RTOSDestroy(void);`

Requirements:

- Deallocate any allocated memory in the RTOS Initialization API.
- De-initialize global data structures if there are any.

Related test:

`pal_init_test`.

6.1.3.4 RTOS kernel

This section contains kernel functions. Most of them are meant to be used for time comparison. Some are for converting between kernel systems ticks to different time units.

NOTE! The referred tick is the OS (Platform) kernel system tick counter.

Function declaration:

`uint32_t pal_plat_osKernelSysTick(void);`

Requirements:

Return the RTOS kernel system timer counter.

Related test:

- `pal_osKernelSysTick_Unity`.
- `pal_osDelay_Unity`.
- `BasicTimeScenario`.
- `TimerUnityTest`.

Function declaration:

`uint64_t pal_plat_osKernelSysTick64(void);`

Requirements:

Return the RTOS kernel system timer counter.

NOTE! This is an optional API. It depends on the `PAL_RTOS_64BIT_TICK_SUPPORTED` configuration flag.

Related test:

`pal_osKernelSysTick64_Unity`.

Function declaration:

`uint64_t pal_plat_osKernelSysTickMicroSec(uint64_t microseconds);`

Requirements:

Convert the value from microseconds to kernel system ticks.

Related test:

- `pal_osKernelSysTickMicroSec_Unity`.
- `pal_osKernelSysMilliSecTick_Unity`.
- `BasicTimeScenario`.

Function declaration:

`uint64_t pal_plat_osKernelSysMilliSecTick(uint64_t sysTicks);`

Requirements:

Convert the value from kernel system ticks to milliseconds.

Related test:

`pal_osKernelSysMilliSecTick_Unity`.

Function declaration:

`uint64_t pal_plat_osKernelSysTickFrequency(void);`

Requirements:

Get the system tick frequency.

NOTE! The system tick frequency MUST be more than 1KHz (at least one tick per millisecond).

Related test:

`pal_osKernelSysTickFrequency_Unity`.

6.1.3.5 Threads

The PAL thread functionality relies on the following:

- The number of threads is limited by the configuration definition `PAL_MAX_NUMBER_OF_THREADS`.
- Each thread has a unique priority, unless the configuration flag `PAL_UNIQUE_THREAD_PRIORITY` is set to false.

NOTE! If unique thread priority is defined, the max number of threads cannot exceed the number of predefined thread priorities.

- When a thread execution is completed (was not terminated by user), the priority array must be updated accordingly.

TIP! In some cases, a thread wrapper function can make implementation easier. For example, refer to thread implementation for mbedOS.

- Threads can share data, state or communicate via `palThreadLocalStore_t`. More than one local store can be shared shared between different thread groups. For each thread, there is a single thread local store.
- The thread stack is allocated by the application code.

Function declaration:

```
palStatus_t pal_plat_osThreadCreate(palThreadFuncPtr function, void* funcArgument, palThreadPriority_t priority, uint32_t stackSize, uint32_t* stackPtr, palThreadLocalStore_t* store, palThreadID_t* threadID);
```

Requirements:

Create a new thread and validate that the given priority can be used.

Related test:

- `PrimitivesUnityTest1`. – Via `palRunThreads()`.
- `PrimitivesUnityTest2`.

Function declaration:

```
palStatus_t pal_plat_osThreadTerminate(palThreadID_t* threadID);
```

Requirements:

Terminate and free allocated data for the thread.

Related test:

`PrimitivesUnityTest1` via `palRunThreads()`.

Function declaration:

```
palThreadID_t pal_plat_osThreadGetId();
```

Requirements:

Get the ID of the current thread.

Related test:

`PrimitivesUnityTest1` via thread running functions.

Function declaration:

```
void* pal_plat_osThreadGetLocalStore();
```

Requirements:

Get the storage of the current thread.

Related test:

`PrimitivesUnityTest1` via thread running functions.

6.1.3.6 Delay

Function declaration:

`palStatus_t pal_plat_osDelay(uint32_t milliseconds);`

Requirements:

Wait for a specified period of time in milliseconds.

Related test:

- `pal_osDelay_Unity`.
- `BasicTimeScenario`.
- `TimerUnityTest`.

6.1.3.7 Timers

Function Declaration:

`palStatus_t pal_plat_osTimerCreate(palTimerFuncPtr function, void* funcArgument, palTimerType_t timerType, palTimerID_t* timerID);`

Requirements:

Create a timer.

Related test:

`TimerUnityTest`.

Function declaration:

`palStatus_t pal_plat_osTimerStart(palTimerID_t timerID, uint32_t millisec);`

Requirements:

Start or restart the timer.

Related test:

`TimerUnityTest`.

Function declaration:

`palStatus_t pal_plat_osTimerStop(palTimerID_t timerID);`

Requirements:

Stop the timer.

Related test:

`TimerUnityTest`.

Function declaration:

`palStatus_t pal_plat_osTimerDelete(palTimerID_t* timerID);`

Requirements:

- Delete the timer object.
- In case of a running timer, the function MUST stop the timer before deletion.

Related test:

`TimerUnityTest`.

6.1.3.8 Mutex

Function declaration:

`palStatus_t pal_plat_osMutexCreate(palMutexID_t* mutexID);`

Requirements:

Create and initialize a mutex object.

Related test:

`PrimitivesUnityTest1` via thread running functions.

Function declaration:

```
palStatus_t pal_plat_osMutexWait(palMutexID_t mutexID, uint32_t
millisec);
```

Requirements:

Wait until a mutex becomes available.

Related test:

`PrimitivesUnityTest1` via thread running functions.

Function declaration:

`palStatus_t pal_plat_osMutexRelease(palMutexID_t mutexID);`

Requirements:

Release a mutex that was obtained by `osMutexWait`.

Related test:

`PrimitivesUnityTest1` via thread running functions.

Function declaration:

`palStatus_t pal_plat_osMutexDelete(palMutexID_t* mutexID);`

Requirements:

Delete a mutex object. In success, `*mutexID = NULL`.

Related test:

`PrimitivesUnityTest1` via thread running functions.

6.1.3.9 Semaphore

Function declaration:

```
palStatus_t pal_plat_osSemaphoreCreate(uint32_t count,
palSemaphoreID_t* semaphoreID);
```

Requirements:

Create and initialize a semaphore object.

Related test:

`PrimitivesUnityTest1` via thread running functions.

Function declaration:

```
palStatus_t pal_plat_osSemaphoreWait(palSemaphoreID_t semaphoreID,
uint32_t millisec, int32_t* countersAvailable);
```

Requirements:

Wait until a semaphore token becomes available.

Related test:

`PrimitivesUnityTest1` via thread running functions.

Function declaration:

```
palStatus_t pal_plat_osSemaphoreRelease(palSemaphoreID_t
semaphoreID);
```

Requirements:

Release a semaphore token.

Related test:

`PrimitivesUnityTest1` via thread running functions.

Function declaration:

```
palStatus_t pal_plat_osSemaphoreDelete(palSemaphoreID_t*
semaphoreID);
```

Requirements:

Delete a semaphore object. In success, `*semaphoreID = NULL`.

Related test:

- `PrimitivesUnityTest1`. – Via thread running functions.
- `PrimitivesUnityTest2`.

6.1.3.10 Memory pool

PAL provides an API for a memory pool which is allocated according to two parameters:

- Block size.
- Block count.

Function declaration:

```
palStatus_t pal_plat_osPoolCreate(uint32_t blockSize, uint32_t
blockCount, palMemoryPoolID_t* memoryPoolID);
```

Requirements:

Create and initialize a memory pool.

Related test:

`MemoryPoolUnityTest`.

Function declaration:

```
void* pal_plat_osPoolAlloc(palMemoryPoolID_t memoryPoolID);
```

Requirements:

Allocate a single memory block from a memory pool.

Related test:

`MemoryPoolUnityTest`.

Function declaration:

```
void* pal_plat_osPoolCAlloc(palMemoryPoolID_t memoryPoolID);
```

Requirements:

Allocate a single memory block from a memory pool and set memory block to zero.

Related test:

`MemoryPoolUnityTest`.

Function declaration:

```
palStatus_t pal_plat_osPoolFree(palMemoryPoolID_t memoryPoolID,
void* block);
```

Requirements:

Return the `memoryPoolID` of the memory block back to a specific memory pool.

Related test:

`MemoryPoolUnityTest`.

Function declaration:

`palStatus_t pal_plat_osPoolDestroy(palMemoryPoolID_t* memoryPoolID);`

Requirements:

- Delete a memory pool object.
- All allocated memory will be deallocated. In success, `*memoryPoolID = NULL`.

Related test:

`MemoryPoolUnityTest`.

6.1.3.11 Message queue

PAL provides an API for `uint32_t` message queue which is allocated according to the messages count parameter.

Function declaration:

```
palStatus_t pal_plat_osMessageQueueCreate(uint32_t messageQCount,
palMessageQID_t* messageQID);
```

Requirements:

Create and initialize a message queue.

Related test:

`MessageUnityTest`.

Function declaration:

```
palStatus_t pal_plat_osMessagePut(palMessageQID_t messageQID,
uint32_t info, uint32_t timeout);
```

Requirements:

Put a message to a queue.

Related test:

`MessageUnityTest`.

Function declaration:

```
palStatus_t pal_plat_osMessageGet(palMessageQID_t messageQID,
uint32_t timeout, uint32_t* messageValue);
```

Requirements:

Get a message or wait for a message from a queue.

Related test:

`MessageUnityTest`.

Function declaration:

`palStatus_t pal_plat_osMessageQueueDestroy(palMessageQID_t* messageQID);`

Requirements:

Delete a message queue object.

Related test:

`MessageUnityTest`.

6.1.3.12 Atomic operations

Function declaration:

`int32_t pal_plat_osAtomicIncrement(int32_t* valuePtr, int32_t increment);`

Requirements:

Perform an atomic increment for a signed 32 bit value.

Related test:

`AtomicIncrementUnityTest`.

6.2 Update

6.2.1 Overview

The Update service is responsible for handling flash I/O operations as part of updating the binary image of the device.

It provides the following operations:

- Initialize.
- Write.
- Read.

The following APIs MUST be implemented by the platform side:

```
pal_plat_imageInitAPI
pal_plat_imageDeInit
pal_plat_imageGetMaxNumberOfImages
pal_plat_imageReserveSpace
pal_plat_imageSetHeader
pal_plat_imageWrite
pal_plat_imageSetVersion
pal_plat_imageFlush
pal_plat_imageGetDirectMemAccess
pal_plat_imageReadToBuffer
pal_plat_imageActivate
pal_plat_imageGetActiveHash
```

```
pal_plat_imageGetActiveVersion
pal_plat_imageWriteHashToMemory
```

6.2.2 Update types

The PAL Update data types are defined in the `pal_update.h` header file.

- `palImageId_t` holds the image ID number.
- `palImageHeaderDetails_t` is a struct containing 3 members that describe the basic image header parameters.
  - `imageSize` holds the image size.
  - `hash` is the `palBuffer` for the image hash.
  - `version` holds the version number of the image.
- `palImagePlatformData_t` is an enumerator for the `pal_imageWriteDataToMemory` API, please see below.
- `palImageEvents_t` is an enumerator for the update callback function. Each API must call the PAL update callback function with the appropriate value from this enumerator.
- `palImageSignalEvent_t` is a pointer function type for the PAL update callback function.

```
typedef uint32_t palImageId_t;

typedef struct _palImageHeaderDetails_t
{
                size_t imageSize;
                palBuffer_t hash;
                uint64_t version;
}palImageHeaderDetails_t;


typedef enum _palImagePlatformData_t
{
    PAL_IMAGE_DATA_FIRST = 0,
    PAL_IMAGE_DATA_HASH = PAL_IMAGE_DATA_FIRST,
    PAL_IMAGE_DATA_LAST
}palImagePlatformData_t;

typedef enum _palImageEvents_t
{
        PAL_IMAGE_EVENT_FIRST = -1,
        PAL_IMAGE_EVENT_ERROR = PAL_IMAGE_EVENT_FIRST,
        PAL_IMAGE_EVENT_INIT ,
        PAL_IMAGE_EVENT_PREPARE,
        PAL_IMAGE_EVENT_WRITE,
        PAL_IMAGE_EVENT_FINALIZE,
        PAL_IMAGE_EVENT_READTOBUFFER,
        PAL_IMAGE_EVENT_ACTIVATE,
        PAL_IMAGE_EVENT_GETACTIVEHASH,
        PAL_IMAGE_EVENT_GETACTIVEVERSION,
        PAL_IMAGE_EVENT_WRITEDATATOMEMORY,
        PAL_IMAGE_EVENT_LAST
} palImageEvents_t;

typedef void (*palImageSignalEvent_t)( palImageEvents_t event);
```

6.2.3 API

This section defines the requirements for each update API. It also includes tips and warnings to save time when porting. The requirements cannot replace the documentation found in the `pal_plat_update.h`.

For each API, there is information about:

- The exact declaration.
- Requirements.
- Special notes.
- Related tests in PAL test code.

6.2.3.1 Initialization

Declaration

`palStatus_t pal_plat_imageInitAPI(palImageSignalEvent_t CBfunction);`

Requirements:

This function sets the Update service callback function.

Related test:

All PAL update tests.

6.2.3.2 De-initialization

Declaration

`palStatus_t pal_plat_imageDeInit(void);`

Requirements:

This function releases all resources allocated by the PAL update code.

Related Test:

All PAL update tests.

6.2.3.3 Get max number of images

Declaration

```
palStatus_t pal_plat_imageGetMaxNumberOfImages(uint8_t*
imageNumber);
```

Requirements:

This function returns, in `imageNumber` output parameter, the max number of images that can be written to the device.

Related test:

Not tested – for future use only.

6.2.3.4 Reserve Space

Declaration

```
palStatus_t pal_plat_imageReserveSpace(palImageId_t imageId, size_t
imageSize);
```

Requirements:

This function reserves the flash memory for writing the image.

Related test:

All PAL update tests.

6.2.3.5 Set Header

Declaration

```
palStatus_t pal_plat_imageSetHeader(palImageId_t
imageId,palImageHeaderDeails_t* details);
```

Requirements:

This function sets the header of an image.

Related test:

All PAL update tests.

6.2.3.6 Write

Declaration

```
palStatus_t pal_plat_imageWrite(palImageId_t imageId, size_t offset,
palConstBuffer_t*  chunk);
```

Requirements:

This function writes the data stored in `chunk` at the specified offset.

Related test:

All PAL update tests.

6.2.3.7 Set version

Declaration

```
palStatus_t pal_plat_imageSetVersion(palImageId_t imageId, const
palConstBuffer_t* version);
```

Requirements:

This function sets the version of an image.

Related test:

All PAL update tests.

6.2.3.8 Memory flush

Declaration

`palStatus_t pal_plat_imageFlush(palImageId_t imageId);`

Requirements:

This function flushes the memory that was previously written for the selected image.

Related test:

All PAL update tests.

6.2.3.9 Get direct memory access

Declaration

```
palStatus_t pal_plat_imageGetDirectMemAccess(palImageId_t imageId,
void** imagePtr, size_t* imageSizeInBytes);
```

Requirements:

This function retrieves a pointer to the location of the image in a readable memory segment (if supported by the platform).

Related test:

Not tested – For future use only.

6.2.3.10 Read to buffer

Declaration

```
palStatus_t pal_plat_imageReadToBuffer(palImageId_t imageId, size_t
offset, palBuffer_t* chunk);
```

Requirements:

This function reads a chunk of data from an image and copies it to the `chunk` buffer.

Related test:

All PAL update tests.

6.2.3.11 Activate image

Declaration

`palStatus_t pal_plat_imageActivate(palImageId_t imageId);`

Requirements:

This function sets an image with the selected ID to be the active image loaded after platform reboot.

Related test:

- Not tested – For future use only.

6.2.3.12 Get active image hash

Declaration

```
palStatus_t pal_plat_imageGetActiveHash(palBuffer_t* hash);
```

Requirements:

This function retrieves the hash value of the active image.

Related test:

Not tested.

6.2.3.13 Get active image version

Declaration

```
palStatus_t pal_plat_imageGetActiveVersion (palBuffer_t* version);
```

Requirements:

This function retrieves the version value of the active image.

Related test:

Not tested – For future use only.

6.2.3.14 Write image hash

Declaration

```
palStatus_t pal_plat_imageWriteHashToMemory(const palConstBuffer_t*
const hashValue);
```

Requirements:

This function writes the hash value of the selected image to a location accessible by the bootloader of the platform.

Related test:

Not tested – For future use only.

6.3 Networking

6.3.1 Overview

The Networking module is responsible for providing socket based networking functionality. This includes creating and communicating using TCP and UDP sockets. Support for client and server modes.

The API is similar to the POSIX networking API with several modifications to make it more portable and meet the service requirements.

The following APIs MUST be implemented by the platform side:

```
pal_plat_socketsInit
pal_plat_socketsTerminate
pal_plat_RegisterNetworkInterface
pal_plat_getNumberOfNetInterfaces
pal_plat_getNetInterfaceInfo
pal_plat_socket
pal_plat_asynchronousSocket
pal_plat_getSocketOptions
pal_plat_setSocketOptions
pal_plat_bind
pal_plat_receiveFrom
pal_plat_sendTo
pal_plat_close
pal_plat_listen
pal_plat_connect
pal_plat_accept
pal_plat_recv
pal_plat_send
pal_plat_socketMiniSelect
pal_plat_getAddressInfo
```

6.3.2 Netwoking types

The Networking data types are defined in the `pal_network.h` header file.

6.3.2.1 Types

- `palSocketLength_t` is used to hold buffer lengths.
- `palSocket_t` is designed to hide the underlying socket definition.
- `palIpV4Addr_t` holds the IPv4 raw address bytes.
- `palIpV6Addr_t` holds the IPv6 raw address bytes.

```
typedef uint32_t palSocketLength_t;  /*! length of data */
typedef void* palSocket_t;           /*! PAL socket handle type */
typedef uint8_t palIpV4Addr_t[PAL_IPV4_ADDRESS_SIZE];
typedef uint8_t palIpV6Addr_t[PAL_IPV6_ADDRESS_SIZE];
```

6.3.2.2 Enumerations

- palSocketDomain_t lists the supported socket domains (IPV4/IPv6).
- palSocketType_t lists the supported socket types (TCP client/TCP server/UDP socket).
- palSocketOptionName_t lists the supported socket options.

```
typedef enum {
   PAL_AF_UNSPEC = 0,
   PAL_AF_INET = 2,      /*! Internet IP Protocol  */
   PAL_AF_INET6 = 10,    /*! IP version 6          */
} palSocketDomain_t;     /*! network domains supported by PAL*/

typedef enum {
#if PAL_NET_TCP_AND_TLS_SUPPORT
   PAL_SOCK_STREAM = 1,              /*! stream socket   */
   PAL_SOCK_STREAM_SERVER = 99,      /*! stream socket   */
#endif //PAL_NET_TCP_AND_TLS_SUPPORT
   PAL_SOCK_DGRAM = 2                /*! datagram socket       */
} palSocketType_t;                   /*! socket types supported by PAL */

typedef enum {
   PAL_SO_REUSEADDR = 0x0004,      /*! allow local address reuse */
#if PAL_NET_TCP_AND_TLS_SUPPORT // socket options below supported only if TCP
is supported.
   PAL_SO_KEEPALIVE = 0x0008,       /*! keep TCP connection open even if idle
using periodic messages*/
#endif //PAL_NET_TCP_AND_TLS_SUPPORT
   PAL_SO_SNDTIMEO = 0x1005,        /*! send timeout */
   PAL_SO_RCVTIMEO = 0x1006,        /*! receive timeout */
} palSocketOptionName_t;            /*! socket options supported by PAL */
```

6.3.2.3 Structures

- palSocketAddress_t is used for IP addresses. The same external structure is used for both IPv4 and IPv6 for convenience.

NOTE! The values in this structure should not be manually manipulated. Use the functions defined in pal_network.h for this (for example, pal_setSockAddrPort or pal_setSockAddrIPV4Addr).

- palNetInterfaceInfo_t is a structure used to return information regarding a specific network interface.

NOTE! Not all fields may be relevant for all interfaces on all platforms, but the address field is mandatory.

- pal_timeVal_t is the time value used to specify timeouts.

```
typedef struct palSocketAddress {
   unsigned short    addressType;     /*! address family for the socket*/
   char              addressData[PAL_NET_MAX_ADDR_SIZE];   /*! address
(based on protocol)*/
} palSocketAddress_t; /*! address data structure with enough room to support
IPV4 and IPV6*/
```

```
typedef struct palNetInterfaceInfo{
   char interfaceName[16]; //15 + '\0'
   palSocketAddress_t address;
   uint32_t addressSize;
} palNetInterfaceInfo_t;

typedef struct pal_timeVal{
   int32_t    pal_tv_sec;      /*! seconds */
   int32_t    pal_tv_usec;     /*! microseconds */
} pal_timeVal_t;
```

6.3.2.4 Defines

The networking module uses two types of definitions:

- Compile time configuration options defined in `pal_configuration.h`:
    a. `PAL_NET_TCP_AND_TLS_SUPPORT` enables support for TCP. Without this define, all TCP related APIs are not supported.
    b. `PAL_NET_ASYNCHRONOUS_SOCKET_API` enables support for asynchronous sockets.
    c. `PAL_NET_DNS_SUPPORT` enables DNS name resolution support.
- **NOTE!** Currently, all three configuration options must be enabled for services to function correctly (this is the default setting in `pal_configuration.h`).
- Defined values used by the implementation:
    a. `PAL_NET_MAX_ADDR_SIZE` is the max address size in bytes.
    b. `PAL_NET_DEFAULT_INTERFACE` indicates the default network interface.
    c. `PAL_IPV4_ADDRESS_SIZE` is the IPv4 address size in bytes.
    d. `PAL_IPV6_ADDRESS_SIZE` is the IPv6 address size in bytes.
    e. `PAL_NET_SOCKET_SELECT_MAX_SOCKETS` is the max number of sockets supported by the PAL `select` function.
    f. `PAL_NET_SOCKET_SELECT_RX_BIT/` `PAL_NET_SOCKET_SELECT_TX_BIT/` `PAL_NET_SOCKET_SELECT_ERR_BIT` is the internal bit structure of the socket status returned by select.

```
// internal defnitions
#define PAL_NET_MAX_ADDR_SIZE 32
#define PAL_NET_DEFAULT_INTERFACE 0xFFFFFFFF
#define PAL_IPV4_ADDRESS_SIZE 4
#define PAL_IPV6_ADDRESS_SIZE 16
#define PAL_NET_SOCKET_SELECT_MAX_SOCKETS 8
#define PAL_NET_SOCKET_SELECT_RX_BIT (1)
#define PAL_NET_SOCKET_SELECT_TX_BIT (2)
#define PAL_NET_SOCKET_SELECT_ERR_BIT (4)
// external defintions from pal_configuration.h
#define PAL_NET_TCP_AND_TLS_SUPPORT = 1
```

```
#define PAL_NET_ASYNCHRONOUS_SOCKET_API = 1
#define PAL_NET_DNS_SUPPORT = 1
```

6.3.2.5 Macros

- `PAL_NET_SELECT_IS_RX` checks if the socket has data available for retrieval.
- `PAL_NET_SELECT_IS_TX` checks if the socket is ready for sending data.
- `PAL_NET_SELECT_IS_ERR` checks if the socket returned an error.

```
#define PAL_NET_SELECT_IS_RX(socketStatus, index) ((socketStatus[index] |
PAL_NET_SOCKET_SELECT_RX_BIT) != 0) /*! check if RX bit is set in select result
for a given socket index*/
#define PAL_NET_SELECT_IS_TX(socketStatus, index) ((socketStatus[index] |
PAL_NET_SOCKET_SELECT_TX_BIT) != 0) /*! check if TX bit is set in select result
for a given socket index*/
#define PAL_NET_SELECT_IS_ERR(socketStatus, index)
   ((socketStatus[index] | PAL_NET_SOCKET_SELECT_ERR_BIT) != 0) /*! check if
ERR bit is set in select result for a given socket index*/
```

6.3.2.6 Function pointers

`palAsyncSocketCallback_t` is a callback function for asynchronous sockets. It must be called when an event happens to an asynchronous socket. The most important events are data arrived and connection closed.

```
typedef void(*palAsyncSocketCallback_t)(void)
```

6.3.3 API

This section defines the requirements for each Networking API. Tt also includes tips and warnings to save time when porting. The requirements cannot replace the documentation found in the `pal_plat_network.h`.

For each API, there is information about:

- The exact declaration.
- Requirements.
- Special notes.
- Related tests in PAL test code.

6.3.3.1 Introduction

The API is similar to the POSIX networking API with several modifications to make it more portable and meet the service requirements.

The main changes are:

- `errno` is not used as part of the API. Functions now return a status explicitly as the return value (as a result, some parameters are now passed as output arguments of functions, instead of being returned by function as a value).

- Reductions to supported functionality have been made to simplify porting (for example, reduced supported socket options/socket types).
- Sockets are bound to a particular interface on creation.

The function names are similar to their POSIX counterparts, so anyone familiar with the POSIX API can recognize the basic functionality.

6.3.3.2 Initialization and termination

The following functions are called upon socket initialization and termination. If any initialization process is needed at the platform level, it should be placed here.

Function declaration:

```
palStatus_t pal_plat_socketsInit(void* context)
```

Requirements:

- Return the status of the initialization.
- Perform any steps needed for initialization at the platform level.

NOTE! If not necessary, please ignore the argument. If some external data is required for initialization, it is passed in through this parameter.

Related test:

```
socketUDPCreationOptionsTest
```

Function declaration:

```
palStatus_t pal_plat_socketsTerminate(void* context)
```

Requirements:

- Return the status of the termination.
- Perform any steps needed for termination at the platform level.

NOTE! If not necessary, please ignore the argument. If some external data is required for initialization, it is passed in through this parameter.

Related test:

```
socketUDPCreationOptionsTest
```

6.3.3.3 Network interfaces

The PAL network API is designed to allow the services to use more than one network interface. The basic idea is the following:

- The available network interfaces will be enumerated 0 to N in one of the following ways (one of these must be implemented):
    - The service must register network interfaces using the `pal_plat_RegisterNetworkInterface` function.
    - The initialization function can automatically enumerate the interfaces in systems where this is possible (such as Linux) and populate the network interfaces enumeration.
- Once the interfaces are enumerated the following functionality needs to be available:
    - Get the number of enumerated interfaces.
    - Get information regarding an interface given its index (specifically important is retrieving the IP address).
    - When creating a socket, an interface number must be specified. The socket created must be bound to this interface and use it for incoming and outgoing communication.

NOTE! `PAL_NET_DEFAULT_INTERFACE` defines the default interface. You can select it based on the platform.

Function declaration:

```
palStatus_t pal_plat_RegisterNetworkInterface(void*
networkInterfaceContext, uint32_t* interfaceIndex)
```

Requirements:

- Return the status of the registration and the index assigned to the registered network interface in case of success.
- The registered interface must be added to the count of registered interfaces returned by the `pal_plat_getNumberOfNetInterfaces` function. It provides information regarding the interface (most important is the IP address) when queried by the `pal_plat_getNetInterfaceInfo` function.
- The socket created with the `interfaceIndex` of a registered interface uses this interface for incoming and outgoing communications.

Related test:

`socketUDPCreationOptionsTest`

Function declaration:

```
palStatus_t pal_plat_getNumberOfNetInterfaces(uint32_t*
numInterfaces)
```

Requirements:

Return the number of enumerated network interfaces in `numInterfaces`.

Related test:

`socketUDPCreationOptionsTest`

Function declaration:

```
palStatus_t pal_plat_getNetInterfaceInfo(uint32_t interfaceNum,
palNetInterfaceInfo_t* interfaceInfo)
```

Requirements:

- Return in `palStatus_t` the status of the query for interface information.
- Return the information regarding the interface at the given index through the `interfaceInfo` output parameter.

NOTE! The address field in the `palNetInterfaceInfo_t` structure is mandatory.

Related test:

- `socketUDPCreationOptionsTest`
- `basicSocketScenario5`

6.3.3.4 Socket creation and destruction

There are two different socket creation functions in the PAL API. One for normal (synchronous) socket and one for asynchronous sockets. Asynchronous sockets support an asynchronous callback that is called whenever an event happens to the socket.

Supported socket configuration parameters:

- IPv4 or IPv6.
- TCP (Client)/TCP (Server)/UDP .
- Asynchronous callback support.
- Blocking/non-blocking.

Function declaration:

```
palStatus_t pal_plat_socket(palSocketDomain_t domain,
palSocketType_t type, bool nonBlockingSocket, uint32_t interfaceNum,
palSocket_t* socket)
```

Requirements:

- Return the status of the socket creation.
- The socket created is returned in the output socket parameter.
- The socket created is set to blocking upon creation unless the `nonBlockingSocket` parameter is set to true, in which case it is set to non-blocking.

- The socket created is bound to the interface provided by the `interfaceNum` parameter, and all incoming and outgoing traffic use this interface.

Related test:

- `socketUDPCreationOptionsTest`
- `basicUDPclientSendRecieve`
- `basicTCPclientSendRecieve`
- `basicSocketScenario3`
- `basicSocketScenario4`
- `basicSocketScenario5`

Function declaration:

```
palStatus_t pal_plat_asynchronousSocket(palSocketDomain_t domain,
palSocketType_t type, bool nonBlockingSocket, uint32_t interfaceNum,
palAsyncSocketCallback_t callback, palSocket_t* socket)
```

Requirements:

- Return the status of the socket creation.
- The socket created is returned in the output `socket` parameter.
- The socket created is set to blocking upon creation unless the `nonBlockingSocket` is set to true, in which case it is set to non-blocking.
- The socket created is bound to the interface provided by the `interfaceNum` and all incoming and outgoing traffic use this interface.
- When an event happens to the socket after creation, the provided `callback` function is called asynchronously.

NOTE! Mandatory events for callback triggering: data received and connection closed. Additional events may optionally be supported.

Related test:

- `socketUDPCreationOptionsTest`
- `basicSocketScenario3`

Function declaration:

```
palStatus_t pal_plat_close(palSocket_t* socket)
```

Requirements:

- Return the status of the socket destruction.
- The socket data that the `socket` parameter holds is cleared.

Related test:

- `socketUDPCreationOptionsTest`
- `basicUDPclientSendRecieve`
- `basicTCPclientSendRecieve`
- `basicSocketScenario3`
- `basicSocketScenario4`
- `basicSocketScenario5`

6.3.3.5 Socket options

The PAL network API requires a number of socket options to be supported (see `palSocketOptionName_t`) via the following set and get functions.

Function declaration:

```
palStatus_t pal_plat_setSocketOptions(palSocket_t socket, int
optionName, const void* optionValue, palSocketLength_t optionLength)
```

Requirements:

- Return the status of the operation.
- Set the socket option value provided via the `optionValue` parameter.

NOTE! The format of the `optionValue` buffer is different based on the `optionName` selected.

Related test:

- `socketUDPCreationOptionsTest`
- `tProvUDPTest`

Function declaration:

```
palStatus_t pal_plat_getSocketOptions(palSocket_t socket,
palSocketOptionName_t optionName, void* optionValue,
palSocketLength_t* optionLength)
```

Requirements:

- Return the status of the operation.
- Return the value for the requested socket option (`optionName`) in the output buffer (`optionValue`) and its length in the parameter (`optionLength`).

NOTE! The format of the `optionValue` buffer is different based on the option value selected.

Related test:

`socketUDPCreationOptionsTest`

6.3.3.6 General APIs

This APIs serve both TCP and UDP sockets. The first API allows to bind to an incoming address. The second API allows to listen to multiple sockets simultaneously.

Function declaration:

```
palStatus_t pal_plat_bind(palSocket_t socket, palSocketAddress_t*
myAddress, palSocketLength_t addressLength);
```

Requirements:

- Return the status of the operation.
- Bind the socket to the given address and port (fail otherwise).

NOTE! The address should be the same as the address of the interface the socket was bound to on creation or 0.0.0.0 .

Related test:

`basicSocketScenario5`

Function declaration:

```
palStatus_t pal_plat_socketMiniSelect(const palSocket_t
socketsToCheck[PAL_NET_SOCKET_SELECT_MAX_SOCKETS], uint32_t
numberOfSockets, pal_timeVal_t* timeout, uint8_t
palSocketStatus[PAL_NET_SOCKET_SELECT_MAX_SOCKETS], uint32_t *
numberOfSocketsSet)
```

Requirements:

- Return the status of the operation.
- Blocks for up to `timeout` time or until one or more events happen to any of the sockets provided in the `socketsToCheck` array. Returns the status of every socket in the output `palSocketStatus` array and the number of sockets that triggered an event in the `numberOfSocketsSet` parameter.
- This is equivalent to calling select on all the sockets with the events `RX` and `ERR set`.

NOTE! If `TX ready` event happens constantly (like in Linux), please ignore this event and check only the RX and ERR flags.

For example, to learn how to set the flags in the socket status, see the `PAL_NET_SELECT_IS_RX` macro definition.

Related test:

`basicSocketScenario4`

6.3.3.7 UDP APIs

These APIs are intended for use with UDP sockets only.

UDP sockets support two main APIs (in addition to bind) for UDP communication:

- `pal_plat_sendTo`
- `pal_plat_receiveFrom`

Function declaration:

```
palStatus_t pal_plat_sendTo(palSocket_t socket, const void* buffer,
size_t length, const palSocketAddress_t* to, palSocketLength_t
toLength, size_t* bytesSent)
```

Requirements:

- Return the status of the operation.
- Send the `buffer` of `length` bytes to the destination address provided in `to` or fail otherwise. If the socket is blocking, this function should block while waiting for data to be written to the outbound buffer. If the socket is non-blocking and the outbound buffer is not available (full), the function should return `PAL_ERR_SOCKET_WOULD_BLOCK`.
- Return the number of bytes actually sent in the `bytesSent` output parameter.

NOTE! The socket should use the network interface it was bound to perform the sending. Implementation may impose a limit on the max payload size and return the appropriate error code if the data size is too big.

Related test:

- `basicUDPclientSendRecieve`
- `tProvUDPTest`

Function declaration:

```
palStatus_t pal_plat_receiveFrom(palSocket_t socket, void* buffer,
size_t length, palSocketAddress_t* from, palSocketLength_t*
fromLength, size_t* bytesReceived)
```

Requirements:

- Return the status of the operation.
- Receive data from the given UDP socket into the given `buffer` of up to `length` bytes or fail. If the socket is blocking, this function should block waiting for data. If the socket is non-blocking and data is not available, the function should return `PAL_ERR_SOCKET_WOULD_BLOCK`.

- Return the address of the sender to the provided output argument (`from`).
- Return the number of bytes received in the `bytesReceived` output parameter.

NOTE! The socket should use the network interface it was bound to perform the receiving.

Related Test:

- `basicUDPclientSendRecieve`
- `tProvUDPTest`

6.3.3.8 TCP APIs

These APIs are intended for use with TCP sockets only.

Function declaration:

`palStatus_t pal_plat_recv(palSocket_t socket, void* buf, size_t len, size_t* recievedDataSize)`

Requirements:

- Return the status of the operation.
- Receive data from the given TCP socket into the given `buffer` of up to `length` bytes or fail. If the socket is blocking, this function should block waiting for data. If the socket is non-blocking and data is not available, the function should return `PAL_ERR_SOCKET_WOULD_BLOCK`.
- Return the number of bytes received in the `recievedDataSize` output parameter.

NOTE! The socket should use the network interface it was bound to perform the receiving.

Related Test:

- `basicTCPclientSendRecieve`
- `basicSocketScenario5`
- `basicSocketScenario4`
- `basicSocketScenario3`

Function declaration:

`palStatus_t pal_plat_send(palSocket_t socket, const void* buf, size_t len, size_t* sentDataSize)`

Requirements:

- Return the status of the operation.

- Send the given `buffer` of up to `len` bytes using the given TCP socket or fail. If the socket is blocking, this function should block waiting for data. If the socket is non-blocking and the outbound buffer is not available (full), the function should return `PAL_ERR_SOCKET_WOULD_BLOCK`.
- Return the number of bytes sent in the `sentDataSize` output parameter.

NOTE! The socket should use the network interface it was bound to perform the sending. Implementation may impose a limit on the max payload size and return the appropriate error code if the data size is too big.

Related test:

- `basicTCPclientSendRecieve`
- `basicSocketScenario5`
- `basicSocketScenario4`
- `basicSocketScenario3`

6.3.3.8.1 Client APIs

The APIs used by TCP clients only.

Function declaration:

```
palStatus_t pal_plat_connect(palSocket_t socket, const
palSocketAddress_t* address, palSocketLength_t addressLen)
```

Requirements:

- Return the status of the operation.
- Connect the given TCP socket to the given `address` or fail. If the socket is blocking, this function should block waiting for connection. If the socket is non-blocking and the connection establishment takes a significant amount time the function should return `PAL_ERR_SOCKET_WOULD_BLOCK`.

Related test:

- `basicTCPclientSendRecieve`
- `basicSocketScenario4`
- `basicSocketScenario3`

6.3.3.8.2 Server APIs

The APIs used by TCP servers only.

The expected flow is:

Create socket -> (bind) -> listen -> accept

Function declaration:

`palStatus_t pal_plat_listen(palSocket_t socket, int backlog)`

Requirements:

- Return the status of the operation.
- Instruct the operating system/network stack that we are now listening to the given socket.

Related test:

`basicSocketScenario5`

Function declaration:

`palStatus_t pal_plat_accept(palSocket_t socket, palSocketAddress_t* address, palSocketLength_t* addressLen, palSocket_t* acceptedSocket)`

Requirements:

- Return the status of the operation.
- Accept an incoming connection and return the new TCP client socket that will be used to communicate using the established connection. If the socket is blocking, this function should block waiting for incoming connection. If the socket is non-blocking and there is no incoming connection the function should return `PAL_ERR_SOCKET_WOULD_BLOCK`.
- If address output parameter is not NULL, return the incoming connection's source address in the `address` output parameter.

Related Test:

`basicSocketScenario5`

6.3.3.9 Name resolution

An API intended for name resolution and conversion from string addresses (hostname or IP) to address structures usable by other functions.

Function declaration:

`palStatus_t pal_plat_getAddressInfo(const char* url, palSocketAddress_t* address, palSocketLength_t* addressLength)`

Requirements:

- Return the status of the operation.
- Decode the given string (expected to be a hostname or IP address) into an address structure. If a host name is given a DNS query is performed to fetch the IP address corresponding to the given host name.
- Return the decoded address in the `address` output parameter.

Related test:

- `basicTCPclientSendRecieve`
- `basicSocketScenario5`
- `basicSocketScenario4`
- `basicSocketScenario3`
- `tProvUDPTest`
- `basicUDPclientSendRecieve`

7 How to build PAL

Currently, PAL does not supply its own build system. PAL is built as part of the mbed Cloud Client solution.

For instructions on how to build PAL tests, please refer to the PAL tests section.

8 PAL tests

As a part of PAL code, PAL provides a Unity based test framework to test the existing APIs. This framework is located in the following folder:

```
{PAL}/Test/
```

8.1 Tests source code

PAL tests source code is located in the following folder:

```
{PAL}/Test/Unitest/
```

For each module, there is a test source file, test runner and test main files for every supported OS.

For example:

- `pal_rtos_test.c`
- `pal_rtos_test_main_mbedOS.cpp`
- `pal_rtos_test_runner.c`

8.2 Tests scenarios

This section describes the PAL tests scenarios, including the list of prerequisite APIs needed to run each test.

8.2.1 RTOS

8.2.1.1 pal_osKernelSysTick_Unity

Description:

Sanity check of the kernel system tick API.

Fails if two calls return the same `sysTick` value.

Prerequisite:

`pal_plat_osKernelSysTick`

8.2.1.2 pal_osKernelSysTick64_Unity

Description:

Sanity check of the kernel 64bit system tick API.

Fails if two calls return the same `sysTick` value.

Prerequisite:

`pal_plat_osKernelSysTick64`

8.2.1.3 pal_osKernelSysTickMicroSec_Unity

Description:

Check conversion from non-zero `sysTick` value to micro seconds.

Verify that the result is different from 0.

Prerequisite:

`pal_osKernelSysTickMicroSec`

8.2.1.4 pal_osKernelSysMilliSecTick_Unity

Description:

Sanity check of non-zero values conversions between micro seconds to ticks to milliseconds.

Verify that the result correct when converting the input (micro seconds) to the test output (milliseconds).

Prerequisite:

- `pal_osKernelSysTickMicroSec`
- `pal_osKernelSysMilliSecTick`

8.2.1.5 pal_osKernelSysTickFrequency_Unity

Description:

Verify that the tick frequency function returns non-zero value.

Prerequisite:

`pal_osKernelSysTickFrequency`

8.2.1.6 pal_osDelay_Unity

Description:

Sanity check for Delay API, verifying that `sysTick` increments after delay.

Prerequisite:

- `pal_osKernelSysTick`
- `pal_osDelay`

8.2.1.7 BasicTimeScenario

Description:

Test for basic timing scenarios based on calls for the ticks and delay functionality while verifying that results meet the defined deltas.

Prerequisite:

- `pal_osKernelSysTick`
- `pal_osDelay`
- `pal_osKernelSysTickMicroSec`

8.2.1.8 TimerUnityTest

Description:

This test creates two timers, periodic and one shot, then starts them. Afterwards, causes a delay to allow print output from the timer functions on the console.

Prerequisite:

- `pal_init`
- `pal_osTimerCreate`
- `pal_osKernelSysTick`
- `pal_osTimerStart`
- `pal_osTimerStop`
- `pal_osTimerDelete`
- `pal_osDelay`

8.2.1.9 PrimitivesUnityTest1

Description:

The test creates mutexes and semaphores and uses them to communicate between the different threads created by the test and defined in `pal_rtos_test_utils.c`.

In this test, we check that thread communication is working as expected between the threads and in the designed order.

In one case, we expect that thread will fail to lock a mutex – (thread1).

Threads are created with different priorities (PAL enforces this attribute).

For each case, the thread function prints the expected result. The test code verifies this result as well.

Prerequisite:

- `pal_init`
- `pal_osMutexCreate`
- `pal_osSemaphoreCreate`
- `pal_osThreadCreate`
- `pal_osMutexWait`
- `pal_osMutexRelease`
- `pal_osSemaphoreRelease`
- `pal_osMutexDelete`
- `pal_osSemaphoreDelete`
- `pal_osDelay`

8.2.1.10 PrimitivesUnityTest2

Description:

This test verifies several RTOS primitives APIs in the aspect of invalid arguments handling. The test calls each API with invalid arguments and verifies the result. It also verifies that semaphore wait API can accept NULL as 3rd parameter.

Prerequisite:

- `pal_osThreadCreate`
- `pal_osSemaphoreCreate`
- `pal_osSemaphoreDelete`
- `pal_osSemaphoreWait`
- `pal_osSemaphoreRelease`

8.2.1.11 MemoryPoolUnityTest

Description:

This test does the following:

1. Creates two memory pools.
2. Allocates blocks from each pool using `pal_osPoolAlloc` and `pal_osPoolCAlloc` APIs.
3. Verifies that none of the allocated blocks is NULL.
4. Deallocates the blocks.
5. Destroys the pools.

Prerequisite:

- `pal_init`
- `pal_osPoolCreate`
- `pal_osPoolAlloc`
- `pal_osPoolCAlloc`
- `pal_osPoolFree`
- `pal_osPoolDelete`

8.2.1.12 MessageUnityTest

Description:

This test does the following:

1. Creates a message queue.
2. Puts a message.
3. Reads the message from the queue.

4. Verifies that the message has the expected value.

Prerequisite:

- `pal_init`
- `pal_osMessageQueueCreate`
- `pal_osMessagePut`
- `pal_ osMessageGet`
- `pal_ osMessageQueueDestroy`

8.2.1.13 AtomicIncrementUnityTest

Description:

This test performs a single atomic increment call to an integer value and verifies that the result is as expected.

Prerequisite:

- `pal_init`
- `pal_osAtomicIncrement`

8.2.2 Update

8.2.2.1 pal_update_start

Description:

Sanity initialization test for the update API – should always pass.

Prerequisite:

`None`

8.2.2.2 pal_update_4k

Description:

Write an image data with the size of 4KB to the device flash in one write operation. Read the image data in a single read operation.

Test passes if the read data and written data is the same.

Prerequisite:

- `pal_imageInitAPI`
- `pal_imagePrepare`
- `pal_imageWrite`
- `pal_imageFinalize`
- `pal_imageReadToBuffer`

8.2.2.3 pal_update_read

Description:

Write an image data with the size of 1500 bytes (unaligned data size) to the device flash in one write operation. Create a read buffer with much smaller size (300 bytes). Perform data read iterations of the previously written data, until finished.

Test passes if the overall read data and written data is the same.

Prerequisite:

- `pal_imageInitAPI`
- `pal_imagePrepare`
- `pal_imageWrite`
- `pal_imageFinalize`
- `pal_imageReadToBuffer`

8.2.2.4 pal_update_4k_write_1k_4_times

Description:

Write an image data with the overall size of 4KB to the device flash in four partial write operations. Read the image data with a single read operation.

Test passes if the read data and overall written data is the same.

Prerequisite:

- `pal_imageInitAPI`
- `pal_imagePrepare`
- `pal_imageWrite`
- `pal_imageFinalize`
- `pal_imageReadToBuffer`

8.2.3 Networking

8.2.3.1 socketUDPCreationOptionsTest

Description:

This is a mixed test that validates the basic networking functionality. It does the following:

1. Registers the same network interface twice and expects to receive the same index.
2. Creates different types of IPV4 sockets (blocking/non-blocking, TCP/UDP, synchronous/asynchronous).
3. Checks that the socket creation was successful.

4. Queries the current number of the network interfaces.
5. Checks that the number of network interfaces is correct.
6. Verifies that it can successfully get information for this network adapter.
7. Sets a socket option for one of the sockets.
8. Closes all sockets.

Prerequisite:

- `pal_socket`
- `pal_asynchronousSocket`
- `pal_registerNetworkInterface`
- `pal_getNetInterfaceInfo`
- `pal_getNumberOfNetInterfaces`
- `pal_setSocketOptions`
- `pal_close`

8.2.3.2 basicTCPclientSendRecieve

Description:

This is a test of basic TCP connectivity. It does the following:

1. Creates a blocking synchronous client TCP socket.
2. Performs a lookup query for the IP address of the test HTTP server.
3. Tries to connect to this server.
4. When connected, sends the server a basic HTTP query.
5. Receives the response.
6. Checks that it is a valid HTTP response.
7. Closes the previously created socket.

Prerequisite:

- `pal_socket`
- `pal_getAddressInfo`
- `pal_connect`
- `pal_send`
- `pal_recv`
- `pal_close`

8.2.3.3 basicUDPclientSendRecieve

Description:

This is a test of basic UDP connectivity. It does the following:

1. Creates a blocking synchronous UDP socket.

2. Performs a lookup query for the IP address of the test DNS server.
3. Sends the server a basic DNS query.
4. Receives the response.
5. Checks that it is valid.
6. Closes the socket.

Prerequisite:

- `pal_socket()`
- `pal_getAddressInfo()`
- `pal_connect()`
- `pal_sendTo()`
- `pal_receiveFrom ()`
- `pal_close()`

8.2.3.4 basicSocketScenario3

Description:

This is a basic asynchronous socket test. It does the following:

1. Creates a blocking asynchronous TCP socket.
2. Performs a lookup query for the IP address of the test HTTP server.
3. Sends the server a basic HTTP query.
4. Receives the response.
5. Checks that it is valid and that the socket got a call to the asynchronous socket callback function.
6. Closes the socket.

Prerequisite:

- `pal_socket`
- `pal_getAddressInfo`
- `pal_connect`
- `pal_send`
- `pal_recv`
- `pal_close`
- `pal_osSemaphoreCreate`
- `pal_osSemaphoreCreate`
- `pal_osSemaphoreWait`
- `pal_osSemaphoreDelete`

8.2.3.5 basicSocketScenario4

<u>Description:</u>

This is a basic test of the `pal_socketMiniSelect` function. It does the following:

1. Creates two blocking synchronous TCP sockets.
2. Performs a lookup query for the IP address of the test HTTP server.
3. Connects the first socket to the test server.
4. Sends the server a basic HTTP request.
5. Calls the `select` function on both sockets.
6. Checks that the socket connected to the server is signaled because it has data ready to read.
7. Receives the response.
8. Checks that it is valid.
9. Connects the second socket to the server (but no data is being sent to the server).
10. Calls the `select` function again only for the second socket.
11. Verifies that the `select` function times out and doesn't return any readable sockets.
12. Closes both sockets.

<u>Prerequisite:</u>

- `pal_socket`
- `pal_getAddressInfo`
- `pal_connect`
- `pal_send`
- `pal_recv`
- `pal_close`
- `pal_socketMiniSelect`

8.2.3.6 basicSocketScenario5

<u>Description:</u>

This is a basic test of the TCP server socket. It does the following:

1. Creates a blocking synchronous TCP server socket.
2. Binds the socket to port and the local address.
3. Calls the `listen` function.
4. Calls `accept` to wait for an incoming connection. At this point an external device imitates a connection to the server (and sends a HTTP query).
5. Checks that the connection was accepted correctly.

6. Reads an incoming request.
7. Sends the external connection its response.
8. Closes all sockets.

Prerequisite:

- `pal_socket`
- `pal_getAddressInfo`
- `pal_bind`
- `pal_send`
- `pal_recv`
- `pal_listen`
- `pal_accept`
- `pal_close`

8.2.3.7 tProvUDPTest

Description:

This is a test of basic UDP connectivity with receive timeout. It does the following:

1. Creates a blocking synchronous UDP socket.
2. Performs a lookup query for the IP address of the test DNS server.
3. Sets the `RCVTIMEOUT` socket option to one second.
4. The client sends the server a basic DNS query.
5. The client receives a response from the server.
6. Checks that it is valid.
7. Tries to receive additional data after the initial response.
8. Verifies that we got a timeout (`PAL_ERR_SOCKET_WOULD_BLOCK`) error.
9. Closes the socket.

Prerequisite:

```
1. pal_socket
1. pal_getAddressInfo
2. pal_connect
3. pal_sendTo
4. pal_receiveFrom
5. pal_close
```

8.3 PAL Test Example

To build a PAL test executable, please refer to the instructions below (for mbedOS5.1.3 over Freescale-K64F board):

1. Define the environment variable: MBEDOS_ROOT to be the folder containing the mbed-os[1]. (Assume ROOT_PROJECT_FOLDER is the folder containing mbed-os)

```
export MBEDOS_ROOT=$(ROOT_PROJECT_FOLDER)
```

2. Go to PAL Test folder.

```
cd $(ROOT_PROJECT_FOLDER)/$(PAL_FOLDER)/Test/
```

3. Build the tests for mbedOS.

```
make mbedOS_all
```

4. To run the tests over the platform, execute:

```
make mbedOS_check
```

5. To see debug prints, set `DEBUG=1` in the compilation command:

```
make mbedOS_check DEBUG=1 VERBOSE=1
```

6. To build tests for a specific module, edit `$(PAL_FOLDER)/Test/makefile` under mbedOS platform. Change the value of the `TARGET_CONFIGURATION_DEFINES` to the desired module: (the default value is for all existing PAL modules)

   * `HAS_RTOS` → RTOS module APIs
   * `HAS_SOCKET` → Networking module APIs

7. To build a specific test, call `make mbedOS_all PAL_TEST=$(TheNameOfTheTest)`. To run a single test, use the same command as in step 4. For example:

```
make mbedOS_all PAL_TEST="MemoryPoolUnityTest"
make mbedOS_check
```

This option enables the platform developer to test the implemented APIs incrementally and separately before running larger scenarios.

[1] A folder containing mbed OS.

8.4 Example tests output result

Below is an example of captured results from a successful test run:

8.4.1 RTOS

```
==== mbedos_pal_rtos ====
--------  ----------------------------------  ----
pal_rtos    pal_osKernelSysTick_Unity           PASS
pal_rtos    pal_osKernelSysTick64_Unity         PASS
pal_rtos    pal_osKernelSysTickMicroSec_Unity   PASS
pal_rtos    pal_osKernelSysMilliSecTick_Unity   PASS
pal_rtos    pal_osKernelSysTickFrequency_Unity  PASS
pal_rtos    pal_osDelay_Unity                   PASS
pal_rtos    BasicTimeScenario                   PASS
pal_rtos    TimerUnityTest                      PASS
pal_rtos    MemoryPoolUnityTest                 PASS
pal_rtos    MessageUnityTest                    PASS
pal_rtos    AtomicIncrementUnityTest            PASS
pal_rtos    PrimitivesUnityTest1                PASS
pal_rtos    PrimitivesUnityTest2                PASS
pal_rtos    pal_init_test                       PASS
--------  ----------------------------------  ----
TOTAL  PASS  FAIL  IGNORE
-----  ----  ----  ------
 14     14    0     0

Final status: PASS.
```