UIS8910DM Programming Guide

Release Date	2019-6-15
Document No.	
Version	V1.1
Document Type	
Platform	UIS8910DM
OS Version	

声明 Statement

本文件所含数据和信息都属于紫光展锐所有的机密信息，紫光展锐保留所有相关权利。本文件仅为信息参考之目的提供，不包含任何明示或默示的知识产权许可，也不表示有任何明示或默示的保证，包括但不限于满足任何特殊目的、不侵权或性能。当您接受这份文件时，即表示您同意本文件中内容和信息属于紫光展锐机密信息，且同意在未获得紫光展锐书面同意前，不使用或复制本文件的整体或部分，也不向任何其他方披露本文件内容。紫光展锐有权在未经事先通知的情况下，在任何时候对本文件做任何修改。紫光展锐对本文件所含数据和信息不做任何保证，在任何情况下，紫光展锐均不负责任何与本文件相关的直接或间接的、任何伤害或损失。

All data and information contained in or disclosed by this document is confidential and proprietary information of UNISOC and all rights therein are expressly reserved. This document is provided for reference purpose, no license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document, and no express and implied warranties, including butwithout limitation, the implied warranties of fitness for any particular purpose, and non-infringement, as well as any performance. By accepting this

material, the recipient agrees that the material and the information contained therein is to be held in confidence and in trust and will not be used, copied, reproduced in whole or in part, nor its contents revealed in any manner to others without the express written permission of UNISOC. UNISOC may make any changes at any time without prior notice. Although every reasonable effort is made to present current and accurate information, UNISOC makes no guarantees of any kind with respect to the matters addressed in this document. In no event shall UNISOC be responsible or liable, directly or indirectly, for any damage or loss caused or alleged to be caused by or in connection with the use of or reliance on any such content.

CONTENTS:

CHAPTER

ONE

GETTING START

Contents

1.1 Overview

This SDK can be built on Windows and Linux. Most of the tools used during building the SDK are included in the SDK, including both Linux version and Windows version. Namely:

- GCC
- cmake
- ninja
- nanopb

- Python 3 (Linux build will use the system version)
- gperf

1.2 Linux Build

1.2.1 System Requirement

Only Ubuntu 16.04 is supported.

The following packages are needed:

```
$ sudo apt install build-essential python3 python3-tk qtbase5-dev
```

1.2.2 Build

```
$ . tools/launch.sh # select target by numerical index
$ cout
$ cmake ../.. -G Ninja
$ ninja
```

. tools/launch.sh It will set `PATH` and several environment variables. Also, it can be called as `. tools/launch.sh <target_name> <debug|release>` for non-interactive mode. Non-interactive mode is useful in building script.

cmake ../.. -G Ninja This SDK uses CMake as the building system. This step will generate *ninja* build file.

Though CMake can support various generators. Only *ninja* is support in this SDK.

It is only needed to run this step once. Afterward, when there are changes in source codes, CMakeLists.txt, Kconfig or target.config, *ninja* will invoke re-configuration automatically.

ninja This command will build the selected target.

In multi-processor system, *Ninja* will use available CPUs for parallel build. It is not needed to specify parallel job count.

ninja clean CMake and ninja can handle dependency very well. In most cases, incremental build is enough. In case to clean build results for a clean build, *ninja clean* can be called.

ninja unittests This SDK supports unit test framework. By default, unit tests are not built. If needed, *ninja unittests* can be called to build all unit tests

cout A function defined at `. tools/launch`. It will create target output directory, if not existed, and change directory to it.

croot A function defined at `. tools/launch`. It will change directory to the root of project.

1.2.3 CMake Options

The supported CMake command line options:

-DWITH_WERROR=on Waning is bad. However, `-Werror` makes development inconvenient. So, `-Werror` compile option isn't added at build. When `-DWITH_WERROR=on` is added in the *cmake* command line, `-Werror` compile option will be added.

It is recommended to add this option is CI, to make sure all warnings are solved.

-DBUILR_REVISION=<revision_name> It defines a string of revision name. Usually, it will be defined as the tag name. When not specified, the name is *DEVEL*.

1.2.4 Directory Convention

<project_root>/out/<target_name>-<debug|release> Output directory of target.

Out-of-source build is followed. That is, during build, there are no any build result will be generated in the source tree. All build result including intermediate build result will be created under the output directory.

<project_root>/out/<target_name>-<debug|release>/lib Directory for libraries.

<project_root>/out/<target_name>-<debug|release>/hex Directory for elf, map, bin and image files.

<project_root>/out/<target_name>-<debug|release>/include Directory for generated header files.

<project_root>/out/<target_name>-<debug|release>/rpcgen Source files generated by `rpcgen.py`.

1.3 Windows Build

1.3.1 System Requirement

Supported Windows versions:

- Windows 7 SP1, x64
- Windows 10, x64
- Visual C++ Redistributable for Visual Studio 2015 x86
- Visual C++ Redistributable for Visual Studio 2015 x64

1.3.2 Build Under cmd.exe

```
> call tools\launch.bat <target_name> <debug|release>
> cd out\target_name>
> cmake ..\.. -G Ninja
> ninja
```

It is very similar to Linux build. `tools\launch.bat` doesn't support interactive mode. The <*target_name*> is necessary, and <*debug|release*> is optional. When not specified, *debug* will be used.

cmd.exe doesn't support function as *bash*. So, `cout` and `croot` can't be used.

1.3.3 Build Under msys2/MINGW/Cygwin/Cygwin64

`mintty` and `bash` coming with msys2/MINGW/Cygwin/Cygwin64 is more convenient than *cmd.exe* for interactive. However, nothing from msys2/MINGW/Cygwin/Cygwin64 will be used for building.

```
$ . tools/launch.sh # select target by numerical index
$ cout
$ cmake ../.. -G Ninja
$ ninja
```

It is very similar to Linux build.

After `. tools/launch.sh` is called, `prebuilts/win32/python3` will be added to the beginning of PATH. This python 3 is Windows native Python 3. It it possible that it will affect msys2/Cygwin/Cygwin64 system.

1.4 Kconfig

Kconfig is used as configuration system.

For each target, `target/<target_name>/target.config` keeps the target configuration. At build, `out/<target_name>-<debug|release>/target.cmake` will be generated, and the configurations will be used by CMake.

Contrary to Linux build, there are no `config.h` will be generated. Rather, each module should use CMake `configuration_file` command to generate header file from a template.

1.5 menuconfig

`mconf` under Linux source tree is great, but can't run on Windows. Kconfiglib is used.

Python script menuconfig can work well on Linux, and Windows *cmd.exe*. However, it can't work well under *mintty* of msys2/Cygwin/Cygwin64.

On Linux:

```
$ cd <project_root> # assume ". tools/launch.sh" is already executed.
$ menuconfig.py
```

On Windows msys2/Cygwin/Cygwin64:

```
$ cd <project_root> # assume ". tools/launch.sh" is already executed.
$ tools/menuconfig.bat
```

On Windows cmd.exe:

```
> cd <project_root> # assume "call tools\launch.bat <target_name>" is already␣
↪executed.
> tools\menuconfig.bat
```

During `. tools/launch.sh` or `call tools\launch.bat`, *KCONFIG_CONFIG* will be set to `target/<target_name>/target.config`.

`minconfig.py` will strip down all default configurations, and only contain configuration value not equal to default value. The strip down version is suitable to keep in revision control.

1.6 guiconfig

`guiconfig` is a Python script from Kconfiglib.

On Linux/msys2/Cygwin/Cygwin64:

```
$ cd <project_root> # assume ". tools/launch.sh" is already executed.
$ guiconfig.py
```

On Windows cmd.exe:

```
> cd <project_root> # assume "call tools\launch.bat <target_name>" is already␣
↪executed.
> python3 tools\guiconfig.py
```

CHAPTER

TWO

ARCHITECTURE

Contents

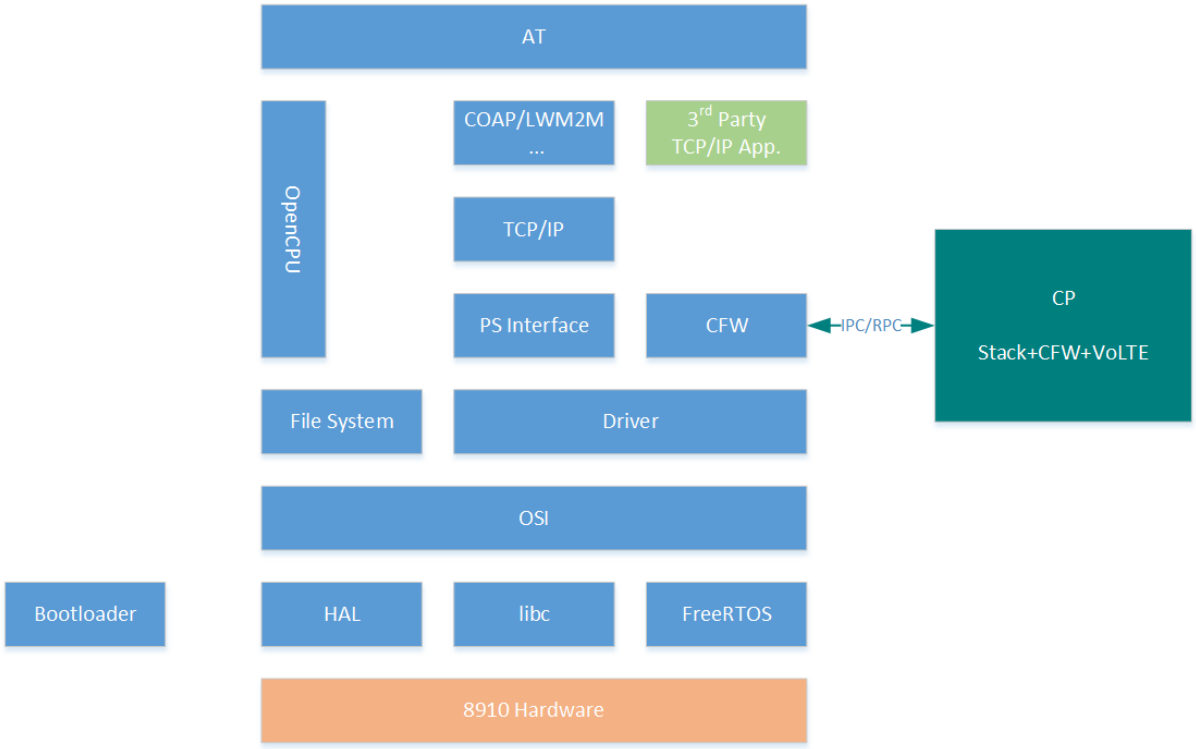
2.1 Overview

This SDK architecture can support various UNISOC IoT platforms. Currently, the followings are supported:

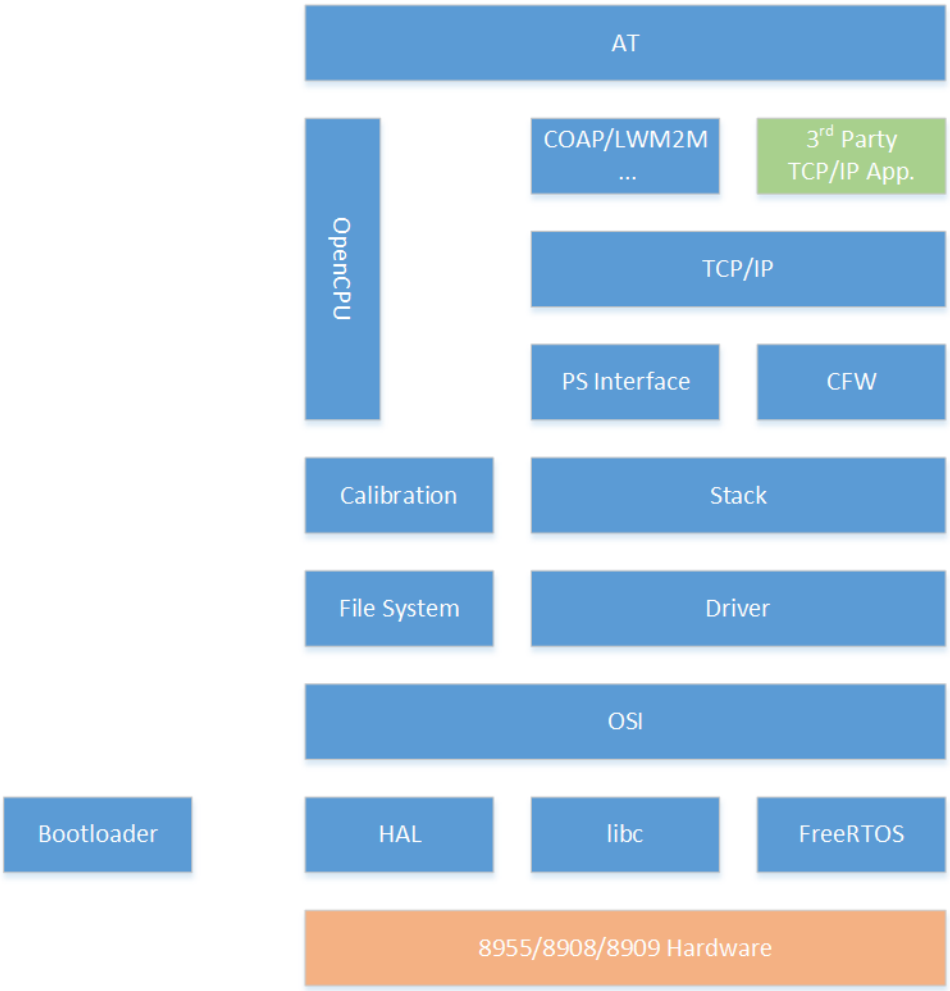
- 8955: 2G
- 8908: NB IoT
- 8909: NB IoT/2G

- 8915: LTE Cat1/2G, eMTC/2G

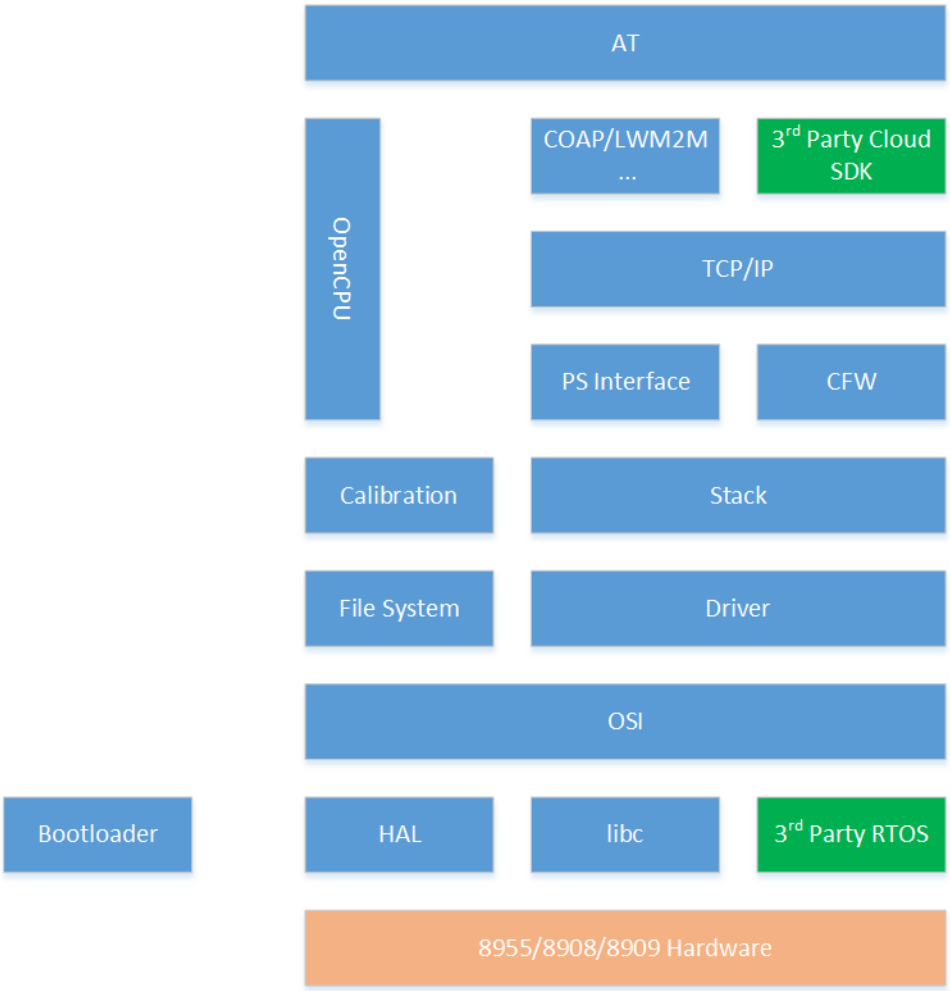
There are separated AP and CP in 8915, and the architecture of the SDK for 8915 is:



There is only one CPU for both application can stack in 8955/8908/8909, and the architecture of the SDK for 8955/8908/8909 is:



Though FreeRTOS is used in this SDK, there is OSI (OS Interface) layer to isolate RTOS and application. So, 3rd party RTOS can be easily port to the SDK, and replace FreeRTOS. Also, 3rd cloud SDK based on 3rd party RTOS can run natively without porting.



2.2 Bootloader

Bootloader is the software entrance after system jump out of ROM. The main features of bootloader:

Differential Upgrade

This SDK is target to platforms using NOR flash, and most of the code are running on NOR flash directly (XIP, eXecute In Place). And there is only one copy of codes on NOR flash. So, at upgrade, it is needed to be done in bootloader.

File system will be used for upgrade, for storing upgrade package. So, bootloader shall support file system. And the file system layout in bootloader should be exactly the same with the file system layout in application.

UART/USB Upgrade

If enabled, bootloader will monitor UART/USB input for upgrade. The use case is an upgrade tool will running on PC. When the communication between PC upgrade tool and platform is established, PC upgrade tool can upgrade

the firmware.

Note: Not all platforms support USB upgrade.

Application Signature Verification

When secure boot is enabled, ROM will verify the signature of bootloader. And bootloader shall verify the application signature.

Note: Not all platforms support secure boot.

Bootloader doesn't support upgrade itself *safely*. Application doesn't rely on any setting in bootloader, so in the life cycles, it doesn't needed to upgrade bootloader.

2.3 HAL (Hardware Abstract Layer)

HAL is platform deeply coupled layer. Examples:

- System clock setting;
- Memory setting;
- Basic power setting;

This layer can be shared with both bootloader and application. So, it can't use RTOS features.

Note: `osiEnterCritical` and `osiExitCritical` can be used in HAL.

2.4 OSI (OS Interface)

Besides RTOS abstraction, OSI will provide other common system features. It is closer to programming environment.

2.5 libc

newlib is used in this SDK.

2.6 Driver

Drivers are various peripheral drivers. It is built on top of HAL and OSI. Full RTOS features can be used in drivers, including thread, semaphore, mutex, timer, work queue and etc.

2.7 File System

SFFS is a file system optimized for NOR flash. All persistent information storage are using file system.

Also, POSIX file system APIs (such as `open`, `read`, `write`, and etc) are provided with prefix `vfs_`.

2.8 IPC/RPC

For platforms with separated AP and CP, IPC is the mechanism of AP/CP communication.

RPC is a mechanism built on top of IPC. With RPC, any CPU can call APIs implemented in other CPUs. Though peer CPU API call is a serial of IPC communication, application can't feel any differences with local API call.

2.9 CFW (Communication FrameWork)

CFW is a layer above protocol stack (2G, NB IoT, LTE Cat1, eMTC).

On some platforms, CFW implementation may be located on other CPU. However, with the help of RPC, application is the same.

Note: It is not supported to call CFW directly in customer development.

2.10 PS Interface

PS interface is a layer for PS data with protocol stack. The APIs is the same for platforms with protocol stack in the same CPU, and platforms with protocol stack in other CPU.

2.11 TCP/IP

lwIP is used in this SDK to provide TCP/IP stack.

2.12 COAP/LWM2M/...

There are many IoT IP based protocols in this SDK. Most of them are ported from open source projects.

2.13 AT Receiver

This SDK provides AT receiver. AT commands can be received from UART, USB (CDC/ACM).

2.14 Open CPU

This SDK provides open CPU features. Open CPU is an friendly development environment based on the base SDK.

OSI, drivers and TCP/IP are well documented, and they can be used in development on top of the base SDK. Also, it supports:

Dynamic Loader

An easy-to-use dynamic loader mechanism is implemented in this SDK. The basic features:

- Application can call exposed APIs in base SDK.
- Base SDK and application can be upgraded independently.
- Application can be loaded from files on file system, and can be loaded from NOR flash directly. When loading from NOR flash directly, application can run on NOR flash directly.

Some limitations:

- Application can't access global variables in base SDK directly.
- Flash and RAM reserved for application shall be planned beforehand.

AT Command by API

There is a virtual AT channel in this SDK. With the virtual AT command channel, legacy application based on sending AT commands can be ported easily.

RIL

Due to it is not supported to call CFW directly in customer development, RIL is provided. It is a bunch of simplified communication API.

CHAPTER

THREE

FLASH LAYOUT

Contents

3.1 Overview

This IoT SDK is based on NOR flash, and SFFS is the file system for NVRAM, modem image and etc.

Typical flash layout for 8MB flash:

Partition	Range	Size	Mount Point
bootloader	`0..0x10000`	64KB	
application	`0x10000..0x340000`	3.2MB	
system FS	`0x340000..0x4a0000`	1.4MB	/
modem FS	`0x4a0000..0x7e0000`	3.2MB	/modem
factory FS	`0x7e0000..0x800000`	128KB	/factory

Typical flash layout for 16MB flash:

Partition	Range	Size	Mount Point
bootloader	`0..0x10000`	64KB	
application	`0x10000..0x980000`	9.6MB	
system FS	`0x980000..0xca0000`	3.1MB	/
modem FS	`0xca0000..0xfe0000`	3.2MB	/modem
factory FS	`0xfe0000..0x1000000`	128KB	/factory

The size of each partition can be configured. However, it is not recommended to change the number and order of the partitions.

The default layout leaves very large rooms for application, and the system file system is tight. When there are many customized data shall be stored, it can be considered to decrease application partition, and increase system file system partition.

When file system configurations (including flash offset, size, erase block size and logic block size) are changed, the original files on file system will be destroyed.

Though overhead of file system is highly optimized on SFFS, it should be considered at flash layout plan.

3.2 Bootloader

At boot, ROM will load bootloader to internal SRAM and execute on internal SRAM. ROM will only load size of `0xbf40` from flash. When secure boot is enabled, the last `0x260` is used for signature. So, for one stage bootloader, the maximum code size is `0xbce0`.

When `0xbce0` is not enough, two stage bootloader is needed. Also, when secure boot is needed, the first stage bootloader should verify the signature of second stage of bootloader. Currently, one stage bootloader is used.

Feature list of bootloader:

- Check application image, and jump to application;
- When secure boot is enabled, verify application signature;
- FOTA upgrade;

Note: Due to ROM can support UART download, it is not needed to support download and/or upgrade in bootloader.

3.3 Application

Application partition is not managed by file system. It is for application, including SDK and customized application. And most of the application shall run on flash directly (XIP).

3.4 System FS

This is the partition for run time data, such as NVRAM, and managed by file system. By default, it is mounted as read-write.

Two directories are used by SDK:

- /modemnvm (CONFIG_FS_MODEM_NVM_DIR)
- /nvm (CONFIG_FS_AP_NVM_DIR)
- /fota (CONFIG_FS_FOTA_DATA_DIR)

FOTA upgrade data will be stored in this partition. So, it should ensure there are enough rooms for FOTA upgrade data.

Also, this partition will be written frequently. For NOR flash file system, performance will downgrade when file system is close to full:

- write will be much slower;
- erase count will increase rapidly;

Note: In case there are data will be stored frequently, flash life-cycle should be considered. Due to SFFS can handle wear-leveling well, the larger spare rooms in file system, the life-cycle of flash can be longer.

FOTA data size is hard to estimate. Though FOTA data size can be very small when the code change is small, 10% of the original size is the minimum requirement.

3.5 Modem FS

This is the partition for modem image, and managed by file system. By default, it is mounted as read-only to avoid overwrite accidentally.

3.6 Factory FS

This is the partition for factory data, such calibration data, IMEI, serial number and etc., and managed by file system. By default, it is mounted as read-only to avoid overwrite accidentally.

3.7 Configurations

- CONFIG_BOOT_FLASH_OFFSET
- CONFIG_BOOT_FLASH_SIZE
- CONFIG_APP_FLASH_OFFSET

- CONFIG_APP_FLASH_SIZE
- CONFIG_FS_MODEM_MOUNT_POINT
- CONFIG_FS_MODEM_FLASH_OFFSET
- CONFIG_FS_MODEM_FLASH_SIZE
- CONFIG_FS_MODEM_EB_SIZE
- CONFIG_FS_MODEM_PB_SIZE
- CONFIG_FS_SYS_MOUNT_POINT
- CONFIG_FS_SYS_FLASH_OFFSET
- CONFIG_FS_SYS_FLASH_SIZE
- CONFIG_FS_SYS_EB_SIZE
- CONFIG_FS_SYS_PB_SIZE
- CONFIG_FS_FACTORY_MOUNT_POINT
- CONFIG_FS_FACTORY_FLASH_OFFSET
- CONFIG_FS_FACTORY_EB_SIZE
- CONFIG_FS_FACTORY_PB_SIZE

3.8 Frequently Overwrite Small File

Here is an example of writing small file frequently. File system configuration:

- flash size: 1MB
- erase block size: 32KB
- logical block size: 512B
- file size: 200B
- write every 5 minute, overwrite the file

At worse case, the file system is very close to full, each write will cause flash erase. And then there are 105,120 flash erase every year.

At best case, the file system is empty. Then, 1950 times of file write will cause one flash erase of every flash sectors. So, there are only 54 flash write every year.

CHAPTER

FOUR

KERNEL (OS INTERFACE)

Contents

4.1 Overview

This IoT SDK is based on RTOS. By design, SDK won't access the used RTOS directly. Rather, OSI (OS Interface) layer is designed. SDK can only access OSI API. OSI can be implemented by various RTOS. Even the underlay RTOS is changed, the SDK itself can keep unchanged.

OSI based on FreeRTOS is provided in the SDK. And it is possible and not hard to implement OSI based on other RTOS.

4.2 Thread

Thread is the basic RTOS scheduling unit. Threads have independent stack.

OSI thread has a bundled event queue. There are APIs to send event to thread. The basic model of OSI thread is event processing. An typical OSI thread is:

```
void entry(void *argument)
{
    osiThread_t *thread = osiThreadCurrent();
    while (running)
    {
        osiEvent_t event = {};
        osiEventWait(thread, &event);
        _processEvent(&event);
    }
    osiThreadExit();
}
```

There no API to delete another thread. Rather, thread should be terminated by themselves, through some kinds of pre-defined inter-thread communication method. At the end of thread entry function, `osiThreadExit()` should be called.

4.3 osiEvent_t

`osiEvent_t` is the data structure of OSI event:

```
typedef struct osiEvent
{
    uint32_t id;     ///< event identifier
    uint32_t param1; ///< 1st parameter
    uint32_t param2; ///< 2nd parameter
    uint32_t param3; ///< 3rd parameter
} osiEvent_t;
```

The message queue for OSI event is named as event queue. The most important inter-thread communication is to send and wait event.

At send and wait event, the event body is copied. Three words are enough to carry event information in most cases. Then this design can reduce `malloc` and `free` function calls.

The details of the event parameters are defined by each event. In case that three words are not enough, dynamic pointer can be carried as event parameter. The memory management convention is defined by each event. When dynamic memory is used, usually event sender allocate memory, event receiver free memory after process.

4.4 Thread Callback

Thread callback is a callback to be executed inside specified thread. The implementation is based on OSI event. However, the codes will be more readable with `osiThreadCallback`, rather than huge switch/case of event IDs.

With event ID, the codes will look like:

```
// sender
osiEvent_t event = {EVENT_ID, param1};
osiEventSend(thread, &event);

// receiver
for (;;)
{
    osiEvent_t event = {};
    osiEventWait(thread, &event);
    switch(event.id)
    {
    case EVENT_ID:
        function_call(event.param1);
        break;
    }
}
```

With thread call, the codes will look like:

```
osiThreadCallback(thread, function_call, param1);
```

4.5 Thread Notify

Thread notify is thread callback with state. It is used for repeatedly notification to be sent to a thread. With thread callback, there may exist many duplicated event to be sent to thread event queue. And it may cause thread event queue full.

Thread notify will keep the state about whether the event has already sent to the thread event queue. When it is already sent to the thread event queue, no duplicated event will be sent to thread event queue. Conceptually, trigger a thread notify is:

```
if (the_event_not_in_event_queue())
    osiEventSend(event_for_notify());
```

4.6 Semaphore

There are 2 kinds of semaphore:

- binary semaphore
- counting semaphore

OSI semaphore APIs can support these 2 kinds of semaphores.

Semaphore can used in ISR, including `osiSemaphoreRelease` and `osiSemaphoreTryAcquire` with timeout is zero.

4.7 Mutex

Comparing to semaphore, mutex will support recursive lock and possibly priority inherit.

Note: OSI mutex won't ensure priority inherit. That is, if priority inherit is not supported in the underlay RTOS, OSI mutex won't support also. If priority inherit is important for application, an RTOS with priority inherit should be chosen.

FreeRTOS mutex support priority inherit. Though FreeRTOS support both recursive mutex and non-recursive mutex, OSI only support recursive mutex. Non-recursive mutex may be a little faster than recursive mutex, the benifit is not enough to make APIs more complex.

Mutex can't be used in ISR.

4.8 Work and Work Queue

Work is just a function to be executed. Comparing to simple callback:

- `osiWorkEnqueue` will check work state, and avoid duplicated execution.

- Work has complete callback.

Work queue is the execution environment of *work*. If needed, thread pool can be created for a work queue to reduce work execution latency.

Note: Thread pool feature isn't implemented currently.

There are several typical scenario of work usage:

Use work inside ISR

When send `osiEvent_t` in ISR, the event will be lost if the event queue is full due to ISR can't wait. In case that ISR will only notify thread, *work* can ensure the notification won't be lost, even it is possible that *work* will be executed only once when multiple interrupt comes. `osiWorkEnqueue` is non-blocking, and can be called in ISR.

At kernel start, a work queue with `OSI_PRIORITY_HIGH` will be created, it can be used for ISR work to notify thread. Due to this work queue may be shared, independent work queue can be created for latency sensitive works.

Asynchronous Execution

In event loop style programming, it is better not to call functions with long execution time (such as download by HTTP). *Work* can be used for asynchronous execution.

Background Execution

At kernel start, a work queue with `OSI_PRIORITY_LOW` will be created, it can be used for background works.

4.9 Interrupt Latency

We take care about interrupt latency seriously, but not extremely. We will try best to minimize interrupt latency, but goal of the SDK is not hard real-time system. So, it is permitted to disable interrupt, just to make sure that the execution time during interrupt disabled should be as short as possible.

4.10 ISR Programming

The requirement and recommendation should be followed for both ISR and functions which will be called in ISR.

- Most of OSI APIs shouldn't be called in ISR, except:
  - osiEnterCritical
  - osiExitCritical
  - osiEventTrySend (timeout must be 0, event may be lost)
  - osiThreadCallback (event may lost)
  - osiMessageQueueTryPut (timeout must be 0, message may be lost)
  - osiMessageQueueTryGet (timeout must be 0)

  - osiSemaphoreTryAcquire (timeout must be 0)
  - osiSemaphoreRelease
  - osiWorkEnqueue

- Functions to be called in ISR should be as short as possible. The execution time will have direct impact of interrupt latency.
- In timer ISR, multiple callbacks may be called. So, timer callback should be event shorter.
- Suspend callbacks are executed with interrupt disabled. And multiple callbacks may be called before suspend. So, suspend callback should be even shorter.

4.11 Timer

Most RTOS only provides tick based timer. The precision is not enough. So, OSI implements high precision timer without RTOS tick.

Note: Though it is possible to use high precision timer to support RTOS tick, it will make it much harder to port OSI to another RTOS.

There are 3 kinds of behaviors at timer expiration:

1. Call the registered callback directly. The callback is executed in timer ISR. The execution latency after timer expiration will be small, and the callback should be as short as possible.
2. Call the registered callback in specified thread. This is the most common case.
3. Send an OSI event `{EV_TIMER, <TIMER_ID>}` to specified thread.

There are 3 kinds of timer start APIs:

1. one shot timer: Expiration period is in milliseconds, and the maximum period is ~50 days.
2. one shot timer with high precision: Expiration period is in microseconds, the maximum period is ~1.2 hours. The expiration precision is higher, and the real precision depends on hardware.
3. repeated timer: The timer will expire periodic.

Note: There are no API for timer expired at absolute system up time or epoch time. Alarm can be used for that case.

4.11.1 Timer and Sleep

Timer and tick have impact on power consumption, especially when deep sleep (or suspend) is enabled. The design rule is:

> Timer and tick implementation should always be correct. That is, when a timer is active, the callback must be called at expiration no matter system is suspended or not. Power consumption optimization is left to application designer, with proper OS support.

So, all of the followings will wakeup system:

- active timer
- thread sleep
- timeout in various OSI APIs

`osiTimerStartRelaxed` is for power consumption optimization. When proper relaxed timeout is set, system can sleep or suspend longer. And when *relaxed* is OSI_DELAY_MAX, that timer won't wakeup system.

4.11.2 Timer Stop Corner Case

When `osiTimerStop` is called close to timer expiration time, there are several cases about whether the callback will be executed.

Callback in ISR

When `osiTimerStop` is returned, the callback won't be executed. However, it is possible that the callback will be executed during `osiTimerStop` is executing.

Callback in thread, stop in the same thread

This is the recommended usage model. The callback won't be executed.

Callback in thread, stop in another thread

When `osiTimerStop` is returned, the callback won't be executed. However, it is possible that the callback will be executed during `osiTimerStop` is executing.

4.11.3 Timer API inside Callback

Inside timer expiration callback, timer APIs are permitted (with care).

Due to timer expiration callback is executed inside ISR, `osiTimerCreate` and `osiTimerDelete` are prohibited to be called.

Start and stop the timer itself is the same as normal. Stop another timer is the same as normal also. Start another time will follow *stop and start with new configuration*. So, if the other timer will expire in the same ISR, and the callback isn't already called, the callback *won't* be called. Only when it expires with the new configuration, the callback will be called.

4.11.4 Timer Pool

The normal usage of timer is to create timer instance, and call OSI APIs on the timer instance. Timer pool provides another style of timer instance management.

Timer pool manages timer instances inside. Caller can start and stop timer by callback thread, callback function and callback context directly. Inactive timer instances will be reused.

Timer pool can't be used in ISR.

There is a global timer pool. When timer pool instance is not specified, the global timer pool will be used.

The normal usage model (create, start, stop) is recommended.

4.12 Elapsed Timer

Elapsed timer is not a *real* timer. It is similar to *stop watch*. When `osiElapsedTimerStart` is called, it start to count elapsed time. For example:

```
osiElapsedTimer timer;
osiElapsedTimerStart(&timer);
// ......
uint32_t ms = osiElapsedTime(&timer);
uint32_t us = osiElapsedTimeUS(&timer);
```

Conceptually, it is the same as:

```
int64_t start_ms = osiUpTime();
int64_t start_us = osiUpTimeUS();
// ......
uint32_t ms = osiUpTime() - start_ms;
uint32_t us = osiUpTimeUS() - start_us;
```

Just elapsed timer APIs make the codes easier to be read.

Elapsed timer is not suitable for very long time. Typically, it will use 32bits for microseconds. And the maximum time is about 4000 seconds.

4.13 Power management

The following aspects should be considered for power management:

- when and how to turn on and off power
- fine tune power voltage
- when and how to turn on and off clock
- fine tune clock frequency

Considering hardwares are diversified, a generic model is much harder than thought. So, we will try to create *not-so-generic* model. It should be much better than *no model* (all cases are case by case). However, the model will focus on existed and expected hardware.

PM core will just handle *when and how to turn on and off power*. A model to handle all aspects will be over-killed.

4.13.1 PM Source

PM source is the model for sleep. The model is for:

- whether sleep is permitted;
- actions (callbacks) during sleep and wakeup;

For existed hardwares, there are several kinds of sleep:

- change clock to 32KHz;
- power off and memory is kept;

PSM/PM3 isn't considered as sleep. It is close to *hibernate*. However, it is impossible to store enough information to NVRAM. And then it is impossible to implement the same meaning of *hibernate* as PC. Now, we consider that the resume procedure of PSM will be very close to cold boot. Only in several points, we will check whether it comes from PSM. That is, there are no model for PSM, and handled *case by case*.

In new hardwares, CPU power down will be common for power saving, and the actions for sleep are called *suspend* and *resume*.

About whether system can suspend, wakeup source can be in one of 3 states:

- *inactive*: It won't prevent system to suspend.
- *suspend possible*: It provide callback to double check whether it can suspend.
- *active*: It will prevent system to suspend.

suspend possible is similar to polling mode. The module won't report active/inactive explicitly. Rather, system should call callback to check whether the module can be suspended.

Typical PM source is hardware module. However, it is possible to create PM source for pure software module.

4.13.2 Wake Lock

PM source use `osiPmWakeLock` and `osiPmWakeUnlock` to change state.

Though a simple system-wide counter is enough for wake lock, it is hard to debug when system can't sleep. So, PM source is needed for wake up. When something is wrong, we can go through all PM sources to find out the active PM sources.

PM source wake up is designed as counting. That is only after the same number of `osiPmWakeUnlock` are called as the number of `osiPmWakeLock`, the PM source will be changed to inactive state.

4.13.3 Resume Order

It is possible that the resume order of modules are important. Ideally, we can setup model (for example, parent and children concept) for the dependency. However, the model will be over-killed.

So, PM core will provide APIs to adjust the resume callback order.

Suspend callbacks are called in reversed order of resume callbacks. After resume order is adjusted, suspend order is changed corresponding. It *shouldn't* exist cases with special suspend and resume order.

It is assumed that the order of *prepare* callbacks (to double check whether suspend is permitted) doesn't matter.

4.13.4 Auto Sleep

Some active PM source can become inactive after a period automatically. Though it is possible to implement the timeout in PM core, it increases the complexity of PM core. Currently, the timeout should be implemented inside the PM source.

4.13.5 Callback Restriction

All callbacks are called by PM core. During the callbacks are called, interrupts are disabled. So,

- Callbacks execution should be very fast. Otherwise, system interrupt latency will be increased.
- Not call OS thread/mutex/semaphore/queue APIs. With careful design, some APIs can work, but it increases complexity.

4.14 Cache

Cache coherence shall be considered in the following cases:

- write to memory through cache, and the memory will be used as DMA source address;
- for DMA destination address, the memory will be read through cache;
- codes are written to memory through D cache, and then the codes will be executed through I cache.

Even there are difference about cache in the supported chips of this SDK, the cache related APIs are the same. This APIs will just ensure the correctness, however it is possible that uncachable access will have better performance on certain chip.

4.15 atomic

Though atomic is defined in C11 can be used, it can't be used in this SDK. This SDK will target various chips, and not all chips can support atomic well. When atomic operation is needed, protect the operation by `osiEnterCritical` and `osiExitCritical`.

4.16 Floating Point

8910

The CPU of 8910 is ARM Cortex-A5 with neon. To enable applications with heavy floating point usage, the compile option shall be:

```
-mfpu=neon-vfpv4 -mfloat-abi=hard
```

Due to float-abi `soft` and `hard` can't be linked together, the above compile option should be set to all compile units running on 8910.

Due to it is possible that many threads won't use floating point, it will waste cycles to save and restore FPU context at each thread context switch. So, FPU context save and restore is performed on demand. That is, if a thread will use FPU, `osiThreadSetFPUEnabled(true)` should be called *before* floating point instructions. Typical example is:

```
void myThreadEntry(void *param)
{
    osiThreadSetFPUEnabled(true);

    // ......
}
```

ISR is prohibited to use floating point instructions. However, GCC may use floating point registers for optimization. To prevent this behavior, the compile option shall be:

```
-mcpu=cortex-a5 -mtune=generic-armv7-a
```

When compile option is set to:

```
-mcpu=cortex-a5 -mtune=cortex-a5
```

it is possible that GCC will use floating point registers. Only when it is sure that any functions inside the compile unit won't be executed in ISR, including ISR callbacks, and it is verified that `-mtune=cortex-a5` will generate *faster* codes, `-mtune=cortex-a5` can be used.

Platform Without FPU

For platform without FPU, *soft-float* will be used. `osiThreadSetFPUEnabled` will be nothing, and is harmless.

Note: Though floating point behaviors may be different on different platform, and different underlay RTOS, it is not recommended to depend on implementation. For portable codes:

- Not use floating point in ISR;
- Call `osiThreadSetFPUEnabled(true)` at the beginning of thread entry if the thread may use floating point;

4.17 FreeRTOS Integration

At OSI implementation based on FreeRTOS, `configNUM_THREAD_LOCAL_STORAGE_POINTERS` must be greater than 0, and the first thread local storage pointers is used for thread event queue.

So, FreeRTOS thread can be used as parameter of `osiEventSend` and `osiEventWait`. However, OSI thread **is** a FreeRTOS thread.

TICKLESS must be enabled.

Timer in FreeRTOS is not used. OSI timer implementation doesn't depend on underlay RTOS. FreeRTOS tick is generated by periodic OSI high precision timer.

Memory management in FreeRTOS is not used. OSI memory management implementation doesn't depend on underlay RTOS.

OSI semaphore is just the FreeRTOS semaphore. OSI mutex is just the FreeRTOS mutex.

Thread critical section is changed. FreeRTOS itself can be configured that high priority interrupts are enabled during thread critical section. It can reduce the interrupt latency of high priority interrupts. However, this design is not necessary for the applications of this SDK. In this SDK, it is acceptable to disable all interrupts for thread critical section.

4.18 API Reference

Defines

OSI_WAIT_FOREVER

OSI_DELAY_MAX

OSI_ASSERT(expect_true, info)

Typedefs

typedef uint32_t **osiElapsedTimer_t**
elapsed timer for couting elapsed time

typedef struct osiTimer **osiTimer_t**
opaque data structure for timer

typedef struct osiTimerPool **osiTimerPool_t**
opaque data structure for timer pool

typedef struct osiThread **osiThread_t**
opaque data structure for thread

typedef struct osiMessageQueue **osiMessageQueue_t**
opaque data structure for message queue

typedef struct osiEventQueue **osiEventQueue_t**
opaque data structure for event queue

Event queue is just a message queue, and the message is `osiEvent_t` (event itself rather than pointer).

typedef struct osiMutex **osiMutex_t**
opaque data structure for mutex

typedef struct osiSemaphore **osiSemaphore_t**
opaque data structure for semaphore

typedef struct osiWork **osiWork_t**
opaque data structure for work

typedef struct osiWorkQueue **osiWorkQueue_t**
opaque data structure for work queue

typedef struct osiNotify **osiNotify_t**
opaque data structure for thread notify

typedef void (***osiCallback_t**)(void *ctx)
function type of callback

typedef void (***osiThreadEntry_t**)(void *argument)
function type of thread entry

typedef void (***osiIrqHandler_t**)(void *ctx)
function type of interrupt handler

typedef struct *osiEvent* **osiEvent_t**
event, with ID and 3 parameters

typedef enum *osiThreadPriority* **osiThreadPriority_t**
thread priority

The definition is independent of implementation. Though some implementation will use larger value for higher priority and others will use smaller value for highe priority, this enum will use larger value for higher priority.

`OSI_PRIORITY_IDLE` and `OSI_PRIORITY_HISR` are reserved, can't be used.

The definition is the same as CMSIS-RTOS.

typedef enum *osiSuspendMode* **osiSuspendMode_t**
suspend mode

System behavior of suspend modes will be diffrent among underlay platform. Driver shall take care the difference, and most likely application won't take care it.

typedef enum *osiResumeSource* **osiResumeSource_t**
resume wakeup source

Resume wakeup source depends on platform. They should be defined in `hal_chip.h`. One bit indicates one source, and multiple sources are possible. `OSI_RESUME_ABORT` is reserved to indicate suspend is aborted.

typedef enum *osiBootCause* **osiBootCause_t**
boot cause

This list is for cold boot cause. Though it is rare, it is possible there exist multiple boot causes simultaneously.

Usually boot cause is determined from hardware status registers.

typedef enum *osiBootMode* **osiBootMode_t**
boot mode

Besides normal boot, there are several other boot modes. For each platform, not all boot modes are supported.

Usually, boot mode can be determined by hardware (for example, some GPIO) or software (for example, by flags written at `osiShutdown`).

typedef enum *osiShutdownMode* **osiShutdownMode_t**
shudown mode

For each platform, not all shutdown modes are supported.

typedef enum *osiPsmDataOwner* **osiPsmDataOwner_t**
PSM data owner

typedef void (***osiShutdownCallback_t**) (void *ctx, *osiShutdownMode_t* mode)
shuwdown callback function type

Before shutdown, the registered callbacks will be invokes. The callbacks are executed in system high priority work queue, and with interrupt disabled. So, the callbacks shouldn't rely on system high priority work queue and interrupts. However, thread schedule is still working and multi-thread can work still.

typedef struct *osiPmSourceOps* **osiPmSourceOps_t**
PM source callbacks.

All the callbacks are called with interrupt disabled. So, it is not needed to call `osiEnterCritical` or `osiIrqSave` for protection. And it is not error to call them.

Don't call blocking API in the callback.

typedef struct osiPmSource **osiPmSource_t**
PM source opaque data struct

Enums

enum osiThreadPriority
thread priority

The definition is independent of implementation. Though some implementation will use larger value for higher priority and others will use smaller value for highe priority, this enum will use larger value for higher priority.

`OSI_PRIORITY_IDLE` and `OSI_PRIORITY_HISR` are reserved, can't be used.

The definition is the same as CMSIS-RTOS.

Values:

OSI_PRIORITY_IDLE = 1

OSI_PRIORITY_LOW = 8

OSI_PRIORITY_BELOW_NORMAL = 16

OSI_PRIORITY_NORMAL = 24

OSI_PRIORITY_ABOVE_NORMAL = 32

OSI_PRIORITY_HIGH = 40

OSI_PRIORITY_REALTIME = 48

OSI_PRIORITY_HISR = 56

enum osiSuspendMode
suspend mode

System behavior of suspend modes will be diffrent among underlay platform. Driver shall take care the difference, and most likely application won't take care it.

Values:

OSI_SUSPEND_PM1
1st level suspend mode

OSI_SUSPEND_PM2
2nd level suspend mode

enum osiResumeSource
resume wakeup source

Resume wakeup source depends on platform. They should be defined in `hal_chip.h`. One bit indicates one source, and multiple sources are possible. `OSI_RESUME_ABORT` is reserved to indicate suspend is aborted.

Values:

OSI_RESUME_ABORT = (1 << 31)
resume by suspend aborted

enum osiBootCause
boot cause

This list is for cold boot cause. Though it is rare, it is possible there exist multiple boot causes simultanuously.

Usually boot cause is determined from hardware status registers.

Values:

OSI_BOOTCAUSE_UNKNOWN = 0
placeholder for unknown reason

OSI_BOOTCAUSE_PWRKEY = (1 << 0)
boot by power key

OSI_BOOTCAUSE_PIN_RESET = (1 << 1)
boot by pin reset

OSI_BOOTCAUSE_ALARM = (1 << 2)
boot by alarm

OSI_BOOTCAUSE_CHARGE = (1 << 3)
boot by charge in

OSI_BOOTCAUSE_WDG = (1 << 4)
boot by watchdog

OSI_BOOTCAUSE_PIN_WAKEUP = (1 << 5)
boot by wakeup

OSI_BOOTCAUSE_PSM_WAKEUP = (1 << 6)
boot from PSM wakeup

enum osiBootMode
boot mode

Besides normal boot, there are several other boot modes. For each platform, not all boot modes are supported.

Usually, boot mode can be determined by hardware (for example, some GPIO) or software (for example, by flags written at `osiShutdown`).

Values:

OSI_BOOTMODE_NORMAL = 0
normal boot

OSI_BOOTMODE_DOWNLOAD = 0x444e
'DN' boot to download mode

OSI_BOOTMODE_CALIB = 0x434c
'CL' boot to calibration mode

OSI_BOOTMODE_NB_CALIB = 0x4e43
'NC' boot to NB calibration mode

OSI_BOOTMODE_BBAT = 0x4241
'BA' boot to BBAT mode

OSI_BOOTMODE_UPGRADE = 0x4654
'FT' boot to bootloader upgrade

OSI_BOOTMODE_PSM_RESTORE = 0x5053
'PS' boot to PSM restore

enum osiShutdownMode
shudown mode

For each platform, not all shutdown modes are supported.

Values:

OSI_SHUTDOWN_RESET = 0
normal reset

OSI_SHUTDOWN_FORCE_DOWNLOAD = 0x5244
'RD' reset to force download mode

OSI_SHUTDOWN_DOWNLOAD = 0x444e
'DN' reset to download mode

OSI_SHUTDOWN_CALIB_MODE = 0x434c
'CL' reset to calibration mode

OSI_SHUTDOWN_NB_CALIB_MODE = 0x4e43
'NC' reset to NB calibration mode

OSI_SHUTDOWN_BBAT_MODE = 0x4241
'BA' reset to BBAT mode

OSI_SHUTDOWN_UPGRADE = 0x4654
'FT' reset to upgrade mode

OSI_SHUTDOWN_POWER_OFF = 0x4f46
'OF' power off

OSI_SHUTDOWN_PSM_SLEEP = 0x5053
'PS' power saving mode

enum osiPsmDataOwner
PSM data owner

Values:

OSI_PSMDATA_OWNER_KERNEL
kernel

OSI_PSMDATA_OWNER_STACK
stack

OSI_PSMDATA_OWNER_AT
AT engine.

OSI_PSMDATA_OWNER_USER = 100
start owner for user application

Functions

void **osiInvokeGlobalCtors**(void)
invoke global constructors

Global constructors are not called during boot. Rather, they will be called in `osiInvokeGlobalCtors`, and it shall be called in `osiAppStart`.

At `osiAppStart`, RTOS is ready. So, it is permitted to call more OSI APIs at global constructors.

Though global constructors are supported. It is not encouraged to use this feature. Only use this feature when you really know what you are doing.

void **osiKernelStart**(void)
kernel start

Start the kernel. This API will create system threads (at least, idle thread) and start thread scheduler. So, it won't return.

Before `osiKernelStart` is called, kernel data structure may be uninitialized, and many osi APIs can be called. Including

- timer

- IRQ
- power management

uint32_t **osiSchedulerSuspend**(void)
suspend thread scheduler

After scheduler is suspended, there are no thread context switch. However, interrupt handlers will be executed.

The meaning of return flag depends on underlay RTOS. Don't assume the meaning of the return value.

Return scheduler suspend flag.

void **osiSchedulerResume**(uint32_t *flag*)
resume thread scheduler

Scheduler suspend and resume is not *recursive*. That is, after `osiSchedulerResume` is called, scheduler is resumed no matter how many times `osiSchedulerSuspend` are called.

Parameters

- `flag`: scheduler suspend flag returned by the latest `osiSchedulerSuspend`

uint32_t **osiEnterCritical**(void)
enter critical section

The underlay RTOS may have different implementation for critical section. It may manipulate CPU IRQ enable bit(s), or manipulate IRQ mask.

This can be called in ISR.

Return critical section flags

void **osiExitCritical**(uint32_t *critical*)
exit critical section

Critical section flags is implementation depend. It should be the value returned by `osiEnterCritical`, and don't change the value manually.

Critical section is *recursive*. That is, after `osiExitCritical` is called, it doesn't mean system will enter *unprotected* state. Rather, it will return to state before last call of `osiEnterCritical`. For example:

```
uint32_t critical1 = osiEnterCritical();
uint32_t critical2 = osiEnterCritical();
// ...
osiExitCritical(critical2);
osiExitCritical(critical1);
```

After the first call of `osiExitCritical`, system is in *protected* state still.

In recursive, the exit order must be the reverse order of enter. For example, the following codes are wrong:

```
uint32_t critical1 = osiEnterCritical();
uint32_t critical2 = osiEnterCritical();
// ...
osiExitCritical(critical1);
osiExitCritical(critical2);
```

This can be called in ISR.

Parameters

- `critical`: critical section flags

uint32_t **`osiIrqSave`**(void)
get IRQ flags and disable IRQ

This will always manipulate CPU IRQ enable bit(s).

After this call, `osiEnterCritical` can't be called. When `osiEnterCritical` manipulates IRQ mask, it is very possible that it will change CPU IRQ enable bit(s) without protection.

In most cases, `osiIrqSave` and `osiIrqRestore` shouldn't be used, unless you really know what you are doing.

Return IRQ flags before disable IRQ

void **`osiIrqRestore`**(uint32_t *flags*)
restore IRQ flags

IRQ flags is arch and implementation depend. It should be the value returned by `osiIrqSave`, and don't change the value manually.

Parameters

- `flags`: IRQ flags

bool **`osiIrqSetHandler`**(uint32_t *irqn*, *osiIrqHandler_t* *handler*, void **ctx*)
set interrupt handler

When interrupt arrived, the registered handler will be called with the registered context pointer.

For each interrupt, only one handler can be registered. When a handler is already registered for an interrupt, `osiIrqSetHandler` will replace the old one.

`irqn` depends on interrupt controller in system. When GIC is used, it is the number in GIC.

Return

- true on success
- false on invalid parameters

Parameters

- `irqn`: IRQ number

- `handler`: IRQ handler
- `ctx`: IRQ handler context pointer

bool **`osiIrqEnable`**(uint32_t *irqn*)
enable interrupt

Return

- true on success
- false on invalid parameters

Parameters

- `irqn`: IRQ number

bool **`osiIrqDisable`**(uint32_t *irqn*)
disable interrupt

Return

- true on success
- false on invalid parameters

Parameters

- `irqn`: IRQ number

bool **`osiIrqEnabled`**(uint32_t *irqn*)
whether interrupt is enabled

Return

- true if interrupt is enabled
- false if interrupt is disabled

Parameters

- `irqn`: IRQ number

bool **`osiIrqSetPriority`**(uint32_t *irqn*, uint32_t *priority*)
set interrupt priority

`priority` depends on interrupt controller used on system. When GIC is used, `priority` should follow GIC requirement and meaning.

Return

- true on success
- false on invalid parameters

Parameters

- `irqn`: IRQ number
- `priority`: IRQ priority

uint32_t **`osiIrqGetPriority`**(uint32_t *irqn*)
get interrupt priority

When `irqn` is invalid, 0x80000000U will be returned.

Return IRQ priority

Parameters

- `irqn`: IRQ number

bool **`osiIrqPending`**(void)
check whether there are pending interrupt

This is for special purpose. It shall be called with interrupt disabled (not masked off).

This may be unimplemented in some chips.

Return

- true if there are interrupt pending.

void **`osiDCacheClean`**(**`const`** void **address*, size_t *size*)
clean (write back) D-cache

Usually it shall be called **before** the cachable memory will be read by not cache coherent hardware.

D-cache clean will operate by cache line. So, is the memory range is not cache line aligned, other memory on the cache line will be cleaned also. For example, assuming D-cache line size is 32 byte, `osiDCacheClean((void *)8, 32)` will clean [0-8], [40-64] also.

When D-cache coherence is needed to be considered, it is recommended to declare or allocate memory by the following:

```
char mem1[SIZE] OSI_CACHE_LINE_ALIGNED;
void *mem2 = memalign(CONFIG_CACHE_LINE_SIZE, size);
```

Parameters

- `address`: starting address to be cleaned
- `size`: size to be cleaned

void **`osiDCacheInvalidate`**(**`const`** void **address*, size_t *size*)
invalidate D-cache

Usually it shall be called **before** the cachable memory will be write by not cache coherent hardware.

D-cache line alignment is very important for `osiDCacheInvalidate`. Otherwise, other memory on the cache line will be changed randomly.

Parameters

- `address`: starting address to be cleaned
- `size`: size to be cleaned

void **osiDCacheCleanInvalidate**(**const** void **address*, size_t *size*)
clean and invalidate D-cache

Parameters

- `address`: starting address to be cleaned
- `size`: size to be cleaned

void **osiICacheInvalidate**(**const** void **address*, size_t *size*)
invalidate D-cache range

Parameters

- `address`: starting address to be cleaned
- `size`: size to be cleaned

void **osiICaheSync**(**const** void **address*, size_t *size*)
sync D-cahce with I-cache

It shall be called after code memory is operated as data (such as, copy codes from flash to RAM).

Parameters

- `address`: starting address to be cleaned
- `size`: size to be cleaned

void **osiDCacheCleanAll**(void)
clean all D-cache

void **osiDCacheInvalidateAll**(void)
invalidate all D-cache

void **osiDCacheCleanInvalidateAll**(void)
clean and invalidate all D-cache

void **osiICacheInvalidateAll**(void)
invalidate all I-cache

void **osiICacheSyncAll**(void)
sync D-cahce with I-cache for all

osiThread_t ***osiThreadCreate**(**const** char **name*, *osiThreadEntry_t entry*, void **argument*, uint32_t *priority*, uint32_t *stack_size*, uint32_t *event_count*)
create a thread

Each `osiThread_t` will have an event queue. So, the event queue depth should be specified at creation.

`name` will be copied to thread control block. So, `name` can be dynamic memory.

After a thread is created, it will be executed immediately. So, it is possible that `entry` will be executed before the return value is assigned to some variable, if the new thread priority is higher than current thread priority. Pay attention to use thread pointer in `entry`.

```
void entry(void *argument) {
    osiThread_t *thread = osiThreadCurrent();
    for (;;) {
        osiEvent_t event = {};
        osiEventWait(thread, &event);
        ......
    }
}
```

In some underlay RTOS, there are limitation on maximum stack size. For example, when `configSTACK_DEPTH_TYPE` is defined as `uint16_t`, the stack size must be less than 64KB*4.

Return

- thread pointer
- NULL if failed

Parameters

- `name`: thread name
- `entry`: thread entry function
- `argument`: thread entry function argument
- `priority`: thread priority
- `stack_size`: thread stack size in byte
- `event_count`: thread event queue depth (count of events can be hold)

osiThread_t ***osiThreadCreateWithStack**(**const** char **name*, *osiThreadEntry_t entry*, void **argument*, uint32_t *priority*, void **stack*, uint32_t *stack_size*, uint32_t *event_count*)

create a thread with specified stack

It is similar to `osiThreadCreate`, and the stack won't be dynamic created. Rather, the specified buffer `stack` will be used at the stack of the thread. Typical usage is to create performance sensitive thread, and set stack in SRAM to improve performance.

Application should always use `osiThreadCreate`. It may be unimplemented on some platforms.

Return

- thread pointer
- NULL if failed

Parameters

- `name`: thread name
- `entry`: thread entry function

- `argument`: thread entry function argument
- `priority`: thread priority
- `stack`: thread stack buffer, must be valid
- `stack_size`: thread stack size in byte
- `event_count`: thread event queue depth (count of events can be hold)

osiEventQueue_t ***osiThreadEventQueue**(*osiThread_t *thread*)
get event queue of thread

When `thread` is NULL, return the event queue of current thread.

Return event queue of the thread

Parameters

- `thread`: thread pointer

osiThread_t ***osiThreadCurrent**(void)
get current thread pointer

Return current thread pointer

void **osiThreadSetFPUEnabled**(bool *enabled*)
set whether current thread need FPU

By default, FPU isn't permitted for new created thread. To enable FPU, `osiThreadSetFPUEnabled(true)` should be called before floating point instructions.

Though it is possible to call `osiThreadSetFPUEnabled(false)` if it is known that floating point instructions won't be used any more, it is not necessary. It will only increase a little context save and restore cycles. Typical usage it to call `osiThreadSetFPUEnabled(true)` at the beginning of thread entry.

It is undefined if thread uses floating point instructions whthout call of `osiThreadSetFPUEnabled(true)`.

Parameters

- `enabled`: true for enable FPU, false for disable FPU.

uint32_t **osiThreadPriority**(*osiThread_t *thread*)
get thread priority

When `thread` is NULL, return the priotity of current thread.

Return priotity of the thread

Parameters

- `thread`: thread pointer

bool **osiThreadSetPriority**(*osiThread_t* *thread, uint32_t *priority*)
set thread priority

When `thread` is NULL, set the priotity of current thread.

After the priority changed, it is possible thread context switch will occur. However, don't depend on this feature.

Return

- true on success
- false on invalid parameter

Parameters

- `thread`: thread pointer
- `priority`: priority to be set

void **osiThreadSuspend**(*osiThread_t* *thread)
suspend a thread

When `thread` is NULL, suspend current thread.

Parameters

- `thread`: thread pointer

void **osiThreadResume**(*osiThread_t* *thread)
resume a thread

`thread` can't be NULL.

Parameters

- `thread`: thread pointer

void **osiThreadYield**(void)
current thread yield

When there are threads with the same priority, other threads will be scheduled.

void **osiThreadSleep**(uint32_t *ms*)
current thread sleep

Change current thread into sleep mode, and will be rescheduled after the specified period.

This will use the underlay RTOS mechanism. It is possible the sleep time precision is the tick of underlay RTOS.

Parameters

- `ms`: sleep time in milliseconds

void **osiThreadExit** (void)
current thread exit

When a thread is finished, `osiThreadExit` must be called. And kernel will release thread resources at appropriate time.

void **osiShowThreadState** (void)
show thread information through trace

It is for debug purpose only.

bool **osiEventSend** (*osiThread_t *thread*, **const** *osiEvent_t *event*)
send an event to a thread

At send, the body of `event` will be copied to event queue rather then send the pointer of `event`.

When event queue of the target thread is full, the caller thread will be block until there are rooms in target thread event queue.

Return

- true on success
- false on invalid parameter

Parameters

- `thread`: thread pointer, can't be NULL
- `event`: event to be sent

bool **osiEventTrySend** (*osiThread_t *thread*, **const** *osiEvent_t *event*, uint32_t *timeout*)
send an event to a thread with timeout

When `timeout` is 0, this will return false immediately. When `timeout` is `OSI_WAIT_FOREVER`, this will wait forever until there are rooms in target thread event queue.

This can be called in ISR. And in ISR, `timeout` must be 0.

Return

- true on success
- false on invalid parameter, or timeout

Parameters

- `thread`: thread pointer, can't be NULL
- `event`: event to be sent
- `timeout`: timeout in milliseconds

bool **osiEventWait** (*osiThread_t *thread*, *osiEvent_t *event*)
wait an event

Wait an event from current thread event queue. When current thread event queue is empty, it will be blocked forever until there are event in event queue.

The event body will be copied to `event`.

There are some event ID used by system. After system event is received and process, this will return an event with ID of 0. Application can ignore event ID 0 safely.

Return

- true on success
- false on invalid parameter

Parameters

- `thread`: thread pointer, can't be NULL
- `event`: event pointer

bool **`osiEventTryWait`** (*osiThread_t* *thread, *osiEvent_t* *event, uint32_t *timeout*)

wait an event with timeout

When `timeout` is 0, this will return false immediately. When `timeout` is `OSI_WAIT_FOREVER`, this will wait forever until there are events in target thread event queue.

Return

- true on success
- false on invalid parameter, or timeout

Parameters

- `thread`: thread pointer, can't be NULL
- `event`: event pointer
- `timeout`: timeout in milliseconds

bool **`osiThreadCallback`** (*osiThread_t* *thread, *osiCallback_t* *cb*, void **cb_ctx*)

set callback to be executed on thread

Thread callback is implemented by `osiEvent_t`.

This can be called in ISR. In ISR, the callback event will be lost when the event queue of target thread is full.

Return

- true on success
- false on invalid parameter, or event queue is full in ISR

Parameters

- `thread`: thread pointer, can't be NULL
- `cb`: callback to be executed
- `cb_ctx`: callback context

osiWork_t ***osiWorkCreate** (*osiCallback_t run*, *osiCallback_t complete*, void *ctx)
create a work

`run` can't be NULL, and `complete` can be NULL.

Return

- the created work
- NULL if invalid parameter or out of memory

Parameters

- `run`: execute function of the work
- `complete`: callback to be invoked after the work is finished
- `ctx`: context of `run` and `complete`

void **osiWorkDelete** (*osiWork_t *work*)
delete the work

When `work` is running when it is called, it will be deleted after the current run finished.

Parameters

- `work`: the work to be deleted

bool **osiWorkEnqueue** (*osiWork_t *work*, *osiWorkQueue_t *wq*)
enqueue a work in specified work queue

When `work` is running, it will be queued to again and then it will be invoked again.

When `work` is queued, and `wq` is the same as original work queue, nothing will be done. When `wq` is not the same as the original work queue, it will be removed from the original work queue, and queue the work into the specified work queue.

This can be called in ISR.

Return

- true on success
- false for invalid parameter, or work is running

Parameters

- `work`: the work pointer, must be valid
- `wq`: work queue to run the work, must be valid

void **osiWorkCancel** (*osiWork_t *work*)
cancel a work

When `work` is running, the current execution won't be interrupted.

Parameters

- `work`: the work pointer, must be valid

osiWorkQueue_t ***osiWorkQueueCreate**(**const** char **name*, size_t *thread_count*, uint32_t *priority*, uint32_t *stack_size*)

create work queue

Multiple threads can be created to reduce work execution latency. For example, when one thread in a work queue is blocked, other threads of the work queue can execute other works queued to the work queue.

The maximum thread count is implementation dependent. So, it is possible that the created thread count is less than `thread_count`. And also it is possible that `thread_count` is ignored.

The created threads have the same priority and stack size.

The work queue thread entry function can't be customized

Return

- the work queue pointer
- NULL if failed

Parameters

- `name`: work queue name
- `thread_count`: thread count to be created for the work queue
- `priority`: work queue thread priority
- `stack_size`: thread stack size in byte

void **osiWorkQueueDelete**(*osiWorkQueue_t* **wq*)

delete work queue

All resources of the work queue will be deleted.

The works in running will continue, and `complete` will be invoked as normal. However, queued work in the work queue won't be executed any more.

Parameters

- `wq`: work queue to be deleted

osiWorkQueue_t ***osiSysWorkQueueHighPriority**(void)

get the system high priority work queue

A work queue with priority `OSI_PRIORITY_HIGH` will be created at kernel start.

Return the system high priority work queue

osiWorkQueue_t ***osiSysWorkQueueLowPriority**(void)

get the system low priority work queue

A work queue with priority `OSI_PRIORITY_LOW` will be created at kernel start.

Return the system low priority work queue

osiWorkQueue_t ***osiSysWorkQueueFileWrite**(void)
get the system work queue for asynchronuous file system write

A work queue with priority `OSI_PRIORITY_BELOW_NORMAL` will be created at kernel start. This work queue shall be used for asynchronuous file system write.

Usually file write is slow, especially for file system on NOR flash. When faster response is needed, file write can be deferred to this work queue. Also, the work queue will be flushed before system shutdown.

Return the system file write work queue

osiNotify_t ***osiNotifyCreate**(*osiThread_t* **thread*, *osiCallback_t* *cb*, void **ctx*)
create a thread notify

Thread notify is thread callback with state to avoid duplicated event to be sent to thread event queue.

Return

- created notify
- NULL if parameters are invalid or out of memory

Parameters

- `thread`: thread to execute the callback, can't be NULL
- `cb`: callback to be executed, can't be NULL
- `ctx`: callback context

void **osiNotifyDelete**(*osiNotify_t* **notify*)
delete a thread notify

The memory of the thread notify will be released.

When the thread notify is already in thread event queue, the callback won't be invoked, and the memory may be released delayed.

Parameters

- `notify`: thread notify pointer, must be valid

void **osiNotifyTrigger**(*osiNotify_t* **notify*)
trigger a thread notify

When the thread notify event isn't in thread event queue, an event for thread notify will be queued to the tail of thread event queue.

Parameters

- `notify`: thread notify pointer, must be valid

void **osiNotifyCancel**(*osiNotify_t* **notify*)
cancel a thread notify

When the thread notify event has already sent to thread event queue, the invocation of the callback will be cancelled.

Parameters

- `notify`: thread notify pointer, must be valid

osiTimer_t ***osiTimerCreate**(*osiThread_t *thread*, *osiCallback_t cb*, void *ctx*)
create a timer

Create a timer with specified callback and callback context. After create, the timer is in stop state. Application can start it in various mode.

When `thread` is NULL, the callback will be executed in timer ISR. So, the callback should follow ISR programming guide.

When `thread` is not NULL, the callback will be executed in the specified thread through *event* mechanism.

It is needed to call `osiTimerDelete` to free resources.

Return

- the created timer instance
- NULL at out of memory, or invalid parameter

Parameters

- `thread`: thread to execute the callback, or NULL for execute in ISR
- `cb`: callback to be executed after timer expire
- `ctx`: callback context

osiTimer_t ***osiTimerEventCreate**(*osiThread_t *thread*, uint32_t *timerid*)
create a timer

Create a timer with specified timerid. After the timer expired, the specified thread will receive an event { EV_TIMER, timerid, 0, 0}.

It is needed to call `osiTimerDelete` to free resources.

Return

- the created timer instance
- NULL at out of memory, or invalid parameter

Parameters

- `thread`: thread to execute the callback, it can't be NULL
- `timerid`: timerid in expiration event

bool **osiTimerSetCallback**(*osiTimer_t *timer*, *osiThread_t *thread*, *osiCallback_t cb*, void *ctx*)
set/change callback of timer

In most cases, timer callback set at create is not needed to be changed.

Return

- true on success
- false on invalid parameter

Parameters

- `timer`: timer to be changed
- `thread`: thread to execute the callback, or NULL for execute in ISR
- `cb`: callback to be executed after timer expire
- `ctx`: callback context

void **osiTimerDelete** (*osiTimer_t *timer*)

delete a timer

Delete the timer, and free associated resources.

When the timer callback will be executed in thread rather than ISR, and the event for timer expiration has been sent, some resources won't be freed immediately. Rather, they will be freed after the timer expiration event is popped out from thread event queue. However, the callback won't be executed even delete is delayed.

Refer to document about corner case of timer thread callback.

Parameters

- `timer`: the timer to be deleted

bool **osiTimerStart** (*osiTimer_t *timer*, uint32_t *ms*)

start a timer

Start a timer, and the timer will be expired in specified period from now. After the callback is executed, the timer will come to stopped state automatically.

When the timer is in stated state before this function, the previous expiration won't be executed.

Refer to document about corner case of timer thread callback.

It is valid that the timeout period is 0. In this case, the timer will expire very soon.

Due to timeout is 32bits of milliseconds, The maximum timeout period is ~50 days.

Return

- true on success
- false on invalid parameter

Parameters

- `timer`: the timer to be started
- `ms`: timeout period

bool **osiTimerStartRelaxed**(*osiTimer_t* *timer, uint32_t *ms*, uint32_t *relax_ms*)
start a timer with relaxed timeout

It is a power optimization version. Due to power saving, it is possible the callback will be executed later then *normal timeout*, but it will be executed before *relaxed timeout* event system will enter suspend.

When `relax_ms` is `OSI_DELAY_MAX`, it means the timer can be ignored completed for power saving. However, after system is awoken, the callback will be executed still if the *normal timeout* is expired.

Relaxed timeout must be greater than normal timeout period.

Return

- true on success
- false on invalid parameter

Parameters

- `timer`: the timer to be started
- `ms`: normal timeout period
- `relax_ms`: relaxed timeout period

bool **osiTimerStartHWTickRelaxed**(*osiTimer_t* *timer, uint32_t *ticks*, uint32_t *relax_ticks*)
start a timer with relaxed timeout, in unit of hardware tick

Don't call this in application code. It is only for legacy codes.

The frequency of hardware tick is chip dependent, and implementation dependent.

Return

- true on success
- false on invalid parameter

Parameters

- `timer`: the timer to be started
- `ticks`: normal timeout period in hardware tick
- `relax_ticks`: relaxed timeout period in hardware tick

bool **osiTimerStartMicrosecond**(*osiTimer_t* *timer, uint32_t *us*)
start a timer, timeout in unit of microseconds

Timeout in microseconds can support higher precision. However, the maximum timeout is shorter, ~1.2 hours.

The real precision depends on hardware.

Return

- true on success
- false on invalid parameter

Parameters

- `timer`: the timer to be started
- `us`: timeout period in microseconds

bool **osiTimerStartPeriodic**(*osiTimer_t* *timer, uint32_t *ms*)

start a periodic timer

The periodic interval can't be 0.

Return

- true on success
- false on invalid parameter

Parameters

- `timer`: the timer to be started
- `ms`: interval in microseconds

bool **osiTimerStartPeriodicRelaxed**(*osiTimer_t* *timer, uint32_t *ms*, uint32_t *relaxed_ms*)

start a periodic timer with relaxed timeout

The periodic interval can't be 0.

When `relax_ms` is `OSI_DELAY_MAX`, it means the timer can be ignored completed for power saving. However, after system is awoken, the callback will be executed still if the *normal timeout* is expired.

Return

- true on success
- false on invalid parameter

Parameters

- `timer`: the timer to be started
- `ms`: interval in microseconds
- `relaxed_ms`: relaxed timeout period

bool **osiTimerStop**(*osiTimer_t* *timer)

stop a time

Stop a not-started or stopped timer is valid, just do nothing.

Refer to document about corner case of timer thread callback.

Return

- true on success
- false on invalid parameter

Parameters

- `timer`: the timer to be stopped

bool **osiTimerLightSleep**(uint32_t *idle_tick*)
tickless light sleep for timer

When OS tick is implemented in timer, `idle_tick` is the maximum idle OS tick count (not hardware tick count) for sleep.

The timer interrupt will be moved to minimum of:

- OS timer, after `idle_tick`
- other timers timeout time

When there are no timers, timer interrupt will be disabled.

The normal timeout time of timer, rather than the relaxed timeout time of timer, will be used.

When the timer interrupt is moved, it will return true. In that case, the timer interrupt will be moved back after light sleep.

Return

- true if timer interrupt is moved = false if timer interrupt is not moved

Parameters

- `idle_tick`: OS tick sleep count

int64_t **osiTimerDeepSleepTime**(uint32_t *idle_tick*)
calculate the deep sleep time of all timers

When OS tick is implemented in timer, `idle_tick` is the maximum idle OS tick count (not hardware tick count) for sleep.

When there are timers which will wakeup system, the return value is the earliest timer wakeup time from now in milliseconds.

When there are no timer will wakeup system, return `INT64_MAX`.

The relaxed timeout time of timer, rather than the normal timeout time, is used in checking sleep time.

Usually, it will be called by system sleep module. And it is not prohibited to be called in other cases.

It **must** be called with interrupt disabled. The implementation won't perform protection.

Return

- the earliest sleep timer from now in milliseconds. `INT64_MAX` for no need to wakeup.

Parameters

- `idle_tick`: OS tick sleep count

int64_t **osiTimerPsmWakeUpTime**(void)
calculate the PSM wakup time of all timers

When OS tick is implemented in timer, OS tick is not considered. That is, PSM wont't be waken by OS tick.

The there are times which will wakeup system, the return value is the earliest timer wakeup time in milliseconds.

When there are no timer will wakeup system, return `INT64_MAX`.

The relaxed timeout time of timer, rather than the normal timeout time, is used in checking sleep time.

It **must** be called with interrupt disabled. The implementation won't perform protection.

Return

- the earliest wakeup timer from now in milliseconds. `INT64_MAX` for no need to wakeup.

void **osiTimerWakeupProcess**(void)
timer module processing after wakeup

In case `osiUpHWTick` will be discontinued at sleep, it should be called after `osiUpHWTick` is stable.

Usually, it will be called by system sleep module. And it is not prohibited to be called in other cases.

It **must** be called with interrupt disabled. The implementation won't perform protection.

osiTimerPool_t ***osiTimerPoolCreate**(void)
create a timer pool

Return

- timer pool instance
- NULL if out of memory

void **osiTimerPoolDelete**(*osiTimerPool_t* **pool*)
delete a timer pool

All timers created inside the timer pool will be deleted also.

Parameters

- `pool`: timer pool instance

void **osiTimerPoolGC**(*osiTimerPool_t* **pool*)
delete not used timers in timer pool

When `pool` is NULL, the global timer pool will be used.

Parameters

- `pool`: timer pool instance

bool **osiTimerStartFromPool**(*osiTimerPool_t* **pool*, uint32_t *ms*, *osiThread_t* **thread*, *osiCallback_t* *cb*, void **ctx*)
start a timer from pool

Start the timer which matches all of the callback thread, callback function and callback context. When there are no matched timer, a timer will be created and started.

When `pool` is NULL, the global timer pool will be used.

Return

- the find or created timer
- NULL if out of memory

Parameters

- `pool`: the timer pool, NULL for the global timer pool
- `ms`: timeout period
- `thread`: timer callback thread, NULL for executed in ISR
- `cb`: timer callback function
- `ctx`: timer callback context

bool **osiTimerStartFromPoolRelaxed**(*osiTimerPool_t* *pool, uint32_t *ms*, uint32_t *relaxed_ms*, *osiThread_t* *thread, *osiCallback_t* *cb*, void *ctx)

start a timer from pool with relaxed timeout

Start the timer which matches all of the callback thread, callback function and callback context. When there are no matched timer, a timer will be created and started.

When `pool` is NULL, the global timer pool will be used.

Return

- the find or created timer
- NULL if out of memory

Parameters

- `pool`: the timer pool, NULL for the global timer pool
- `ms`: timeout period
- `relax_ms`: relaxed timeout period
- `thread`: timer callback thread, NULL for executed in ISR
- `cb`: timer callback function
- `ctx`: timer callback context

void **osiTimerStopFromPool**(*osiTimerPool_t* *pool, *osiThread_t* *thread, *osiCallback_t* *cb*, void *ctx)

stop a timer from pool

Stop the timer which matches all of the callback thread, callback function and callback context.

When `pool` is NULL, the global timer pool will be used.

Return

- the find or created timer
- NULL if out of memory

Parameters

- `pool`: the timer pool, NULL for the global timer pool
- `thread`: timer callback thread, NULL for executed in ISR
- `cb`: timer callback function
- `ctx`: timer callback context

int **osiTimerDump**(void **mem*, int *count*)

dump timer information to memory

It is for debug only. The data format of timer information dump is not stable, and may change. Currently, there is 10 bytes header, and 20 bytes for each timer.

Caller should make sure `mem` is enough for hold timer information of `count`.

Return dump memory size

Parameters

- `mem`: memory for timer information dump
- `count`: maximum timer count to be dump

osiMessageQueue_t ***osiMessageQueueCreate**(uint32_t *msg_count*, uint32_t *msg_size*)

create a message queue

Return

- message queue pointer
- NULL on invalid parameter or out of memory

Parameters

- `msg_count`: maximum message count can be hold in queue
- `msg_size`: size of each message in bytes

void **osiMessageQueueDelete**(*osiMessageQueue_t* **mq*)

delete a message queue

When `mq` is NULL, nothing will be done, just as `free`.

Parameters

- `mq`: message queue pointer

bool **osiMessageQueuePut**(*osiMessageQueue_t* *mq, **const** void *msg)
put a message to message queue

`msg` should hold content size the same as `msg_size` specified at `osiMessageQueueCreate`.

After put, the content of `msg` will be copied to message queue.

When `mq` is full, it will be blocked until there are rooms.

Return

- true on success
- false on invalid parameter

Parameters

- `mq`: message queue pointer
- `msg`: mesage pointer

bool **osiMessageQueueTryPut**(*osiMessageQueue_t* *mq, **const** void *msg, uint32_t *timeout*)
put a message to message queue with timeout

This can be called in ISR. And in ISR, `timeout` must be 0.

Return

- true on success
- false on invalid parameter or timeout

Parameters

- `mq`: message queue pointer
- `msg`: mesage pointer
- `timeout`: timeout in milliseconds

bool **osiMessageQueueGet**(*osiMessageQueue_t* *mq, void *msg)
get a message to message queue

`msg` should be able tp hold content size of `msg_size` specified at `osiMessageQueueCreate`.

After get, the content of message will be copied to `msg`.

When `mq` is empty, it will be blocked until there are messages.

Return

- true on success
- false on invalid parameter

Parameters

- `mq`: message queue pointer
- `msg`: mesage pointer

bool **osiMessageQueueTryGet**(*osiMessageQueue_t* *mq, void *msg, uint32_t *timeout*)
get a message to message queue with timeout

This can be called in ISR. And in ISR, `timeout` must be 0.

Return

- true on success
- false on invalid parameter or timeout

Parameters

- `mq`: message queue pointer
- `msg`: mesage pointer
- `timeout`: timeout in milliseconds

osiSemaphore_t ***osiSemaphoreCreate**(uint32_t *max_count*, uint32_t *init_count*)
create a semaphore

When `max_count` is 1, it is a binary semaphore. Otherwise, it is counting semaphore.

Return

- semaphore pointer
- NULL on invalid parameter or out of memory

Parameters

- `max_count`: maximum count of the semaphore
- `init_count`: initial count of the semaphore

void **osiSemaphoreDelete**(*osiSemaphore_t* *semaphore)
delete the semaphore

When there are blocked thread on the semaphore, the behavior is undefined for `osiSemaphoreDelete`.

Parameters

- `semaphore`: semaphore pointer

void **osiSemaphoreAcquire**(*osiSemaphore_t* *semaphore)
acquire from semaphore

After acquire, the count of semaphore will be decreased by 1.

When the count of semaphore is 0, it will be blocked until the count becomes non-zero (increased by `osiSemaphoreRelease`).

Parameters

- `semaphore`: semaphore pointer

bool **osiSemaphoreTryAcquire**(*osiSemaphore_t* *semaphore, uint32_t timeout)
acquire from semaphore with timeout

This can be called in ISR. And in ISR, `timeout` must be 0.

Return

- true on success
- false on timeout

Parameters

- `semaphore`: semaphore pointer
- `timeout`: timeout in milliseconds

void **osiSemaphoreRelease**(*osiSemaphore_t* *semaphore)
release to semaphore

After release, the count of semaphore will be increased by 1. When there are blocked thread on the semaphore, one of the blocked thread will be unblocked.

This can be called in ISR.

Parameters

- `semaphore`: semaphore pointer

osiMutex_t ***osiMutexCreate**(void)
create a mutex

After creation, the mutex is in *open* state.

Return

- mutex pointer
- NULL on out of memory

void **osiMutexDelete**(*osiMutex_t* *mutex)
delete the mutex

When `mutex` is NULL, nothing will be done, just as `free`.

Parameters

- `mutex`: mutex pointer to be deleted

void **osiMutexLock**(*osiMutex_t* *mutex)
lock the mutex

When `mutex` is locked by another thread, it will wait forever until the mutex is unlocked.

Parameters

- `mutex`: mutex pointer to be locked

bool **osiMutexTryLock** (*osiMutex_t* *mutex, uint32_t *timeout*)
lock the mutex with timeout

Return

- true on success
- false on timeout

Parameters

- `mutex`: mutex pointer to be locked
- `timeout`: timeout in milliseconds

void **osiMutexUnlock** (*osiMutex_t* *mutex)
unlock the mutex

Parameters

- `mutex`: mutex pointer to be unlocked

int64_t **osiUpTime** (void)
monoclinic system time

It is a relative time from system boot. Even after suspend and resume, the monoclinic system time will be contiguous.

Return monoclinic system time in milliseconds

int64_t **osiUpTimeUS** (void)
monoclinic system time in microsecond

The monoclinic system time in unit of microsecond.

Return monoclinic system time in microseconds

void **osiSetUpTime** (int64_t *ms*)
set monoclinic system time

When it is known that the hardware resource for monoclinic system time is discontinued, such as power off during deep sleep, this is called to set monoclinic system time.

When up time is changed, epoch time and local time aren't be changed.

It should only be called in system integration.

Parameters

- `ms`: target monoclinic system time in microseconds

int64_t **osiEpochTime**(void)
get the epoch time

The time is millisecond (1/1000 second) from 1970-01-01 UTC. To avoid overflow, the data type is 64 bits.

Epoch time is not monoclinic time. For example, when system time is synchronized with network, there may exist a jump (forward or backward) of epoch time.

In 2 cases, this system time may be not reliable:

- During boot, and before RTC is initialized.
- During wakeup, and the elapsed sleep time hasn't compensated.

Return epoch time in milliseconds

int64_t **osiEpochSecond**(void)
get the epoch time in second

The time is seconds from 1970-01-01 UTC. To avoid overflow, the data type is 64 bits.

- signed 32bits will overflow at year 2038
- unsigned 32bits will overflow at year 2106

Epoch time is not monoclinic time. For example, when system time is synchronized with network, there may exist a jump (forward or backward) of epoch time.

In 2 cases, this system time may be not reliable:

- During boot, and before RTC is initialized.
- During wakeup, and the elapsed sleep time hasn't compensated.

Return epoch time in seconds

void **osiSetEpochTime**(int64_t *ms*)
set the monoclinic system time

After the system time is changed, RTC won't be updated automatically. It is needed to call RTC API to sync system time to RTC.

When epoch time is changed, the monoclilinc system up time isn't changed, and local time is changed correspondingly. The delta bewteen epoch time and local time is only affected by timezone.

Parameters

- `ms`: epoch time in milliseconds

int **osiTimeZoneOffset**(void)
get time zone in second

Time zone is the offset between local time and epoch time: local_time = epoch_time + time_zone

OSI won't keep time zone at power off. Other module should store time zone in NVRAM, and set to OSI at boot.

Return time zone in second

void **osiSetTimeZoneOffset**(int *offset*)
set time zone in second

Time zone is the offset between local time and epoch time: local_time = epoch_time + time_zone

Time zone should be in [-12*3600, 12*3600]. However, it is not checked inside this. The caller should make sure the `offset` is reasonable.

Parameters

- `offset`: time zone in second

int64_t **osiLocalTime**(void)
get the local time

It is just: epoch_time + time_zone

Return local time in milliseconds

int64_t **osiLocalSecond**(void)
get the local time in seconds

It is just: epoch_time + time_zone

Return local time in seconds

int64_t **osiUpHWTick**(void)
monoclinic system hardware tick

Don't call this in application code. It is only for legacy codes.

The frequency of hardware tick is chip dependent, and implementation dependent.

Return monoclinic system hardware tick

uint32_t **osiHWTickCount**(void)
raw hardware tick count

The frequency of hardware tick is chip dependent. And in some chips, the tick may be discontinuous at suspend resume. So, this API shall only be used for quick debug purpose.

The return value should be fulll 32bits value. That is, the next tick of 0xffffffff will be 0. Then the simple substract will always provide the tick delta.

Return hardware tick count

int64_t **osiEpochToUpTime**(int64_t *epoch*)
convert epoch time to up time

When some values has special meanings, such as INT64_MAX means invalid or not exist, caller should check special values before call this.

Return up time in milliseconds

Parameters

- `epoch`: epoch time in milliseconds

void **osiElapsedTimerStart** (*osiElapsedTimer_t *timer*)
start couting of elsapsed timer

Parameters

- `timer`: elapsed timer, must be valid

uint32_t **osiElapsedTime** (*osiElapsedTimer_t *timer*)
elapsed milliseconds after start

Return

- elapsed milliseconds after start

Parameters

- `timer`: elapsed timer, must be valid

uint32_t **osiElapsedTimeUS** (*osiElapsedTimer_t *timer*)
elapsed microseconds after start

Return

- elapsed microseconds after start

Parameters

- `timer`: elapsed timer, must be valid

int64_t **osiUpTimeToEpoch** (int64_t *uptime*)
convert uptime to epoch time

When some values has special meanings, such as INT64_MAX means invalid or not exist, caller should check special values before call this.

Return epoch time in milliseconds

Parameters

- `epoch`: up time in milliseconds

uint32_t **osiPmCpuSuspend** (*osiSuspendMode_t mode*, int64_t *ms*)
CPU suspend.

Usually, it shouldn't be called directly by application. Rather, it is implemented by BSP, and called by `osiPmSuspend`.

Return suspend resume source. `OSI_RESUME_ABORT` bit indicates suspend aborted.

Parameters

- `mode`: suspend mode
- `ms`: suspend time in milliseconds

void **osiPmInit**(void)
power management module initialization

It should be called in early stage of boot, due to many other modules may register suspend and resume callback at initialization.

void **osiPmStart**(void)
power management core start

To avoid suspend too early, PM core won't be started at initialization automatically. After system is initialized, and necessary PM sources are created, `osiPmStart` shall be called.

Only after `osiPmStart` is called, PM core will check and enter suspend.

void **osiPmStop**(void)
power management core stop

Stop PM core, and systen will never suspend.

It is only for debug. And it shouldn't be used in real application.

osiPmSource_t ***osiPmSourceCreate**(uint32_t *tag*, **const** *osiPmSourceOps_t* **ops*, void **ctx*)
create PM source

Modules are distinguished by FOURCC *tag*. So, tag should be unique system wise.

When the *tag* is already registered, it will return existed PM source.

`ops` is permitted to be NULL, and all callbacks inside it are permitted to be NULL.

Resume callbacks are called in create order, suspend callbacks are called in revered order. When resume order does matter, call `osiPmResumeReorder` or `osiPmResumeFirst` to change resume order.

The returned pointer should be destroyed and freed by `osiPmSourceDelete`.

PM source lock and unlock is binary. That is, there is no *counter* inside. When `osiPmWakeUnlock` is called, the PM source won't prevent system suspend, no matter how many times of `osiPmWakeLock` is called.

Return

- PM source pointer
- NULL on out of memory

Parameters

- `tag`: module tag
- `ops`: callbacks during suspend and resume
- `ctx`: callback context, shared by all callbacks

void **osiPmSourceDelete**(*osiPmSource_t *ps*)
destroy PM source

Parameters

- ps: PM source

void **osiPmResumeReorder**(uint32_t *tag_later*, uint32_t *tag_earlier*)
ensure resume callback order

In cases that the resume callback order is important, this API will check and change resume callback order if needed. After the call, the resumed order is ensured.

When resume callback order is changed, suspend callback order is changed also.

It won't be checked whether the tags are valid.

Parameters

- tag_later: module tag to be resumed laster
- tag_earlier: module tag to be resumed earlier

void **osiPmResumeFirst**(uint32_t *tag*)
move PM source to the first in resume order

Find the PM source, and move it to the head of resume list. When PM source with tag is not found in resume list, nothing will be done.

Parameters

- tag: module tag to be moved

void **osiPmWakeLock**(*osiPmSource_t *ps*)
wake lock to prevent suspend

Indicate the PM source will prevent system suspend. Due to PM source wake lock is counting, successive call of osiPmWakeLock will increase the internal counter. And the lock count won't exceed the max_count specified at create.

When ops.prepare is not NULL, the internal state of PM source is *suspend possible* rather than *active*. When system wants to suspend, ops.prepare will be called to double check whether suspend is permitted.

Parameters

- ps: PM source

void **osiPmWakeUnlock**(*osiPmSource_t *ps*)
wake unlock to permit suspend

Lock and Unlock are counted. That is, only when the unlock count reach the lock count, this PM source will permit suspend.

When Unlock is called in unlocked state, it will return silently.

Parameters

- `ps`: PM source

void **osiPmSleep**(uint32_t *idle_tick*)
power management suspend

Usually, it shouldn't be called directly by application. Rather, it will be called after suspend criteria are met.

int **osiPmSourceDump**(void **mem*, int *count*)
dump PM source information to memory

It is for debug only. The data format of timer information dump is not stable, and may change. Currently, there is 4 bytes header, and 4 bytes for each PM source.

Caller should make sure `mem` is enough for hold PM source information of `count`.

Return

- dump memory size
- -1 if `count` is too small

Parameters

- `mem`: memory for PM source information dump
- `count`: maximum PM source count to be dump

bool **osiRegisterShutdownCallback**(*osiShutdownCallback_t cb*, void **ctx*)
register a shutdown callback

When the callback and context is already registered, it will return false.

The order to invoke callbacks is undefined.

Don't call `osiRegisterShutdownCallback` or `osiUnregisterShutdownCallback` inside the callbacks.

Return

- true if callback registered
- false if callback is already registered, or callback is NULL

Parameters

- `cb`: shutdown callback to be registered, can't be NULL
- `ctx`: shutdown callback context

bool **osiUnregisterShutdownCallback**(*osiShutdownCallback_t cb*, void **ctx*)
unregister a shutdown callback

Return

- true if callback unregistered

- false if callback isn't found

Parameters

- `cb`: shutdown callback to be unregistered, can't be NULL
- `ctx`: shutdown callback context

int64_t **osiGetPsmWakeUpTime**(void)
get PSM expiration time in uptime

Return

- PSM expiration time in uptime
- INT64_MAX if PSM not enabled

void **osiSetPsmWakeUpTime**(int64_t *uptime*)
set PSM expiration time in uptime

Parameters

- `uptime`: PSM expiration time in uptime or INT64_MAX to disable PSM

void **osiSetPsmSleepTime**(int64_t *ms*)
set PSM sleep time from now

Parameters

- `ms`: PSM sleep time from now or INT64_MAX to disable PSM

int64_t **osiGetPsmElapsedTime**(void)
get PSM elapsed time in milliseconds

The starting point is the time `osiSetPsmWakeUpTime` or `osiSetPsmSleepTime` is called. And this returns the elapsed time from the starting point to now.

Return PSM elsapsed time

bool **osiShutdown**(*osiShutdownMode_t mode*)
shutdown to specified mode

When parameter is wrong, it will return false:

- When `mode` is OSI_SHUTDOWN_PSM_SLEEP, and PSM isn't enabled. Caller should check whether PSM is enabled before calling `osiShutdown`.
- When `mode` is not supported in the platform.

It never return true.

Return

- false on parameter error

Parameters

- `mode`: shutdown mode

uint32_t **osiGetBootCauses**(void)
get boot causes

It is possible there are multiple boot causes. The returned value is bit or of all boot causes.

It is possible hardware registers will be cleared after accessed. So, always call `osiGetBootCauses` to get the boot causes, rather than accessing hardware registers directly.

Return boot causes

void **osiSetBootCause**(*osiBootCause_t cause*)
set a boot cause

It is intended only for system integration.

Parameters

- `cause`: boot cause

void **osiClearBootCause**(*osiBootCause_t cause*)
clear a boot cause

It is intended only for system integration.

Parameters

- `cause`: boot cause

osiBootMode_t **osiGetBootMode**(void)
get boot mode

Return boot mode

void **osiSetBootMode**(*osiBootMode_t mode*)
set boot mode

It is intended only for system integration. At calling, caller should take care conflict of other boot mode detection.

Parameters

- `mode`: boot mode

void **osiPsmSavePrepare**(*osiShutdownMode_t mode*)
PSM save preparation.

It will be called before any `osiPsmDataSave`. Usually, it will prepare PSM data memory.

Parameters

- `mode`: shutdown mode

void **osiPsmSave**(*osiShutdownMode_t mode*)
PSM save.

Save all owner's PSM data to persistent storage.

Parameters

- `mode`: shutdown mode

void **osiPsmRestore**(void)
PSM restore.

It should be called in system initialization, after file system initialization.

bool **osiPsmDataSave**(*osiPsmDataOwner_t owner*, **const** void **buf*, uint32_t *size*)
save PSM data

PSM data save is implemented in platform. PSM data should be saved in persistent storage, which can be flash or aon memory. Caller shouldn't assume the storage type.

It should only be called in `osiShutdown` callbacks.

For each owner, `osiPsmDataSave` shall be called only once at most.

It is recommended to use fixed size buffer, which is the same at each PSM shutdown. However, variable size buffer is also supported.

As a special case, `size` of 0 is permitted. It just keep record that the owner is saved without real data.

Return

- true on success
- false on fail
  - duplicated owner
  - out of memory

Parameters

- `owner`: PSM data owner
- `buf`: owner's PSM data buffer
- `size`: owner's PSM data buffer size

int **osiPsmDataRestore**(*osiPsmDataOwner_t owner*, void **buf*, uint32_t *size*)
restore PSM data

Restore PSM data. At PSM resume boot, PSM data owner calls this to get the data saved by `osiPsmDataSave`

The buffer size should be equal or larger than the saved size. If it is smaller than saved size, return -1.

When `buf` is NULL, it will return PSM data size without copy. It can be used to get the PSM data size.

Return

- PSM data size on success
- 0 if there are no PSM data
- -1 on error

Parameters

- `owner`: PSM data owner
- `buf`: buffer for owner's PSM data
- `size`: buffer size

void **osiDebugEvent**(uint32_t *event*)
send out debug event to trace tool

If system and trace tool support debug event, this will send a word to trace tool.

It should be only used in quick debug. It is possible that platform can't support debug event, and debug event output may be turned off by compiling option.

Parameters

- `event`: word which will appear on trace tool

void **osiPanic**(void)
panic

Called on fatal error, and system can't go on.

It is different from `assert`. It will always cause system panic, and there are no compiling option to ignore it.

bool **osiIsPanic**(void)
whether system is in panic mode

In panic mode, system will still run a small daemon. And there is no interrupt in panic mode. So, features used in panic daemon shall take care of this.

Return

- true if system is in panic mode
- false if system is not in panic mode

void **osiDelayUS**(uint32_t *us*)
busy loop delay

The precision of delay depends on platform. And it will ensure to delay at least the specified time.

Parameters

- `us`: delay time in microseconds

struct osiEvent
#include <osi_api.h> event, with ID and 3 parameters

Public Members

uint32_t **id**
event identifier

uint32_t **param1**
1st parameter

uint32_t **param2**
2nd parameter

uint32_t **param3**
3rd parameter

struct osiPmSourceOps
#include <osi_api.h> PM source callbacks.

All the callbacks are called with interrupt disabled. So, it is not needed to call `osiEnterCritical` or `osiIrqSave` for protection. And it is not error to call them.

Don't call blocking API in the callback.

Public Members

void (***suspend**)(void *ctx, *osiSuspendMode_t* mode)
callback to be called before suspend

void (***resume**)(void *ctx, *osiSuspendMode_t* mode, uint32_t source)
callback to be called after resume

bool (***prepare**)(void *ctx)
callback to be called at suspend check

void (***prepare_abort**)(void *ctx)
callback to be called at suspend check fail

CHAPTER

FIVE

OSI LIBRARIES

Contents

5.1 Overview

There are several utilities inside OSI, though they are not *kernel* functionality.

5.2 Event Hub and Event Dispatch

Event hub is a map from event ID to event handler. When an event arrives, the handler can be found from event hub. Functionally, it is same as huge *switch/case*. However, the code will be more readable, and more flexible. Usually, the registration is event hub is static.

Event dispatch is for dynamic event handling. In case that there are several event handler for the same event ID depends on thread state, the handler can be registered to event dispatch as event callback. When the matched event

arrives, the registered callback will be invoked. After the callback is invoked, the callback will be unregistered automatically.

Event dispatch shall only be used that the event is predictable.

Both event hub and event dispatch are helper to decouple modules inside one thread. Without the helpers, there will exist a central *switch/case*, and the *switch/case* should know everything about modules under the thread.

Both event hub and event dispatch are *not* thread safe. They are designed to be called in one thread.

Note: Event ID 0 and 0xFFFFFFFF can't be registered.

5.3 FIFO

FIFO is a simple data structure for first-in-first-out. FIFO is designed to safe only if `get` or `put` are called in one context. Examples of multiple contexts:

- call in multiple thread;
- call in both thread and ISR;
- call in multiple ISR;

When it is really needed to be called in multiple contexts, caller should add protection.

5.4 Value String Map

Value string map is a simple unsigned int to string table. There are 2 usage models:

sorted array

Typical usage is to translate an ID to string. And usually it is constant.

When it constant and pre-sorted, it is RAM efficient and the performance of `osiVsmapBsearch` is good.

NULL string terminated

It is used to small table. The search is NULL string (not empty string) terminated. It is used in case performance is unimportant.

5.5 Memory Recycler

Memory recycler is for *delete later*. After pointers are put to memory recycler and before empty the memory recycler, the pointers are still valid. And at empty the memory recycler, all pointers inside will be freed.

Typically, it is used in event loop style thread. At event handling functions, temporal pointers which should be freed after the event loop can be put to the memory recycler bind to the thread. And at the end of event loop, the thread memory recycler should be emptied.

It is helpful to avoid memory leak. To avoid memory leak, dynamic allocated memory should be freed at every return point. Such as:

```
void *mem = malloc(size);
if (error_1)
{
    free(mem);
    return;
}
if (error_2)
{
    free(mem);
    return;
}
// ......
```

With memory recycler, it can be rewritten to:

```
void *mem = malloc(size);
osiMemRecyclerPut(recycler, mem);
if (error_1)
    return;
if (error_2)
    return;
// ......
```

The code is much clearer, and it possibility of memory leak will be decreased.

Due to the pointer inside memory recycler will be freed later then the pointer won't be used. So, it is possible that more dynamic memory will consumed by using memory recycler.

Note: Don't put pointers of large memory block in memory recycler. It should be freed manually as early as possible to avoid out of memory.

The time to empty memory recycler. If the pointer is accessed after the memory recycler is emptied, system may crash. It is the user's duty to empty the memory recycler in correct time. Typical usage is to use it for event handling temporal memory, and free at the end of even loop:

```
void thread_entry(void *argument)
{
    osiThread_t *thread = osiThreadCurrent();
    for (;;)
    {
        osiEvent_t event = {};
        osiEventWait(thread, &event);
        process_event(&event);

        osiMemRecyclerEmpty(thread_mem_recycler);
    }
}
```

There are no global memory recycler.

Memory recycler is designed to be used in one thread, and it is *not thread safe*.

5.6 Generic List

Macros in <sys/queue.h> are extensively used. However, when data struct implementation is not wanted to put in the public header, macros for SLIST, TAILQ and etc and be used also.

To reduce public header complexity, a generic SLIST and TAILQ data struct are defined. For example, in implementation:

```
struct slistItem
{
    SLIST_ENTRY(slistItem) iter;
    int field;
    // more fields
};
```

And in public header:

```
typedef struct slistItem slistItem_t;
void processItem(slistItem_t *item);
osiSlistHead_t *getList(void);
```

Then the APIs can be used as without implementation details:

```
osiSlistHead_t *list = getList();
osiSlistItem_t *item;
SLIST_FOREACH(item, list, iter)
{
    slistItem_t *item_imp = (slistItem_t *)item;
    processItem(item_imp);
}
```

Note: To make data struct cast work, the first field of implementation **must** be SLIST_ENTRY (or TAILQ_ENTRY and etc).

5.7 Event Hub and Dispatch API Reference

Typedefs

typedef struct osiEventDispatch **osiEventDispatch_t**
opaque data structure for event dispatch

typedef struct osiEventHub **osiEventHub_t**
opaque data structure for event hub

typedef void (***osiEventHandler_t**)(**const** *osiEvent_t* *event)
function type of event handler

typedef void (***osiEventCallback_t**)(void *ctx, **const** *osiEvent_t* *event)
function type of event callback

Comparing to `osiEventHandler_t`, there is a callback context pointer.

Functions

osiEventHub_t ***osiEventHubCreate**(size_t *depth*)
create an event hub

To avoid dynamic memory complexity, static memory with specified `depth` will be used.

Return

- event hub pointer
- NULL if out of memory

Parameters

- `depth`: maximum registry count

void **osiEventHubDelete**(*osiEventHub_t* **p*)
delete the event hub

When `p` is NULL, nothing will be done.

Parameters

- `p`: the event hub

bool **osiEventHubRegister**(*osiEventHub_t* **p*, uint32_t *id*, *osiEventHandler_t handler*)
register an event handler to event hub

When the event ID is already registered, it will be replaced silently.

Though `handler` of NULL is not useful, it is permitted.

Return

- true on success
- false on too many registrations

Parameters

- `p`: the event hub, must be valid
- `id`: registered event ID
- `handler`: event handler

bool **osiEventHubBatchRegister** (*osiEventHub_t* *p, uint32_t *id*, ...)
batch register events handler to event hub

The variadic parameters are pairs of (id, handler), and ended with id of zero.

Return

- true on success
- false on too many registrations

Parameters

- `p`: the event hub, must be valid
- `id`: registered event ID

bool **osiEventHubVBatchRegister** (*osiEventHub_t* *p, uint32_t *id*, va_list *args*)
batch register events handler to event hub

It is the same as `osiEventHubBatchRegister`.

Return

- true on success
- false on too many registrations

Parameters

- `p`: the event hub, must be valid
- `id`: registered event ID
- `args`: variadic parameters

bool **osiEventHubRun** (*osiEventHub_t* *p, **const** *osiEvent_t* *event)
dispatch an event

Found the registered event handler, and invoke the handler.

Handler of NULL is regarded as unregistered, and return false.

Return

- true if the registered handler is invoked
- false if the event is not registered

Parameters

- `p`: the event hub, must be valid
- `event`: event to be dispatched

osiEventDispatch_t ***osiEventDispatchCreate** (size_t *depth*)
create an event hdispatch

To avoid dynamic memory complexity, static memory with specified `depth` will be used.

Return

- event dispatch pointer
- NULL if out of memory

Parameters

- `depth`: maximum registry count

void **osiEventDispatchDelete**(*osiEventDispatch_t* *p)

delete the event dispatch

When `p` is NULL, nothing will be done.

Parameters

- `p`: the event dispatch

bool **osiEventDispatchRegister**(*osiEventDispatch_t* *p, uint32_t *id*, *osiEventCallback_t* *cb*, void *cb_ctx)

register an event handler to event hub

When the event ID is already registered, it will return false.

Return

- true on success
- false on fail
  - too many registrations
  - event already registered
  - invalid event callback

Parameters

- `p`: the event dispatch, must be valid
- `id`: registered event ID
- `cb`: event callback, can't be NULL
- `cb_ctx`: event callback context

bool **osiEventDispatchRun**(*osiEventDispatch_t* *p, **const** *osiEvent_t* *event)

dispatch an event

Found the registered event callback, and invoke the callback.

Return

- true if the registered callback is invoked
- false if the event is not registered

Parameters

- `p`: the event dispatch, must be valid
- `event`: event to be dispatched

5.8 FIFO API Reference

Functions

bool **`osiFifoInit`** (*osiFifo_t *fifo*, void *data*, size_t *size*)
initialize FIFO

Return

- true on success
- false on invalid parameter

Parameters

- `fifo`: the FIFO pointer
- `data`: FIFO buffer
- `size`: FIFO buffer size

void **`osiFifoReset`** (*osiFifo_t *fifo*)
reset FIFO

After reset, the internal state indicates there are no data in the FIFO.

Parameters

- `fifo`: the FIFO pointer

int **`osiFifoPut`** (*osiFifo_t *fifo*, **`const`** void *data*, size_t *size*)
put data into FIFO

The returned actual put size may be less than *size*.

Return actually put size

Parameters

- `fifo`: the FIFO pointer
- `data`: data to be put into FIFO
- `size`: data size

int **`osiFifoGet`** (*osiFifo_t *fifo*, void *data*, size_t *size*)
get data from FIFO

The returned actual get size may be less than *size*.

Return actually get size

Parameters

- `fifo`: the FIFO pointer
- `data`: data buffer for get
- `size`: data buffer size

int **osiFifoPeek**(*osiFifo_t* *fifo, void *data, size_t size)
peek data from FIFO

On peek, the read position of FIFO won't be updated.

Return actually peek size

Parameters

- `fifo`: the FIFO pointer
- `data`: data buffer for peek
- `size`: data buffer size

void **osiFifoSkipBytes**(*osiFifo_t* *fifo, size_t size)
update read position to skip bytes in FIFO

When *size* is larger than byte count in FIFO, only available bytes will be skipped.

Parameters

- `fifo`: the FIFO pointer
- `size`: byte count to be skipped

bool **osiFifoSearch**(*osiFifo_t* *fifo, uint8_t byte, bool keep)
search a byte in FIFO

At search, the non-matching bytes will be dropped from the FIFO. When *byte* is found, it is configurable to keep the byte or drop the byte.

Return

- true if *byte* is found
- false if *byte* is not found

Parameters

- `fifo`: the FIFO pointer
- `byte`: the byte to be searched
- `keep`: true to keep the found byte, false to drop the found byte

static size_t **osiFifoBytes**(*osiFifo_t* *fifo)
byte count in the FIFO

Return the byte count of the FIFO

Parameters

- `fifo`: the FIFO pointer

static size_t **osiFifoSpace** (*osiFifo_t *fifo*)
available space in the FIFO

Return the available space byte count of the FIFO

Parameters

- `fifo`: the FIFO pointer

static bool **osiFifoIsFull** (*osiFifo_t *fifo*)
chech whether the FIFO is full

Return

- true if the FIFO is full
- false if the FIFO is not full

Parameters

- `fifo`: the FIFO pointer

static bool **osiFifoIsEmpty** (*osiFifo_t *fifo*)
chech whether the FIFO is empty

Return

- true if the FIFO is empty
- false if the FIFO is not empty

Parameters

- `fifo`: the FIFO pointer

struct osiFifo_t
#include <osi_fifo.h> OSI FIFO data structure.

Don't access the field members directly. Rather FIFO APIs should be used.

Public Members

void ***data**
FIFO buffer.

size_t **size**
FIFO buffer size.

size_t **rd**
FIFO read pointer.

size_t **wr**
FIFO write pointer.

5.9 Value String Map API Reference

Defines

OSI_VSMAP_CONST_DECL(name)
helper macro for constant map

For constants by *#define* or *enum*, the followings can be used:

```
osiValueStrMap_t array[] = {
    {OSI_VSMAP_CONST_DECL(SOME_MACRO)},
    {OSI_VSMAP_CONST_DECL(SOME_ENUM)},
};
```

Functions

int **osiUintIdCompare**(**const** void *key, **const** void *p)
function for comparison

In bsearch(3) and qsort(3), a comparison function is needed. When the key for comparing struct is `uint32_t` and it is the first field, this helper function can be used. For example:

```
struct {
    uint32_t id;
    // ......
};
```

Return

- -1 if key < value, though negative can conform
- 0 if key == value
- 1 if key > value, though positive can conform

Parameters

- `key`: key to be compared
- `p`: value to be compared

const char ***osiVsmapBsearch**(uint32_t *value*, **const** *osiValueStrMap_t* *vs, size_t *count*, **const** char *defval)
bsearch in value-string map

The map must be sorted ascending by `value`.

When `value` isn't found in the map, `defval` will be returned.

Return

- found mapped string
- `defval` if not found

Parameters

- `value`: value to be searched
- `vs`: value-string map array, must be valid and sorted
- `count`: value-string map count
- `defval`: default string when not found

bool **osiVsmapIsSorted**(**const** *osiValueStrMap_t* *vs, size_t *count*)

check whether value-string map is sorted

In `osiVsmapBsearch`, value-string map must be sorted ascending by `value`. This is to check whether the map is sorted.

Return

- true if the map is sorted ascending
- false if not sorted

Parameters

- `vs`: value-string map array, must be valid
- `count`: value-string map count

const char ***osiVsmapBsearchEx**(uint32_t *value*, **const** *osiValueStrMap_t* *vs, size_t *count*, size_t *size*, **const** char **defval*)

customized bsearch in value-string map

The real data structure of `vs` is not *`osiValueStrMap_t`*, but it is started with *`osiValueStrMap_t`*, with customized size.

Return

- found mapped string
- `defval` if not found

Parameters

- `value`: value to be searched
- `vs`: value-string map array, must be valid and sorted
- `count`: value-string map count
- `size`: real data structure size

- `defval`: default string when not found

bool **osiVsmapIsSortedEx**(**const** *osiValueStrMap_t* *vs, size_t *count*, size_t *size*)

check whether customized value-string map is sorted

The real data structure of `vs` is not *`osiValueStrMap_t`*, but it is started with *`osiValueStrMap_t`*, with customized size.

Return

- true if the map is sorted ascending
- false if not sorted

Parameters

- `vs`: value-string map array, must be valid
- `count`: value-string map count
- `size`: real data structure size

const *osiValueStrMap_t* ***osiVsmapFindByIStr**(**const** *osiValueStrMap_t* *vsmap, **const** char *str)

find value by string, case insensitive

The map is ended by NULL of *str*

Return

- a map item if found
- NULL if not found

Parameters

- `vsmap`: integer/string map
- `str`: string value

const *osiValueStrMap_t* ***osiVsmapFindByStr**(**const** *osiValueStrMap_t* *vsmap, **const** char *str)

find value by string, case sensitive

The map is ended by NULL of *str*

Return

- a map item if found
- NULL if not found

Parameters

- `vsmap`: integer/string map
- `str`: string value

const *osiValueStrMap_t* ***osiVsmapFindByVal**(**const** *osiValueStrMap_t* **vsmap*, uint32_t *value*)
find string by value

The map is ended by NULL of *str*

Return

- a map item if found
- NULL if not found

Parameters

- `vsmap`: integer/string map
- `value`: integer value

const char ***osiVsmapFindStr**(**const** *osiValueStrMap_t* **vsmap*, uint32_t *value*, **const** char **defval*)
find string by value, with default string

The map is ended by NULL of *str*. When `value` is not found, `defval` is returned.

Return string value

Parameters

- `vsmap`: integer/string map
- `value`: integer value
- `defval`: default string at not found

uint32_t **osiVsmalFindVal**(**const** *osiValueStrMap_t* **vsmap*, **const** char **str*, uint32_t *defval*)
find value by string, with default value

The map is ended by NULL of *str*. When `str` is not found, `defval` is returned.

Return

- a map item if found
- NULL if not found

Parameters

- `vsmap`: integer/string map
- `str`: string value
- `defval`: default integer value at not found

uint32_t **osiVsmalFindIVal**(**const** *osiValueStrMap_t* **vsmap*, **const** char **str*, uint32_t *defval*)
find value by case insensitive string, with default value

The map is ended by NULL of *str*. When `str` is not found, `defval` is returned.

Return

- a map item if found
- NULL if not found

Parameters

- `vsmap`: integer/string map
- `str`: string value
- `defval`: default integer value at not found

bool **`osiIsUintInList`**(uint32_t *value*, **`const`** uint32_t **varlist*, unsigned *count*)
little helper to check wether an unsigned integer in list

Return

- true if value in the list
- false if value not in the list

Parameters

- `value`: value to be checked
- `varlist`: value list
- `count`: value list count

bool **`osiIsUintInRange`**(uint32_t *value*, uint32_t *minval*, uint32_t *maxval*)
little helper to check wether an unsigned integer in range

Return

- true if value in the range
- false if value not in the range

Parameters

- `value`: value to be checked
- `minval`: minimal value, inclusive
- `maxval`: maximum value, inclusive

bool **`osiIsUintInRanges`**(uint32_t *value*, **`const`** *osiUintRange_t* **ranges*, unsigned *count*)
little helper to check wether an unsigned integer in range list

Return

- true if value in the range list
- false if value not in the range list

Parameters

- `value`: value to be checked

- `ranges`: valid range list
- `count`: valid range list count

bool **osiIsIntInList** (int *value*, **const** int **varlist*, unsigned *count*)
little helper to check wether a signed integer in list

Return

- true if value in the list
- false if value not in the list

Parameters

- `value`: value to be checked
- `varlist`: value list
- `count`: value list count

bool **osiIsIntInRange** (int *value*, int *minval*, int *maxval*)
little helper to check wether a signed integer in range

Return

- true if value in the range
- false if value not in the range

Parameters

- `value`: value to be checked
- `minval`: minimal value, inclusive
- `maxval`: maximum value, inclusive

bool **osiIsIntInRanges** (int *value*, **const** *osiIntRange_t* **ranges*, unsigned *count*)
little helper to check wether a signed integer in range list

Return

- true if value in the range list
- false if value not in the range list

Parameters

- `value`: value to be checked
- `ranges`: valid range list
- `count`: valid range list count

struct osiValueStrMap_t
#include <*osi_vsmap.h*> helper data structure for integer string map

Public Members

uint32_t **value**
: integer value

const char ***str**
: string value

struct osiUintRange_t
#include <osi_vsmap.h> data structure to define an unsigned integer range

Public Members

uint32_t **minval**
: minimal value, inclusive

uint32_t **maxval**
: maximum value, inclusive

struct osiIntRange_t
#include <osi_vsmap.h> data structure to define a signed integer range

Public Members

int **minval**
: minimal value, inclusive

int **maxval**
: maximum value, inclusive

5.10 Memory Recycler API Reference

Typedefs

typedef struct osiMemRecycler **osiMemRecycler_t**
opaque data structure of memory recycler

Functions

osiMemRecycler_t ***osiMemRecyclerCreate**(size_t *depth*)
creat memory recycler

To avoid dynamic memory complexity, static memory with specified `depth` will be used.

Return

- memory recycler pointer

- NULL if out of memory

Parameters

- `depth`: maximum pointer count

void **osiMemRecyclerDelete**(*osiMemRecycler_t* *d)
delete the memory recycler

When `p` is NULL, nothing will be done.

Parameters

- `d`: the memory recycler

bool **osiMemRecyclerPut**(*osiMemRecycler_t* *d, **const** void *p)
put a pointer into memory recycler

To mimic free (3), `p` can be NULL. In this case, it will be ignored silently.

When the pointer is already in the memory recycler, it can't be added again.

Return

- true on success
- false on fail
  - invalid parameter
  - the pointer already in memory recycler

Parameters

- `d`: the memory recycler, must be valid
- `p`: pointer to be put to memory recycler

bool **osiMemRecyclerUndoPut**(*osiMemRecycler_t* *d, **const** void *p)
undo put a pointer into memory recycler

Though it is rare to undo put, this is also provided.

Return

- true on success
- false on fail
  - invalid parameter
  - the pointer not in memory recycler

Parameters

- `d`: the memory recycler, must be valid
- `p`: pointer to be extract from memory recycler

void **osiMemRecyclerEmpty**(*osiMemRecycler_t* *d)
free pointers into memory recycler

After pointers are freed, they are extracted from the memory recycler also.

Parameters

- d: the memory recycler, must be valid

5.11 Generic List API Reference

Typedefs

typedef struct *osiSlistItem* **osiSlistItem_t**
data type for generic SLIST item

typedef struct *osiTailqItem* **osiTailqItem_t**
data type for generic TAILQ item

struct osiSlistItem
#include <osi_generic_list.h> data type for generic SLIST item

Public Members

osiSlistIter_t **iter**
SLIST_ENTRY iterator.

struct osiTailqItem
#include <osi_generic_list.h> data type for generic TAILQ item

Public Members

osiTailqIter_t **iter**
TAILQ_ENTRY iterator.

CHAPTER

SIX

C++ LANGUAGE

Contents

C++ language is supported in this SDK. C++ *may* introduce resource overhead, including code size and RAM consumption unintentionally. On embedded systems, the overhead may be unacceptable. However, when developer has enough knowledge for all used C++ features, it is permitted to use C++.

Even module is implemented in C++, C style public header *must* be provided. That is, C codes should always work.

6.1 Compiling Options

The following compiling options are used to compile C++ codes:

- -std=c++11
- -fno-exceptions
- -fno-rtti
- -fno-threadsafe-statics

6.2 Global Object

Example:

```
static SomeClass gSomeObject;
```

Global objects are permitted, but should be avoided if possible. When used, developer should make sure that the functions inside constructor are available.

Constructor of global objects are called in `osiAppStart`. RTOS is started, and RTOS APIs are ready.

When it is desired that constructor should be called before `osiAppStart`, *don't* use it.

6.3 Static Object

Example:

```
void function(void)
{
    static SomeClass gSomeObject;
    ......
}
```

Again, static objects are permitted and not encouraged. Contrary to global objects, constructors are not called in `osiAppStart`. Rather, it is called when the function is executed at the first time.

For simplicity, *-fno-threadsafe-statics* compiling option is added. That is, when the function will be called in multiple threads, there exists potential risk about the constructor. Thread safe of static object constructor should be considered by developers, or even better, not use this C++ feature.

CHAPTER

SEVEN

CLOCK MANAGEMENT

Contents

Clock management is a part of power management. *PM source* is for sleep and suspend management. That will handle:

- Whether system can sleep or suspend;
- Callback before and after sleep or suspend;

Clock management is system power consumption fine tune in active state. It will handle:

- Make sure system is running in lower speed, while performance is enough;
- Callback before and after system clock is changed;
- Constrain for fixed system clock;

Note: Not all platforms support dynamic system clock. At platforms not support dynamic system clock, the related APIs will be properly implemented as doing nothing. Application won't notice the differences.

7.1 clk_sys Callback

Clock of some hardware module is divided from clk_sys. In this case, the hardware module divider should be changed when clk_sys is changed. And it is needed to register a callback.

To make sure that the hardware module won't work on higher frequency then expected, system will take care the order of change clk_sys and change hardware module divider. Example of increase clk_sys:

```
uint32_t critical = osiEnterCritical();
call_all_registered_callbacks();
increase_clk_sys();
osiExitCritical(critical);
```

Example of decrease clk_sys:

```
uint32_t critical = osiEnterCritical();
decrease_clk_sys();
call_all_registered_callbacks();
osiExitCritical(critical);
```

So, the hardware module will work on *lower* frequency then expected in short period. Also, it means clk_sys change callback should be fast as possible.

7.2 Fix clk_sys Constrain

When clock of hardware module is divided from clk_sys, and the hardware module can't tolerant short period clock change in work mode, driver of hardware module can set *fix clk_sys constrain* before the hardware module is about to work, and release the constrain after the hardware module work is done.

Note: This feature isn't implemented now.

7.3 Hardware Minimal Clock Constrain

When it is known that the hardware module can't work normal when system clock is too low, the hardware module can set a hardware minimal clock constrain.

7.4 Software Minimal Clock Constrain

It is just a system performance estimation. When it is predicted that the following software processing needs high performance, software minimal clock constrain can be set.

Due to software performance requirement is estimation, and it is for the peak performance requirement, it is possible that the requested performance is not needed in some period. So, when system enter idle thread, it will indicate that the requested performance is not needed at that moment.

So, software minimal clock constrain won't prevent system power saving in idle thread. However, hardware minimal clock constrain will be satisfied even in idle thread.

When there are only software minimal clock constrain, in idle thread it is possible that:

- Decrease clk_sys to lowest supported by hardware;
- Turn off PLL;
- Sleep or suspend;

The power saving strategy is implemented in system. User shouldn't rely on certain power saving strategy. Rather user should set proper constrain to make itself work.

Note: Hardware minimal clock constrain and software minimal clock constrain can't be set at the same time. The later one will replace the previous one.

Both hardware minimal clock constrain and software minimal clock constrain are released by the same API.

7.5 Hardware External RAM Access Constrain

It is only for 8955 now. To save power, PLL may be turned off when the PLL is not needed. And PSRAM will work on 52MHz. When there are new request, PLL can't be turned on immediately if there are hardware PSRAM access, such as DMA. And there are cases that system will panic if the requested performance can't satisfied.

To handle this case, external RAM access constrain should be set before hardware PSRAM access. When system handle the constrain, PLL will be turned on beforehand.

RX IFC is a special case. Even RX IFC is active, PLL can't be turned on safely. So, it is not needed to set hardware external RAM access constrain for RX IFC from PSRAM.

It is only for hardware module PSRAM access, it is not needed to set constrain for software PSRAM access.

7.6 Hardware Clock Constrain and PM Source

When hardware minimal clock constrain is set, it will prevent system to sleep or suspend. However, when hardware module is working, it is recommended to set hardware clock constrain, rather than call `osiPmWakeLock`. Otherwise, hardware module may be affected by various power consumption optimization, even system doesn't enter sleep.

7.7 Reapply Constrains

It is a concept for 8955 only. In 8955, increase clk_sys should be satisfied immediately. And it is possible that decrease clk_sys constrains aren't satisfied immediately. So, it is needed to call `osiReapplySysClk` in idle thread to make sure clk_sys will be really decreased.

It is a system implementation concept, application doesn't need to take care of this.

7.8 Thread Safe

Constrain request and release APIs are thread safe. Also, they can be called inside ISR.

7.9 API Reference

Typedefs

typedef uint32_t (***osiSysClkChangeCallback_t**)(uint32_t sysClkFreq)
callback function for clk_sys change

The callback is platform dependent. Typical usage is for hardware module, whose clock is divided from clk_sys. So, when clk_sys is changed, it is needed to update the divider from clk_sys.

It is recommended to add callback only in hardware driver, and divider change is needed.

To ensure atomic, the callback will be called with interrupt disabled. So, the callback should be as fast as possible, even trace inside callback is discouraged.

Return hardware module working frequency

Parameters

- `sysClkFreq`: new clk_sys to be changed

Functions

void **osiClockManInit**(void)
clock management module initialization

void **osiClockManStart**(void)
clock management start

At initialization, clock management will just record clock constrains. Only after this is called, clock management will start to change system clock based on clock constrains.

void **osiRegisterSysClkChangeCallback**(*osiSysClkCallbackRegistry_t* *r, *osiSysClkChangeCallback_t* *cb*)
register clk_sys change callback

When `cb` is NULL, it is the same as `osiUnregisterSysClkChangeCallback(r)`.

Parameters

- `r`: callback registry, must be valid
- `cb`: callback function

void **osiUnregisterSysClkChangeCallback**(*osiSysClkCallbackRegistry_t* *r)
unregister clk_sys change callback

Parameters

- `r`: callback registry, must be valid

void **`osiInvokeSysClkChangeCallbacks`**(uint32_t *freq*)
invoke all registered clk_sys change callbacks

It should **only** be called in HAL, during clk_sys change. This must be called with interrupt disabled.

It shouldn't be called by application.

Parameters

- `freq`: clk_sys frequency to be changed

void **`osiRequestSysClk`**(*osiClockConstrainRegistry_t* *r, uint32_t *freq*)
set hardware minimal clock constrain

It is not needed to specify *to* be supported by hardware divider. The **closest** supported hardware divider will be chosen.

When *freq* is too high to be supported by hardware, the highest supported frequency will be chosen.

After this call, the actual clk_sys may be different with *freq:*

- When there are higher frequency requests, the actual clk_sys frequency may be higher than *freq*.
- When the closest supported frequency is lower, the actual clk_sys frequency may be lower than *freq*.
- When *freq* is too high, clk_sys will be set to be the highest supported frequency.

Parameters

- `r`: clock constrain registry, must be valid
- `freq`: minimum clk_sys frequency

void **`osiRequestSysClkActive`**(*osiClockConstrainRegistry_t* *r)
set hardware minimal clock constrain, without specified frequency

Besides it will prevent system sleep, it will prevent clk_sys related power consumption optimization.

Parameters

- `r`: clock constrain registry, must be valid

void **`osiRequestPerfClk`**(*osiClockConstrainRegistry_t* *r, uint32_t *freq*)
set software minimal clock constrain

It is similar to `osiRequestSysClk`. However, it won't prevent system sleep or clk_sys related power consumption optimization.

Parameters

- `r`: clock constrain registry, must be valid
- `freq`: minimum clk_sys frequency

void **osiReleaseClk** (*osiClockConstrainRegistry_t* *r)
unset software or hardware minimal clock constrain

Parameters

- r: clock constrain registry, must be valid

bool **osiIsSlowSysClkAllowed** (void)
whether slow down sys clock is allowed

Return

- true if allowed
- false if not allowed

void **osiRequestExtRamAccess** (*osiClockConstrainRegistry_t* *r)
set hardware external RAM access constrain

Parameters

- r: clock constrain registry, must be valid

void **osiReleaseExtRamAccess** (*osiClockConstrainRegistry_t* *r)
unset hardware external RAM access constrain

Parameters

- r: clock constrain registry, must be valid

void **osiReleaseAllConstrain** (*osiClockConstrainRegistry_t* *r)
unset hardware external RAM access constrain

Parameters

- r: clock constrain registry, must be valid

void **osiReapplySysClk** (void)
reapply clk_sys based on constrain

It is possible that some constrains aren't satisfied immediately, especially to decrease clk_sys. So, it may be needed to reapply constrains. Usually, it is called in idle thread.

It should be called by system only.

int **osiClockConstrainDump** (void *mem, int count)
dump clock constrain information to memory

It is for debug only. The data format of timer information dump is not stable, and may change. Currently, there is 2 bytes header, and 12 bytes for each clock constrain.

Caller should make sure mem is enough for hold clock constrain information of count.

Return dump memory size

Parameters

- `mem`: memory for clock constrain information dump
- `count`: maximum clock constrain count to be dump

struct osiSysClkCallbackRegistry_t
#include <*osi_clock.h*> clk_sys change callback registry

Public Members

uint32_t **tag**
name tag

struct osiClockConstrainRegistry_t
#include <*osi_clock.h*> clk_sys constraint registry

Public Members

uint32_t **tag**
name tag

CHAPTER

EIGHT

MEMORY MANAGEMENT

Contents

8.1 Overview

This SDK memory management is based on multiple pools. Use cases:

- create separated pools on internal SRAM and external RAM;
- create pool from one big memory block allocated from system pool;

Note: The memory of each heap should be one contiguous memory.

For application, the recommended practice is to use dynamic memory APIs in `stdlib.h`:

```
* malloc
* calloc
* realloc
* memalign
* free
```

The implementation follows posix standard.

8.2 Pool Type

There are 2 kinds of pools:

- fixed block pool: The memory is split into blocks with fixed blocks. It is not flexible. However, there are no fragmentation issue. Also, the allocation and free is very fast.
- dynamic block pool: Support arbitrate size of block. It is flexible. However, it may suffer by fragmentation issue, and it is slower than fixed block pool.

Note: Again, in most cases, standard dynamic memory APIs are enough.

Note: Fixed block pool is experimental, don't use it in real application.

8.3 Reference Count

A simple reference count mechanism is implemented. In the block header of each block memory, there is a reference count. At malloc, the initial count is 1. `osiMemRef` can be used to increase the reference count, `free` will decrease the reference count. Only when reference count is decreased to 0, the memory block will be *really* released.

To save space of memory block header, only several bits are used to record reference count. And when the reference count exceeds the maximum permitted value, system will panic. So, when it is known that the reference count will be increased to a larger value, **don't** use the built-in reference count mechanism.

8.4 Pattern Check

At allocation, at lease one more byte will be requested. At the end of each memory block, a magic byte `0xFD` will be written at allocation. At free, the magic byte will be checked, and system will panic when the magic byte is changed. It is used to detect memory overflow.

8.5 Alignment

The default alignment of dynamic memory is **8 bytes**. So, the allocated memory is suitable for `double` and `long long`.

`memalign` can allocate memory with specified alignment. The typical case is to allocate D-cache line aligned memory.

8.6 Thread Safe and ISR

Memory management APIs are thread safe. Also, all of them can be called inside ISR.

8.7 Limitation

To save space of block memory header, the are some limitations:

- The maximum pool size is 32MB.
- The maximum pool count is 16.
- The maximum reference count is 63.

8.8 API Reference

Typedefs

typedef struct osiMemPool **osiMemPool_t**
opaque data structure for memory pool

Functions

osiMemPool_t ***osiFixedPoolInit**(void **ptr*, size_t *size*, size_t *block_size*)
initialize a fixed pool

Initialize and register a fixed size block pool. The pool management data structure will be located inside the provided memory, Due to alignment it may exist an offset.

Note Due to there are management overhead, the available block count is not `pool_size/block_size`.

Return

- the pool pointer on success
- NULL on failure

Parameters

- `ptr`: starting pointer of the memory
- `size`: memory total size
- `block_size`: block size

osiMemPool_t ***osiBlockPoolInit** (void **ptr*, size_t *size*, ...)
initialize a block pool

Initialize and register a block pool pool. The variadic parameters should be ended with a zero. Each pair of (size_t count, size_t size) indicates to create a fixed pool child for the block pool. When allocating from the block pool, fixed pool children will be checked automatically. It may be helpful to reduce memory fragmentation.

The management data structure of block pool and all children will be located inside the provided memory.

Note The feature of fixed block pool inside is experimental.

Return

- the pool pointer on success
- NULL on failure

Parameters

- `ptr`: starting pointer of the memory pool.
- `size`: memory pool size

void **osiPoolSetDefault** (*osiMemPool_t* **pool*)
set the default pool

Default pool will be used by malloc/calloc, and it must be block pool. The first created block pool will be set to default automatically. And this API will change default pool.

When *pool* is NULL or invalid, it will return silently.

Parameters

- `pool`: pool to be set as default.

void ***osiPoolMalloc** (*osiMemPool_t* **pool*, size_t *size*)
allocate from specified pool

When *pool* is NULL or invalid, or *size* is zero, NULL will be returned.

Refer to malloc(3).

Return

- allocated memory pointer on success
- NULL at failure

Parameters

- `pool`: the pool
- `size`: size to be allocated

void *__osiPoolMallocUnlikelyFree__ (*osiMemPool_t* *pool, size_t *size*)
allocate from specified pool, which is unlikely to be freed

Comparing to *osiPoolMalloc*, allocator will try to allocate from location with less fragmentation impact.

Usually, it only be called at system initialization. It is a replacement of global variable (either DATA or BSS).

Return

- allocated memory pointer on success
- NULL at failure

Parameters

- `pool`: the pool
- `size`: size to be allocated

void *__osiPoolCalloc__ (*osiMemPool_t* *pool, size_t *nmemb*, size_t *size*)
allocate from specified pool, and clear to zero

It is the exact same as `osiPoolMalloc(pool, nmemb*size)` and clear memory to zero.

Refer to calloc(3).

Return

- allocated memory pointer on success
- NULL at failure

Parameters

- `pool`: the pool.
- `nmemb`: member count to be allocated.
- `size`: size of each member.

void *__osiPoolRealloc__ (*osiMemPool_t* *pool, void *ptr, size_t *size*)
change size of memory block

Refer to realloc(3).

Return

- allocated memory pointer on success
- NULL at failure

Parameters

- `pool`: the pool.
- `ptr`: pointer to be changed.
- `size`: changed size.

void *`osiPoolMemalign` (*osiMemPool_t* *pool, size_t *alignment*, size_t *size*)
allocate from specified pool with specified alignment

The pool must be block pool.

alignment should be power of 2. When *alignment* is less than default alignment, it will behavior the same as `osiPoolMalloc`.

Refer to memalign(3).

Return

- allocated memory pointer on success
- NULL at failure

Parameters

- `pool`: The pool
- `alignment`: Requested alignment
- `size`: Size to be allocated

void `osiMemSetCaller` (void *ptr, void *caller)
set memory block caller address

Caller address is for debug only. And won't affect behavior. All APIs in this module will set caller automatically. Only for memory management wrappers, and it is wanted to set caller to the caller of wrapper, this function may be called.

Parameters

- `ptr`: pointer of memory block.
- `caller`: caller address

int `osiMemAllocSize` (void *ptr)
get the allocated size for the pointer

The size is allocated by memory management. The size may be larger than the size requested, due to tail padding.

Return

- allocated memory block size
- 0 if `ptr` is NULL
- -1 if `ptr` is invalid

Parameters

- `ptr`: pointer of memory block

void **osiMemRef**(void *ptr)
increase reference count of the pointer

When the reference count reach the maximum allowed reference count, system will panic.

When `ptr` is NULL, nothing will be done.

Parameters

- `ptr`: pointer of memory block

size_t **osiMemRefCount**(void *ptr)
get reference count of the pointer

When `ptr` is NULL, 0 will be returned.

Return reference count of the pointer

Parameters

- `ptr`: pointer of memory block

bool **osiMemUnrefNotLast**(void *ptr)
decrease reference if the reference count is not 1

At object oriented design with refrence count, it is not enough to consider memory reference count only. Rather, it is needed to consider the object reference count. So, the *delete* function of object should be:

```
void objectDelete(void *object)
{
    if (osiMemUnrefNotLast(object))
        return;

    objectCleanup(object);
    free(object);
}
```

When the reference count of the object, which is stored inside memory management, is not 1, only decrease the reference count. Only when it is the last reference, the cleanup shall be called.

When `ptr` is NULL, it will return true, and do nothing. Conceptually, NULL pointer can be regarded as reference count of 0. So, it is not the last reference.

When reference count of `ptr` is greater than 1, the reference count will be decreased by 1, and it is the same as `free`.

Return

- true if it is not the last reference, or `ptr` is NULL
- false if it is the last reference

Parameters

- `ptr`: pointer of memory block

void *__osiMalloc__(size_t *size*)
refer to malloc(3).

void *__osiCalloc__(size_t *nmemb*, size_t *size*)
refer to calloc(3).

void *__osiRealloc__(void **ptr*, size_t *size*)
refer to realloc(3).

void *__osiMemalign__(size_t *alignment*, size_t *size*)
refer to memalign(3).

void __osiFree__(void **ptr*)
refer to free(3).

bool __osiMemPoolStat__(*osiMemPool_t* **pool*, *osiMemPoolStat_t* **stat*)
get memory pool information

max_block_size is the maximum allocatable size. *realloc* and *memalign* will use more extra spaces, they may fail with that size.

```
malloc(stat->max_block_size);         // will success
realloc(ptr, stat->max_block_size);   // may fail
memalign(32, stat->max_block_size);   // may fail
```

Return

- true on success
- false if there are no memory pool, or *stat* is NULL

Parameters

- `pool`: the memory pool. *NULL* for default memory pool
- `stat`: output memory pool information

struct osiMemPoolStat_t
#include <osi_mem.h> memory pool information

Public Members

void *__start__
memory pool start pointer

uint32_t __size__
memory pool total size

uint32_t __index__
memory pool internal index

uint32_t __avail_size__
available size. The actual allocatable size may be less than this

uint32_t **max_block_size**
maximum allocatable block size

CHAPTER

NINE

TRACE

Contents

9.1 Overview

Trace is embedded system implementation of `printf`.

All trace are formated on PC by PC trace tool.

- `printf` is more complex then thought, especially floating point support. It can save CPU cycles to format on PC trace tool. This SDK should be able to run on system with limited performance.
- If needed, `vsprintf` on platform can be a trimmed version, for example, not support floating point. Even for that case, trace won't be affected.
- In most cases, less bytes are needed by format on PC trace tool.
- When trace ID is used, it is necessary to format on PC trace tool.

9.2 Format String

Nearly all printf(3) formatting are supported, except:

- `*, $1, $2` are not supported
- Flags `I` is not supported.
- Length `L` for long double is not supported.
- Conversion `C, S, n, m` is not supported.
- `%*s` will be interpreted as memory dump. The parameter is `size, pointer`. Parameters are compatible with `printf`, just PC trace tool will be formated it differently, such as: `(0x82000000/6) 00 01 02 03 04 05`.
- `%4c` will be interpreted as 4 characters. For example, `('A')|('B'<<8)|('C'<<16)|('D'<<24)` will be shown as `ABCD`. The standard `printf` will show ␣␣␣A.

Due to traces are interpreted in PC trace tool, the syntax is not limited by `vsnprintf` function running on target.

9.3 Trace Level

Five trace levels are defined:

- ERROR
- WARN
- INFO
- DEBUG
- VERBOSE

Trace level can be used as static compile-time control. At debugging, more detailed traces are preferred, and at release, less traces are preferred. Trace level will make this change much easier. Also, the trace level information is output to PC trace tool also. PC trace tool can selected show or hide certain trace by trace level. It can make trace more readable.

Macros for trace level:

- OSI_LOG_DISABLED: When defined, trace are disabled. This macro shouldn't defined globally. Usually, it can be defined for one file or one module.
- OSI_LOCAL_LOG_LEVEL: Static compile-time control. When importance of the trace statement is less than this macro, the trace statement will be expanded to empty (no instructions will be generated). Usually, it is defined for one file. The default trace level is OSI_LOG_LEVEL_INFO.

9.4 Trace Tag

Trace tag is to indicate which module each trace belongs to. Ideally, string is the best choice for this information. However, deep embedded system is hard to afford (code size and trace interface bandwidth) a string for each trace. As a compromise, a tag of 4 characters are used for this information.

Macros for trace tag:

- OSI_LOCAL_LOG_TAG: Usually, it is defined for one file.

- OSI_LOG_TAG: Only when OSI_LOCAL_LOG_TAG is not defined, OSI_LOG_TAG will be used. Usually, it is defined for one module.

An example:

```
#define OSI_LOCAL_LOG_TAG OSI_MAKE_LOG_TAG('K', 'E', 'R', 'N')
#define OSI_LOCAL_LOG_LEVEL OSI_LOG_LEVEL_DEBUG
#include "osi_log.h"
```

Note: OSI_MAKE_LOG_TAG is different from OSI_MAKE_TAG. The former uses 7 bits for each character, and the later uses 8 bits for each character.

9.5 Trace ID

To save code size and trace interface further, the format string in each trace can be replaced by a 32 bits ID, which is called trace ID. When trace ID is used, PC trace tool should be able to synchronize the trace ID and format string map.

When CONFIG_KERNEL_DISABLE_TRACEID is defined, trace ID won't be used and the format string will be output.

9.6 Basic and Extended

Traditionally, the type of `printf` variadic arguments are determined by format string. When format on PC trace tool, especially trace ID is used, trace APIs won't parse the format string. In this case, additional information is needed for variadic arguments.

The followings arguments aren't in form of 32 bits integer, and called extended argument:

- `%s`: string output
- `%e/E/f/F/g/G/a/A`: double floating point
- `%ll`: 64 bits integer
- `%*s`: self defined memory dump

Others are called basic argument. When all arguments are basic argument, `OSI_LOG` can be used. Otherwise when at least one argument is extended, `OSI_LOGX` shall be used. `OSI_LOGX` needed to explicitly specify the format of each argument.

For `OSI_LOGX`, argument type is expressed by 4 bits for each. I for basic, S (string), F (double floating point), D (64 bits integer) and M (memory dump) for extended.

For argument count not more than 4, there are macros for all argument type combinations, such as `OSI_LOGPAR_IDSF`. When argument count more than 4, `OSI_LOGPAR()` macro can be used, such as `OSI_LOGPAR(I, D, S, F, M)`.

As implementation limitation, at most 16 arguments are supported in `OSI_LOG` and 8 arguments are supported for `OSI_LOGX`.

`OSI_PRINTF` macro will check variadic argument type by format string. It can support both basic and extended arguments. However, it is slower than `OSI_LOG` and `OSI_LOGX`.

9.7 API Reference

Defines

OSI_MAKE_LOG_TAG(a, b, c, d)
macro for trace tag

OSI_LOGPAR(...)
macro for extended trace argument types

OSI_LOGE_EN
macros for trace level condition

```
if (OSI_LOGD_EN) {
    ......
}
```

When DEBUG trace is not enabled, the above codes will be expanded as empty.

OSI_LOGW_EN

OSI_LOGI_EN

OSI_LOGD_EN

OSI_LOGV_EN

OSI_LOGE(trcid, fmt, ...)
macros for basic trace

When the trace level is not enabled, the macro will be expanded as empty.

At most 16 arguments are supported.

Parameters

- `trcid`: trace ID, 0 for not use trace ID
- `fmt`: format string, only used when trace ID is 0

OSI_LOGW(trcid, fmt, ...)

OSI_LOGI(trcid, fmt, ...)

OSI_LOGD(trcid, fmt, ...)

OSI_LOGV(trcid, fmt, ...)

OSI_LOGXE (partype, trcid, fmt, ...)
macros for extended trace

When the trace level is not enabled, the macro will be expanded as empty.

At most 16 arguments are supported.

Parameters

- `partype`: arguments types
- `trcid`: trace ID, 0 for not use trace ID
- `fmt`: format string, only used when trace ID is 0

OSI_LOGXW (partype, trcid, fmt, ...)

OSI_LOGXI (partype, trcid, fmt, ...)

OSI_LOGXD (partype, trcid, fmt, ...)

OSI_LOGXV (partype, trcid, fmt, ...)

OSI_PRINTFE (fmt, ...)
macros for trace with format string parsing

When the trace level is not enabled, the macro will be expanded as empty.

At most 16 arguments are supported.

Parameters

- `fmt`: format string, only used when trace ID is 0

OSI_PRINTFW (fmt, ...)

OSI_PRINTFI (fmt, ...)

OSI_PRINTFD (fmt, ...)

OSI_PRINTFV (fmt, ...)

OSI_SXPRINTF (id, fmt, ...)
macros for SX style trace and dump

`id` is a complex bit fields. The definition follows SX definition.

OSI_SXDUMP (id, fmt, data, size)

OSI_SX_TRACE (id, trcid, fmt, ...)
macros for new SX style trace

Only module level in `id` will be used. When SX style trace is wanted to be kept, it is suggested to migrate to these 2 macros.

OSI_SX_TRACEX (id, partype, trcid, fmt, ...)

OSI_PUB_TRACE (module, category, trcid, fmt, ...)
macros for stack trace of pub modules

OSI_PUB_TRACEX(module, category, partype, trcid, fmt, ...)

OSI_LTE_TRACE(module, category, trcid, fmt, ...)
macros for stack trace of lte modules

OSI_LTE_TRACEX(module, category, partype, trcid, fmt, ...)

OSI_TRACE(trcid, fmt, ...)
macros for stack trace without module and category control

It is suggested to use these 2 for quick debug only. The above macros with module and category control should be used.

OSI_TRACEX(partype, trcid, fmt, ...)

Enums

enum [anonymous]
trace level, larger value is less important

Values:

OSI_LOG_LEVEL_NEVER
only used in control, for not to output trace

OSI_LOG_LEVEL_ERROR
error

OSI_LOG_LEVEL_WARN
warning

OSI_LOG_LEVEL_INFO
information

OSI_LOG_LEVEL_DEBUG
for debug

OSI_LOG_LEVEL_VERBOSE
verbose

Functions

void **osiTraceVprintf**(unsigned *tag*, **const** char **fmt*, va_list *ap*)
trace vprintf by parsing format string

This can be used to implement trace functions in third party library. When there are chances to modify source codes, the appropriate macros from above should be used.

Parameters

- `tag`: packed trace tag and trace level
- `fmt`: format string
- `ap`: variadic argument list

CHAPTER

TEN

SFFS (SMALL FLASH FILE SYSTEM)

Contents

10.1 Overview

SFFS is designed for small NOR flash. Features:

- power failure safe
- wear-leveling
- high utilization
- predictable garbage collection

- high performance on one file write
- quick format

Note: When there are too many files are opened for write, the write performance will be downgraded.

SFFS is designed into 2 layers:

- flash block device
- SFFS core

10.2 Flash Block Device

Flash block device layer will take care of:

- logical block to physical block mapping
- power failure safe
- wear-leveling
- garbage collection

erase block (EB) The block size of flash erase. It should be multiple of flash sector size (usually 4KB). For performance, it is recommended to consider block size supported by flash directly, such as 32KB and 64KB.

logical block (LB) Typical logical block size is 512B. For extremely small file system, 256B can be considered. Don't use other size unless you know what you are doing.

physical block (PB) There is 1:1 mapping between logical block and physical block. Just the index is different.

There are 2 versions of flash block device. Version 2 is more reliable at power failure. Version 1 will be phased out, except existed projects.

10.2.1 Version 1

So, the logical block count of flash block device is:

```
(EB_COUNT - 1) * (EB_SIZE/PB_SIZE - 1) - 1
```

Examples:

Flash_size	EB_size	PB_size	Usable	Utilization
3MB	64K	512B	5968 x512	97.1%
1MB	32K	512B	1952 x512	95.3%
128KB	4K	256B	464 x512	90.6%

10.2.2 Version 2

So, the logical block count of flash block device is:

```
(EB_COUNT - 1) * (EB_SIZE/PB_SIZE) - 1
```

And LB size is 12 bytes less than PB size. Examples:

Flash_size	EB_size	PB_size	LB_count	Utilization
3MB	64K	512B	6015 x500	95.6%
1MB	4K	512B	2039 x500	97.2%
128KB	4K	256B	495 x244	92.1%

10.3 SFFS Blocks

There is one root block in each SFFS file system. Each directory can only use one block. There are 3 kind of files:

- tiny file: file data is stored in the spare space of file meta-block;
- small file: file data block index are stored in the spare space of file meta-block;
- large file: second level file data block index are used;

The maximum file name size (not including directory) is 64 bytes, including the ending 0 terminate char.

The maximum one level directory name size is 64 bytes, including the ending 0 terminate char.

PB_size	512B (V1)	256B (V1)	512B (V2)	256 (V2)
root children count	240	112	234	106
directory children count	208	80	202	74
tiny file max	416B	160B	404B	148B
small file max	104KB	20KB	98.6KB	17.6KB
large file max	26MB	2.5MB	23.9MB	2.1MB

10.4 Power Failure Safe

Power failure safe is carefully designed in flash block device and SFFS. There is only one exception:

> When a file is opened for write, and power failure occurs before close or sync, the written data may be lost.

10.5 `vfs_sfile_write`

`vfs_sfile_write` is the API for power failure safe write. Inside, the original contents will only be deleted from flash after the new contents are written. So, even there is power failure at any time during write, the file can be rollback to original contents. There are no contents lost.

Extra space on file system are needed for `vfs_sfile_write`. So, when file system is full, `vfs_sfile_write` will fail even the size of the new contents is the same with the size of original contents.

10.6 Quick Format

For invalid file system, the time to format the file system is negligible. SFFS won't erase all blocks for format. Rather the flash blocks will be erased progressively at garbage collection.

So, at production, even the file system reign is not filled with preset empty file system data, the first boot time won't be significant increased.

At development, after the flash layout is changed, the file system format time is insignificant also.

10.7 Memory Usage

At flash block device create, the following memory are allocated:

- memory for control information
- memory for one PB
- memory for PB/LB mapping

Only at error recovery during flash block device creation, dynamic memory for one EB will be allocated, and will be freed after error recovery.

At SFFS mount, besides control information, specified block cache count will be allocated. Each block cache is slightly larger than LB size. The minimum and recommended block cache count is 4.

At run-time, open file and directory will dynamic allocate memory. The dynamic memory is just for control information, and without buffer.

10.8 EBUSY

When a file or directory is opened, the followings will fail:

- unlink
- rmdir
- rename

10.9 Flash Block Device API Reference

Functions

blockDevice_t *__flashBlockDeviceCreate__(__struct__ drvSpiFlash *_flash_, uint32_t _phys_start_, uint32_t _phys_size_, uint32_t _phys_erase_size_, uint32_t _block_size_, bool _read_only_)

create a flash block device

There are cases fail to create flash block device:

- Failed to allocate memory for flash block device;
- Invalid parameters for flash block device;
- There are more than one EB doesn't match the configuration. It occurs when the parameters are not the same as the parameters at create. It should only happen at development.

Except that, other errors will be handled silently. So, at initialization, the return value should be checked, and call *FLASH_BLOCKDEVICE_FORMAT* and then call *FLASH_BLOCKDEVICE_CREATE* again. If it is failed again, usually *osiPanic* can be called.

Return

- NULL if failed to create flash block device
- instance pointer

Parameters

- `flash`: the flash driver instance
- `phys_start`: start offset in flash
- `phys_size`: the region size
- `phys_erase_size`: erase block size
- `block_size`: block size of block device
- `read_only`: at read only, no flash erase and program

bool __flashBlockDeviceFormat__(__struct__ drvSpiFlash *_flash_, uint32_t _phys_start_, uint32_t _phys_size_, uint32_t _phys_erase_size_, uint32_t _block_size_)

format flash region block device

Return

- false if failed to format region for flash block device. Only when the configuration is invalid, it will fail.
- true if the region is formatted

Parameters

- `flash`: the flash driver instance

- `phys_start`: start offset in flash
- `phys_size`: the region size
- `phys_erase_size`: erase block size
- `block_size`: block size of block device

blockDevice_t *__flashBlockDeviceCreateV2__(__struct__ drvSpiFlash *_flash_, uint32_t _phys_start_, uint32_t _phys_size_, uint32_t _phys_erase_size_, uint32_t _block_size_, bool _read_only_)

create a flash block device version 2

There are cases fail to create flash block device:

- Failed to allocate memory for flash block device;
- Invalid parameters for flash block device;
- There are more than one EB doesn't match the configuration. It occurs when the parameters are not the same as the parameters at create. It should only happen at development.

Except that, other errors will be handled silently. So, at initialization, the return value should be checked, and call *FLASH_BLOCKDEVICE_FORMAT* and then call *FLASH_BLOCKDEVICE_CREATE* again. If it is failed again, usually *osiPanic* can be called.

`block_size` is PB of flash block device. In version 2, LB size is not the same with PB size. So, always use blockDevice_t.block_size as LB size.

Return

- NULL if failed to create flash block device
- instance pointer

Parameters

- `flash`: the flash driver instance
- `phys_start`: start offset in flash
- `phys_size`: the region size
- `phys_erase_size`: erase block size
- `block_size`: block size of block device
- `read_only`: at read only, no flash erase and program

blockDevice_t *__flashBlockDeviceQuickCreateV2__(__struct__ drvSpiFlash *_flash_, uint32_t _phys_start_, uint32_t _phys_size_, uint32_t _phys_erase_size_, uint32_t _block_size_, bool _read_only_)

quick create a flash block device version 2

Comparing to `flashBlockDeviceCreateV2`, it is assumed that the flash block device is *clean*, that is, there are no interrupted erase and write. So, it can be faster.

Return

- NULL if failed to create flash block device
- instance pointer

Parameters

- `flash`: the flash driver instance
- `phys_start`: start offset in flash
- `phys_size`: the region size
- `phys_erase_size`: erase block size
- `block_size`: block size of block device
- `read_only`: at read only, no flash erase and program

bool **flashBlockDeviceFormatV2**(**struct** drvSpiFlash **flash*, uint32_t *phys_start*, uint32_t *phys_size*, uint32_t *phys_erase_size*, uint32_t *block_size*)

format flash block device version 2

Return

- false if failed to format region for flash block device. Only when the configuration is invalid, it will fail.
- true if the region is formatted

Parameters

- `flash`: the flash driver instance
- `phys_start`: start offset in flash
- `phys_size`: the region size
- `phys_erase_size`: erase block size
- `block_size`: block size of block device

10.10 SFFS VFS API Reference

Functions

int **sffsVfsMount**(**const** char **base_path*, **struct** blockDevice **bdev*, size_t *cache_count*, size_t *sfile_reserved_lb*, bool *read_only*)

mount SFFS to VFS

Return

- 0 on success
- -1 on fail

Parameters

- `base_path`: mount point in VFS
- `bdev`: block device pointer
- `cache_count`: block cache count in SFFS
- `sfile_reserved_lb`: reserved logical block for sfile
- `read_only`: whether to mount as read-only

int **sffsVfsMkfs**(**struct** blockDevice *_bdev_)
format SFFS on block device

Return

- 0 on success
- -1 on fail

Parameters

- `bdev`: block device pointer

10.11 SFFS API Reference

Typedefs

typedef struct sffsFs **sffsFs_t**
opaque data structure of SFFS

Functions

sffsFs_t ***sffsMount**(**struct** blockDevice *_pdev_, size_t _block_cache_count_, bool _read_only_)
create a SFFS instance

Return

- SFFS pointer on success
- NULL on failure, typical failure is root block mismatch

Parameters

- `pdev`: pointer of block device
- `block_cache_count`: block cache count
- `read_only`: whether to mount as read-only

int **sffsUnmount**(*sffsFs_t* *_fs_)
delete the SFFS instance

Return

- 0 on success

Parameters

- `fs`: SFFS pointer

int **sffsRemount**(*sffsFs_t* *fs, unsigned *flags*)
remount SFFS with flags

The only supported flag is *MS_RDONLY*.

Return

- 0 for success

Parameters

- `fs`: SFFS pointer
- `flags`: remount flags

int **sffsMakeFs**(**struct** blockDevice *pdev)
format SFFS on block device

Return

- 0 for success
- -EINVAL: invalid parameter
- -ENOMEM: out of memory

Parameters

- `pdev`: block device pointer

int **sffsOpen**(*sffsFs_t* *fs, **const** char *path, int *mode*)
refer to open(2)

It is permitted to open one file multiple times.

It is not supported to open an existed directory.

Return

- file descriptor on success
- -ENOENT: not exist, and without O_CREAT
- -EROFS: create on read-only file system
- -ENOTDIR: parent directory not exist at create
- -ENOSPC: out of space
- -ENAMETOOLONG: name is too long

- -ENOMEM: out of memory

Parameters

- `fs`: SFFS pointer
- `path`: path in file system
- `mode`: open mode (O_CREAT, O_ACCMODE, O_TRUNC)

int **sffsClose**(*sffsFs_t* *fs, int *fd*)
refer to close(2)

Return

- 0 on success
- -EINVAL: invalid SFFS pointer
- -EBADF: invalid file descriptor

Parameters

- `fs`: SFFS pointer
- `fd`: file descriptor

ssize_t **sffsWrite**(*sffsFs_t* *fs, int *fd*, **const** void *data, size_t *size*)
refer to write(2)

On failure, the return value is the negative errno. Written bytes won't be returned.

Return

- written size on success
- -EINVAL: invalid SFFS pointer
- -EBADF: invalid file descriptor
- -EROFS: write on read-only file system
- -ENOSPC: out of space
- -EOVERFLOW: exceed maximum large file size

Parameters

- `fs`: SFFS pointer
- `fd`: file descriptor
- `data`: data pointer to be written
- `size`: data size to be written

ssize_t **sffsRead**(*sffsFs_t* *fs, int *fd*, void *data, size_t *size*)
refer to read(2)

When there are no enough size in the file, the read size is returned. On other failures, the return value is the negative errno. Read bytes won't be returned.

Return

- read size on success
- -EINVAL: invalid SFFS pointer
- -EBADF: invalid file descriptor

Parameters

- `fs`: SFFS pointer
- `fd`: file descriptor
- `data`: data pointer to be read
- `size`: data size to be read

off_t **`sffsSeek`** (*sffsFs_t* *fs, int *fd*, off_t *offset*, int *mode*)
refer to lseek(2)

When the seek position is larger than file size, the position will be set to end of file silently.

Return

- 0 on success
- -EINVAL: invalid SFFS pointer, or invalid seek mode
- -EBADF: invalid file descriptor

Parameters

- `fs`: SFFS pointer
- `fd`: file descriptor
- `offset`: seek offset
- `mode`: seek mode (SEEK_SET, SEEK_CUR, SEEK_END)

int **`sffsFtruncate`** (*sffsFs_t* *fs, int *fd*, off_t *length*)
refer to ftruncate(2)

Support both increase and decrease file size. At increase file size, the extended part is undefined.

Return

- 0 on success
- -EINVAL: invalid SFFS pointer
- -EBADF: invalid file descriptor
- -EROFS: write on read-only file system
- -EACCES: file descriptor is opened as read-only

- -ENOSPC: out of space
- -EOVERFLOW: exceed maximum large file size

Parameters

- `fs`: SFFS pointer
- `fd`: file descriptor
- `length`: truncate new length

int **sffsFstat**(*sffsFs_t* *fs, int fd, **struct** stat *st)
refer to fstat(2)

Return

- 0 on success
- -EINVAL: invalid SFFS pointer, or invalid output pointer
- -EBADF: invalid file descriptor

Parameters

- `fs`: SFFS pointer
- `fd`: file descriptor
- `st`: output stat pointer

int **sffsUnlink**(*sffsFs_t* *fs, **const** char *name)
refer to unlink(2)

Return

- 0 on success
- -EINVAL: invalid SFFS pointer
- -EROFS: write on read-only file system
- -ENOENT: file not exist
- -EBUSY: file is opened

Parameters

- `fs`: SFFS pointer
- `name`: file path in file system

int **sffsRename**(*sffsFs_t* *fs, **const** char *src, **const** char *dst)
refer to rename(2)

Support 3 kinds of rename:

- rename file to an existed directory, but destination file not exist
- rename file to an existed file

- rename directory to an existed empty directory

Return

- 0 on success
- -EINVAL: invalid SFFS pointer
- -EROFS: read-only file system
- -ENOENT: source not exist
- -EBUSY: source file or directory is opened
- -ENOTDIR:
  - source is file, and destination in non-exist directory
  - source is directory, and destination is not existed directory
- -EISDIR: source is file, and destination is existed directory
- -ENOTEMPTY: source is directory, and destination is not empty
- -ENAMETOOLONG: destination name is too long

Parameters

- `fs`: SFFS pointer
- `src`: source file or directory path in file system
- `dst`: destination file or directory path in file system

int **sffsStatVfs** (*sffsFs_t *fs*, **struct** statvfs *buf*)
refer to statvfs(3)

Return

- 0 on success
- -EINVAL: invalid SFFS pointer, or output pointer

Parameters

- `fs`: SFFS pointer
- `buf`: output pointer

int **sffsSync** (*sffsFs_t *fs*)
refer to sync(2)

Return

- 0 on success
- -EINVAL: invalid SFFS pointer

Parameters

- `fs`: SFFS pointer

int **sffsMkDir** (*sffsFs_t *fs*, **const** char **name*)
refer to mkdir(2)

Return

- 0 on success
- -EINVAL: invalid SFFS pointer
- -EROFS: read-only file system
- -EEXIST: the path is existed
- -ENOTDIR: parent of the path not exist
- -ENOSPC: out of space
- -ENAMETOOLONG: destination name is too long

Parameters

- `fs`: SFFS pointer
- `name`: directory path in file system

int **sffsRmDir** (*sffsFs_t *fs*, **const** char **name*)
refer to rmdir(2)

Return

- 0 on success
- -EINVAL: invalid SFFS pointer
- -EROFS: read-only file system
- -EACCES: can't remove file system root directory
- -ENOENT: path not exists
- -ENOTDIR: parent of path not exists
- -EBUSY: path is opened
- -ENOTEMPTY: directory is not empty

Parameters

- `fs`: SFFS pointer
- `name`: directory path in file system

DIR ***sffsOpenDir** (*sffsFs_t *fs*, **const** char **name*)
refer to opendir(3)

Return

- DIR pointer on success
- NULL on fail

Parameters

- `fs`: SFFS pointer
- `name`: directory path in file system

int **sffsCloseDir** (*sffsFs_t* *fs, DIR *dir)
refer to closedir(3)

Return

- 0 on success
- -EINVAL: invalid SFFS pointer, or DIR pointer

Parameters

- `fs`: SFFS pointer
- `dir`: DIR pointer to be closed

int **sffsReadDirR** (*sffsFs_t* *fs, DIR *dir, **struct** dirent *entry, **struct** dirent **result)
refer to readdir_r(3)

At the end of directory, the return value is 0, and *result will be NULL.

Return

- 0 on success
- -EINVAL: invalid SFFS pointer, or DIR pointer, or entry pointer
- -EBADF: invalid DIR

Parameters

- `fs`: SFFS pointer
- `dir`: DIR pointer to be read
- `entry`: output dirent pointer
- `result`: output pointer of dirent pointer

struct dirent ***sffsReadDir** (*sffsFs_t* *fs, DIR *dir)
refer to readdir(3)

Return

- dirent pointer on success
- NULL on fail, or end of directory

Parameters

- `fs`: SFFS pointer
- `dir`: DIR pointer to be read

long **`sffsTellDir`** (*sffsFs_t* *fs, DIR *dir)
refer to telldir(3)

Return

- current location on success
- -EINVAL: invalid SFFS pointer, or DIR pointer

Parameters

- `fs`: SFFS pointer
- `dir`: DIR pointer to be read

void **`sffsSeekDir`** (*sffsFs_t* *fs, DIR *dir, long *loc*)
refer to seekdir(3)

Parameters

- `fs`: SFFS pointer
- `dir`: DIR pointer to be read
- `loc`: seek location

ssize_t **`sffsSfileWrite`** (*sffsFs_t* *fs, **const** char *path, **const** void *data, size_t *size*)
write all content to a file in safe mode

Comparing to `sffsFileWrite`, when the file already exists, and there are power failure during write, either the file content to the new data, or the content is kept as original.

Due to the original file won't be deleted before the new data are written, there should exist enough free blocks for the whole new data.

Return

- written size on success, it is the same as `size` parameter
- -EINVAL: invalid SFFS pointer
- -EROFS: create on read-only file system
- -ENOTDIR: parent directory not exist at create
- -EISDIR: path exists, but is directory
- -ENOSPC: out of space
- -ENAMETOOLONG: name is too long
- -EOVERFLOW: exceed maximum large file size

Parameters

- `fs`: SFFS pointer
- `path`: path in file system
- `data`: data pointer to be written
- `size`: data size to be written

ssize_t **sffsFileWrite**(*sffsFs_t* *fs, **const** char *path, **const** void *data, size_t size)
write all content to a file

Return

- written size on success, it is the same as `size` parameter
- -EINVAL: invalid SFFS pointer
- -EROFS: create on read-only file system
- -ENOTDIR: parent directory not exist at create
- -EISDIR: path exists, but is directory
- -ENOSPC: out of space
- -ENAMETOOLONG: name is too long
- -EOVERFLOW: exceed maximum large file size

Parameters

- `fs`: SFFS pointer
- `path`: path in file system
- `data`: data pointer to be written
- `size`: data size to be written

int **sffsFileBlockNeeded**(*sffsFs_t* *fs, size_t size)
estimate block count needed for specified file size

Return

- needed block count
- -EOVERFLOW: exceed maximum large file size

Parameters

- `fs`: SFFS pointer
- `size`: file size to be estimated

int **sffsBlockWriteCount**(*sffsFs_t* *fs)
block write count for statistic

Return

- block write count
- -EINVAL: invalid SFFS pointer

Parameters

- `fs`: SFFS pointer

int **`sffsSetSfileReserveCount`** (*sffsFs_t* *fs, size_t *count*)

set reserved block count only for sffsSfileWrite

Due to `sffsSfileWrite` will need extra blocks, this can set reserved block count, which will only be used during `sffsSfileWrite`.

In most cases, `sffsSfileWrite` is used for important information, and failure is unacceptable. With proper reserved blocks, it won't fail due to ENOSPC.

Even current free block count is less than `count`, it will success.

Another approach is to create independent partition for important files.

Return

- 0 on success
- -EINVAL on invalid parameter

Parameters

- `fs`: SFFS pointer
- `count`: reserved block count only for sffsSfileWrite

CHAPTER

ELEVEN

CFW EVENT DISPATCH

Contents

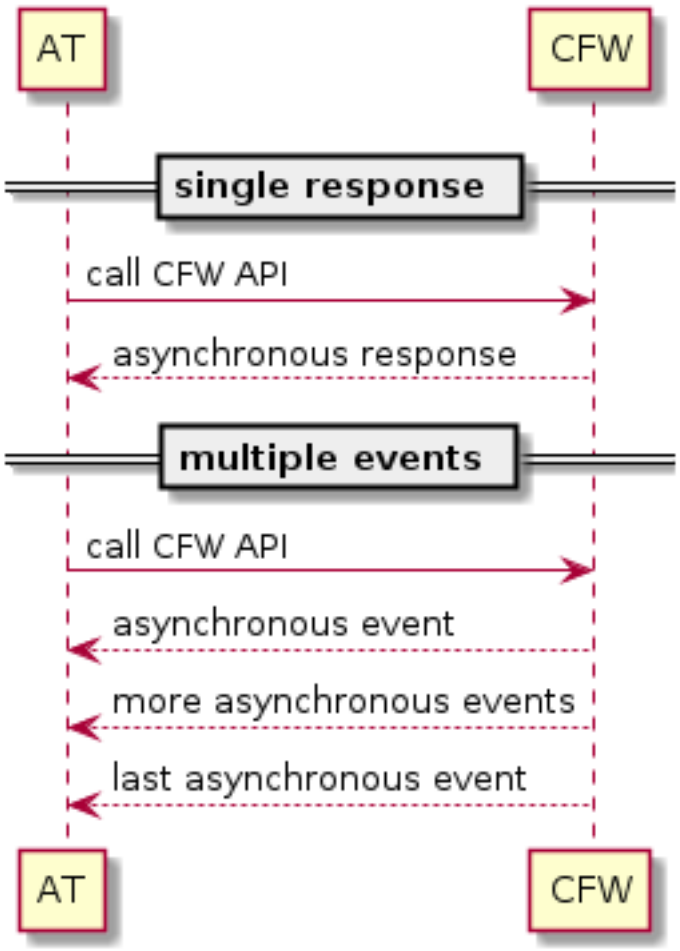
11.1 Overview

There are 2 models of CFW APIs:

- synchronous: after the API returns, the transaction is done;
- asynchronous: after the API returns, there are one or more events belongs to the transaction.

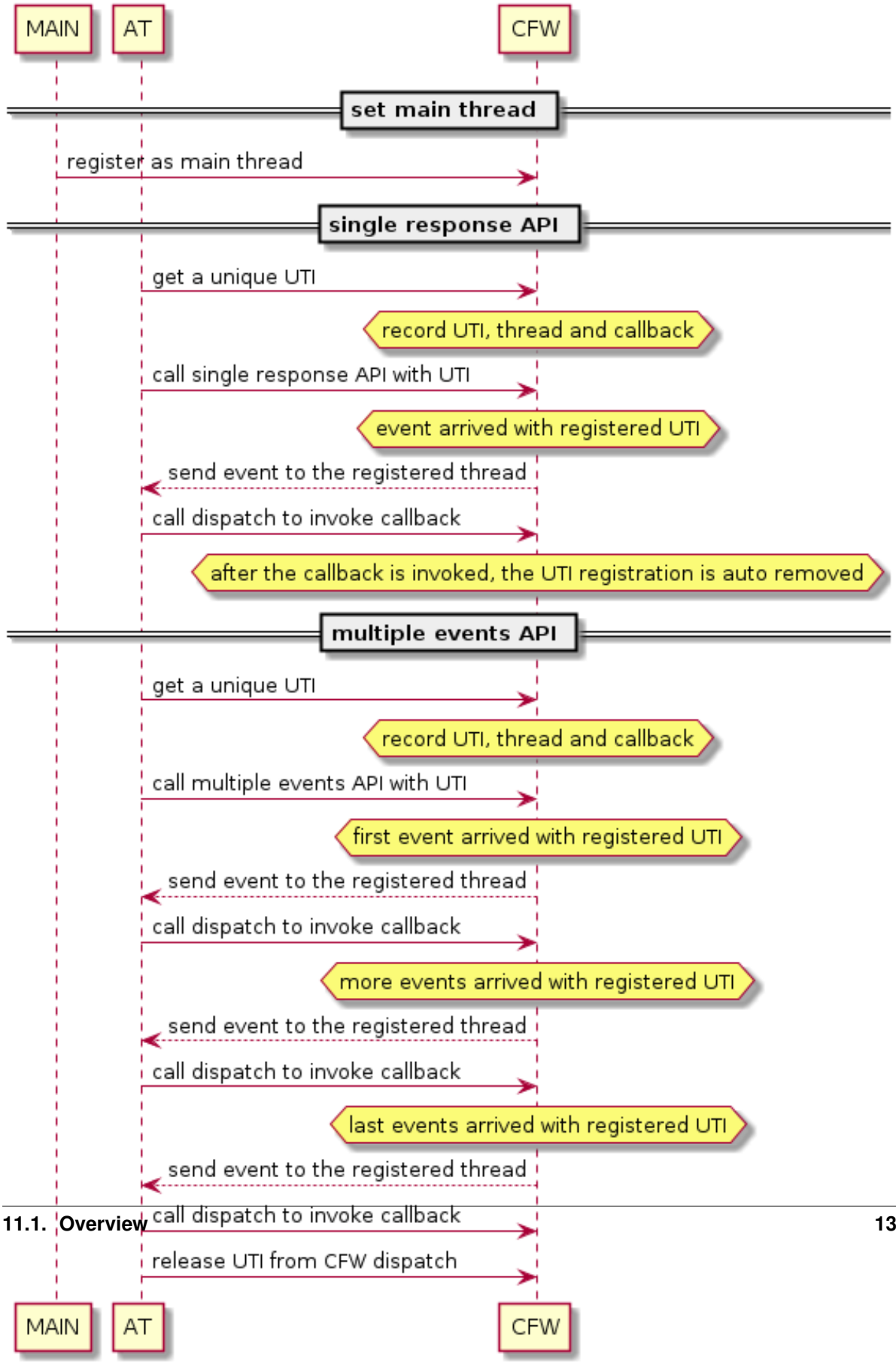
And there are 2 models of asynchronous CFW APIs:

- single response: after the API returns, there is only one response event;
- multiple events: after the API returns, there are more than one response events. And the last event may be different in different cases.



Asynchronous events should be sent to caller thread. However, caller thread information is not recorded. CFW dispatch is a component to record where to send response events. CFW dispatch use UTI to register and dispatch events.

Also, there is another kind of event, which is not bound to API call. This kind of events will be sent to one thread, called *main thread*. It can be a separated thread, or reuse another thread. For example, in module product, AT thread will be reused as *main thread*.


MAIN
AT
CFW
set main thread
register as main thread
single response API
get a unique UTI
record UTI, thread and callback
call single response API with UTI
event arrived with registered UTI
send event to the registered thread
call dispatch to invoke callback
after the callback is invoked, the UTI registration is auto removed
multiple events API
get a unique UTI
record UTI, thread and callback
call multiple events API with UTI
first event arrived with registered UTI
send event to the registered thread
call dispatch to invoke callback
more events arrived with registered UTI
send event to the registered thread
call dispatch to invoke callback
last events arrived with registered UTI
send event to the registered thread
call dispatch to invoke callback
release UTI from CFW dispatch
MAIN
AT
CFW

11.2 UTI Management

`CFW_GetFreeUTI` will be used to allocate UTI. CFW will ensure the uniqueness of UTI.

11.3 Thread Safe

CFW dispatch registration is thread safe.

11.4 API Reference

Defines

CFW_UTI_INVALID
indicate invalid UTI or UTI unavailable

Functions

void **cfwDispatchInit**(void)
CFW event dispatch initialization.

void **cfwDispatchSetMainThread**(*osiThread_t* *thread)
set CFW event dispatch main thread

When the UTI of CFW event is not registered, the event will be sent to the *main thread*.

Parameters

- `thread`: main thread to be set

uint16_t **cfwRequestUTI**(*osiEventCallback_t* cb, void *cb_ctx)
request a UTI, response to current thread

Request a UTI, and set the CFW event with the specified UTI will be send to current thread, that is the caller thread.

Return

- requested UTI
- CFW_UTI_INVALID if no free UTI

Parameters

- `cb`: callback to be invoked when UTI matches
- `cb_ctx`: callback context

uint16_t **cfwRequestUTIEx**(*osiEventCallback_t* *cb*, void **cb_ctx*, *osiThread_t* **thread*, bool *auto_remove*)
request a UTI with more options

The thread for the arriving matched UTI can be specified.

Normally, the UTI registration will be removed automatically when the first event is handled in *cfwInvokeUtiCallback*. In case there are more than one event with the UTI will arrive, *auto_remove* can be set to false. Then the UTI registration won't be removed automatically. Rather, it should be released manually.

Return

- requested UTI
- CFW_UTI_INVALID if no free UTI

Parameters

- `cb`: callback to be invoked when UTI matches
- `cb_ctx`: callback context
- `thread`: the thread should receive response event, NULL for current thread
- `auto_remove`: auto remove the UTI registration after callback is invoked

uint16_t **cfwRequestNoWaitUTI**(void)
request a UTI, and not register response thread

Only when the caller doesn't care about the response event, it should be called. When the response arrived, the caller thread may not be able to receive the response event.

The requested UTI will be reused automatically. It is not needed and not permitted to call `cfwReleaseUTI`.

Return

- requested UTI
- CFW_UTI_INVALID if no free UTI

void **cfwReleaseUTI**(uint16_t *uti*)
request a registered UTI with response to current thread

When it is known that response event won't arrive, for example, the call returns error, it *must* be called. Otherwise, the UTI resource will be leaked.

When there are registered (UTI, thread) combination can match, it will return silently.

Parameters

- `uti`: the UTI to be released

void **cfwDispatchSendEvent**(**const** *osiEvent_t* **event*)
dispatch CFW event

When CFW event is about to be sent, this *must* be called. And it will ensure the event to be sent to proper thread.

The event *MUST* be CFW event, that is, the third parameter follows CFW event convention.

It won't fail to send. When the UTI is registered, it will be sent to the registered thread. Otherwise, it will be sent to the main thread. And if main thread is not set, it will be dropped.

Parameters

- `event`: the event to be sent, and must be CFW event

bool **`cfwIsCfwEvent`**(uint32_t *id*)
check whether the event ID is CFW event

Return

- true if it is CFW event
- false if it is not CFW event

Parameters

- `id`: the event ID

bool **`cfwIsCfwIndicate`**(uint32_t *id*)
check whether the indicate ID is CFW indicate event

Return

- true if it is CFW indicate event
- false if it is not CFW indicate event

Parameters

- `id`: the indicate ID

bool **`cfwInvokeUtiCallback`**(**`const`** *osiEvent_t* **event*)
invoke registered callback

When a registered callback is invoked, and the event id matches the registered last UTI or the registration doesn't specify event id, it will be unregistered automatically

Return

- true if the registered callback is invoked
- false if `event` is not a CFW_EVENT, or the UTI is not registered

Parameters

- `event`: input event, must be `CFW_EVENT`

CHAPTER

TWELVE

IPC (INTER-PROCESSOR COMMUNICATION)

Contents

12.1 Overview

IPC is the mechanism to communicate with CP. There are several pre-defined channels:

- AT1: not used now
- AT2: reused for RPC
- PS: PS packets
- A2C_CTRL: IPC command.
- AUD_CTRL: for audio, not used now

There are 4 types of IPC channel:

- stream (AT1, AT2)
- packets (PS)
- queue (A2C_CTRL)
- command (AUD_CTRL)

The APIs for all channel types are the same. But the meaning has slight differences.

In each channel, there is uplink (from AP to CP) and downlink (from CP to AP) sub-channels. These 2 sub-channels have separated shared memory and control information.

IPC channels are using shared memory with CP. The layout of the shared memory should be shared by both AP and CP. So, when the layout changed, AP and CP codes must be synchronized.

The exception is PS buffer count. At SP initialization, the configuration will be written to registers, CP will used the value from registers. However, the buffer used can't exceed reserved shared memory size.

12.2 Shared Registers

Register	Content
hwp_mailbox->sysmail4	PS uplink big buffer count
hwp_mailbox->sysmail5	PS uplink little buffer count
hwp_mailbox->sysmail6	PS downlink big buffer count
hwp_mailbox->sysmail7	PS downlink little buffer count
hwp_mailbox->sysmail8	MD version offset in shared memory
hwp_mailbox->sysmail9	Exception cause offset in shared memory
hwp_sysCtrl->cfg_reserve9	Write 0 after AP has responsed to mailbox interrupt

12.3 Shared Memory Layout

It is just an example, the detailed offset depends on various configurations.

offset	size	content
	0x100	MD_EXEC_CAUSE
	0x28	MD_VERSION
	0x200 * 308	LITTLE DL ip data
	0x200 * 1580	BIT DL ip data
	0x200 * 140	LITTLE UL ip data
	0x100 * 1540	BIT UL ip data
	0x200 * 4	LITTLE DL vol free_buf_off
	0x200 * 4	BIG DL vol free_buf_off
	0x200 * 4	LITTLE DL net free_buf_off
	0x200 * 4	BIG DL net free_buf_off
	0x200 * 4	LITTLE UL_free_buf_off
0x2790	0x100 * 4	BIG UL_free_buf_off
	0x10	(padding)
	0x800	PS DL control data
	0x800	PS UL control data
	0x80	(padding)
	0x200	AUD DL data
	0x200	AUD UL data
	16 * 8	A2C DL data

Continued on next page

Table 1 – continued from previous page

offset	size	content
	16 * 8	A2C UL data
	0x400	AT2 DL data
	0x400	AT2 UL data
	0x400	AT1 DL data
0x200	0x400	AT1 UL data
	0x4c	(padding)
	0x2c x2	(SYSCMD) AUD_CTRL channel control (UL/DL)
	0x2c x2	(QUEUE) A2C_CTRL channel control (UL/DL)
	0x2c x2	(PACKET) PS channel control (UL/DL)
	0x2c x2	(STREAM) AT2 channel control (UL/DL)
0	0x2c x2	(STREAM) AT1 channel control (UL/DL)

12.4 Thread Safe

IPC APIs are thread safe, except open, close, initialization and etc. It is safe to call read and write in multiple thread, or ISR.

The return value of `ipc_ch_read_avail` and `ipc_ch_write_avail` is *volatile*. That is, multiple calls may return different values. It should be considered at usage.

12.5 API Reference

Defines

IPC_PS_BUF_UL_LEN_B
PS uplink buffer maximum size

When write to PS IPC channel, the maximum packet size can't exceed this.

IPC_PS_UL_HDR_LEN
PS uplink buffer header size

When copy data to PC uplink buffer, the payload should be copied to the allocated buffer with is offset.

Enums

enum smd_ch_id
IPC channel ID.

Values:

SMD_CH_AT = 1
alias for SMD_CH_AT1

SMD_CH_AT1 = 1
AT1, stream type.

SMD_CH_AT2 = 2
AT2, stream type.

SMD_CH_PS = 3
PS, packet type.

SMD_CH_A2C_CTRL = 4
A2C_CTRL, queue type.

SMD_CH_AUD_CTRL = 5
AUD_CTRL, command type.

SMD_CH_MAX
placeholder for count

enum smd_ch_flag
channel Event

Values:

CH_READ_AVAIL = 0x1
indicate read available

CH_WRITE_AVAIL = 0x2
indicate write available

CH_OPENED = 0x4
flag for channel opened

CH_RW_MASK = (*CH_READ_AVAIL* | *CH_WRITE_AVAIL*)
read and write mask

Functions

void **ipcInit** (void)
IPC module initialization.

int **ipc_ch_open** (int *ch_id*, **struct** smd_ch ***ch*, void **priv*, void (**notify*)) void *, uint32_t
open an IPC channel

Each IPC channel can be opened only once.

It is permitted `notify` to be NULL.

`notify` will be called in ISR context. So, it should follow ISR programming guide. In the notify itself, it is not permitted to read or write the channel

Return

- 0 on success
- -ENODEV/-EPERM on error

Parameters

- `ch_id`: channel ID
- `ch`: output channel pointer
- `priv`: channel notify callback first argument
- `notify`: channel notify callback

void **ipc_ch_set_event_mask**(**struct** smd_ch *ch, uint32_t *event_mask*)
set IPC channel event mask

Only masked event will invoke the registered `notify` callback.

The default event mask is 0. That is the callback won't be invoked.

Parameters

- `ch`: IPC channel, must be valid
- `event_mask`: event mask to be set

int **ipc_ch_read**(**struct** smd_ch *ch, void *data, uint32_t *len*)
read from IPC channel

The read content depends on channel type:

- stream: read data. The return value is the actual read bytes, and may be less than `len`.
- packet: `data` is the packet offset, `len` is the offset count rather than size of `data`. However, the return value is the read bytes.
- command: `data` is the command including header and data, the return value is the actual read bytes.
- queue: `data` is *ipc_cmd*, `len` is the count of *ipc_cmd*, the return value is the actual read bytes

Return

- actual read
- -EINVAL, -EAGAIN on fail

Parameters

- `ch`: IPC channel, must be valid
- `data`: data to be read
- `len`: data length

int **ipc_ch_write**(**struct** smd_ch *ch, **const** void *data, uint32_t *len*)
write to IPC channel

The write content depends on channel type:

- stream: write data. The return value is the actual written bytes, and may be less than `len`.
- packet: `data` is the packet offset, `len` is the offset count rather than size of `data`. However, the return value is the written bytes.

- command: write content is command (includes header and data). When `size` is 0, the control fifo will be cleared.
- queue: `data` is *ipc_cmd*, `len` is the count of *ipc_cmd*, the return value is the actual read bytes

After write, even partial write, it will set peer's interrupt.

Return

- actual written
- 0 for no space, and won't set peer's interrupt
- -ENODEV if the uplink channel is not openned
- -EINVAL for invalid parameters

Parameters

- `ch`: IPC channel, must be valid
- `data`: data to be written
- `len`: data length

int **ipc_ch_read_avail**(**struct** smd_ch *ch)
channel available read space

The meaning of the return unit depends on channel type:

- stream: available bytes
- packet: available offset count
- command: available bytes including command header
- queue: available count of *ipc_cmd*

Return

- available
- 0 for unavailable

Parameters

- `ch`: IPC channel, must be valid

int **ipc_ch_write_avail**(**struct** smd_ch *ch)
channel available write space

The meaning of return unit depends on channel type:

- stream: available write bytes
- packet: available offset write count
- command: available write bytes of command data (except command header)
- queue: available count of *ipc_cmd*

Return

- available
- 0 for no space

Parameters

- `ch`: IPC channel, must be valid

int **ipc_free_dl_ps_buf**(**struct** smd_ch *ch, uint32_t *offset*)
free downlink PS buf

Even if the downlink PS buf is in a chain, this will free the specified downlink PS buf only. Other buf in the chain is untouched.

Return

- 0 in success, it will never fail

Parameters

- `ch`: the PS IPC channel, must be valid
- `offset`: the offset from share memory of the downlink buf

int **ipc_alloc_ul_ps_buf**(**struct** smd_ch *ch, uint32_t *size*)
allocate uplink PS buf

The size will be used to decide big buffer or little buffer. It doesn't mean that the allocated buffer is large enough for requested `size`.

Return

- uplink buf offset in shared memory
- -ENOMEM if no available buffer

Parameters

- `ch`: the PS IPC channel, must be valid
- `size`: request size

void **ipc_notify_cp_assert**(void)
notify AP panic to CP

It should be called after AP panic.

void **ipc_switch_cp_trace**(bool *en*)
switch cp trace

Parameters

- `en`: enable cp trace or not

void **ipc_notify_sim_detect**(int *num*, bool *connect*)
notify sim plug in/out to CP

Parameters

- `num`: sim number (1/2)
- `connect`: true indicate the sim plug in otherwise plug out

void **ipc_register_audio_notify**(void (**handler*)) void

void **ipc_register_trace_notify**(void (**handler*)) uint32_t, void *
, void **param*

struct ipc_cmd
#include <*drv_md_ipc.h*> IPC command header

Public Members

uint32_t **id**
command id

uint32_t **para0**
1st parameter

uint32_t **para1**
2nd parameter

uint32_t **para2**
3rd parameter

struct ps_header
#include <*drv_md_ipc.h*> IPC PS buf header

Public Members

uint32_t **next**
next offset (from shared memory) in the chain

uint8_t **cid**
CID.

uint8_t **simid**
SIM number.

uint16_t **len**
data length

uint32_t **flag**
BUF_IN_USR or BUF_IN_IDLE.

uint32_t **buf_size**
300B or 1600B

uint32_t **data_off**
: data offset in buffer

uint32_t **id**
: only for debug

CHAPTER

THIRTEEN

PS IPC INTERFACE

Contents

13.1 Overview

PS IPC is a IPC channel designed for PS packets. There is only one PS IPC channel in the system. Packets are identified by combination of SIM number and CID.

PS interface is an abstract layer, which is bind to one combination of SIM number and CID.

13.2 Thread Safe

Shared resources access are protected by `osiMutex`. So, PS IPC interface is thread safe.

13.3 API Reference

Typedefs

typedef struct drvPsIntf **drvPsIntf_t**
opaque data structure for PS interface

typedef void (***drvPsIntfDataArriveCB_t**)(void *ctx, *drvPsIntf_t* *p)
callback function of PS interface data arrival

The callback is executed in PS path thread, or the thread calling `drvPsPathDataArrive`. Inside the callback, it is permitted to call `drvPsIntfRead`. However, it is not permitted to call `drvPsInterfaceOpen` and `drvPsIntfClose`.

Functions

void **drvPsPathInit** (void)
initialize PS path module

void **drvPsPathDataArrive** (uint8_t *sim*, uint8_t *cid*, **const** void **data*, uint16_t *size*)
PS data arrive callback in external source.

For external source, this API is implemented inside this module and called by external source.

For non-external source, this API may be unimplemented.

Parameters

- `sim`: SIM id
- `cid`: CID
- `data`: PS data
- `size`: PS data size

int **drvPsPathDataSend** (uint8_t *sim*, uint8_t *cid*, **const** void **data*, uint16_t *size*)
PS data send function.

For extenal source, this API will be called in this module, and sould be implemented in other modules.

For non-external source, this API won't be called.

Return

- sent size, it should be equal to `size` on success
- -1 on failed

Parameters

- `sim`: SIM id
- `cid`: CID
- `data`: PS data
- `size`: PS data size

drvPsIntf_t ***drvPsIntfOpen** (uint8_t *sim*, uint8_t *cid*, *drvPsIntfDataArriveCB_t* *cb*, void **cb_ctx*)
open a PS interface

Each sim and cid combination is an interface. For each interface, it can be opened only once.

Return

- PS interface pointer

- NULL on error
  - the SIM/CID is already opened
  - invalid sim cid combination

Parameters

- `sim`: SIM number
- `cid`: CID
- `cb`: callback of data arrival
- `cb_ctx`: callback contex pointer

void **drvPsIntfClose**(*drvPsIntf_t* *p)

close a PS interface

When `p` is NULL or the interface is already opened, there are no operations.

Parameters

- `p`: PS interface pointer

drvPsIntfDataArriveCB_t **drvPsIntfSetDataArriveCB**(*drvPsIntf_t* *p, *drvPsIntfDataArriveCB_t* *cb*)

change PS interface data arrive callback

When `cb` is NULL, the registered callback won't be changed. And the original callback will be returned.

Return the original data arrive callback

Parameters

- `p`: PS interface pointer, can't be NULL
- `cb`: data arrive callback

int **drvPsIntfRead**(*drvPsIntf_t* *p, void *data, size_t *size*)

read from PS interface

When there are no data in PS interface, return 0.

When the input `size` is too small for one packet, return -1. The maximum packet size is `PS_BUF_DL_LEN_B`.

Return

- read size
- 0 if no data
- -1 if input `size` is too small for one packet

Parameters

- `p`: PS interface pointer, can't be NULL

- `data`: memory for read, can't be NULL
- `size`: memory size

int **`drvPsIntfReadAvail`** (*drvPsIntf_t* *p)
available read size of PS interface

Return available read size

Parameters

- `p`: PS interface pointer, can't be NULL

int **`drvPsIntfWrite`** (*drvPsIntf_t* *p, **`const`** void *data, size_t *size*)
write to PS interface

This won't wait available uplink buffer. When uplink buffer is unavailable, this will return 0.

Return

- 0 is uplink buffer is unavailable
- written size, it should be equal to `size` on success
- -1 if `data` is NULL or `size` is too large for one packet

Parameters

- `p`: PS interface pointer, can't be NULL
- `data`: data to be written
- `size`: data size

int **`drvPsIntfWriteMulti`** (*drvPsIntf_t* *p, size_t *count*, **`const`** void *data[], size_t *size*[])
write multiple buffers to PS interface

When `data`[n] is NULL or `size`[n] is too large, the buffers from it won't be written.

This won't wait available uplink buffer. So, the returned written size will be smaller than provided total size.

Return

- written bytes, may be smaller than provided total size

Parameters

- `p`: PS interface pointer, can't be NULL
- `count`: buffer count
- `data`: array of buffer address to be written
- `size`: array of buffer size to be written

CHAPTER

FOURTEEN

RPC (REMOTE PROCEDURE CALL)

Contents

- *API Reference*

14.1 Overview

RPC is built on top of IPC, and provides high level inter-processor communication.

From application's point of view, RPC provides:

- Call functions implemented on remote processor;
- Send event to remote processor;
- Receive event from remote processor;

From implementation's point of view, RPC send and receive variable length packets though IPC channel. On each side, there is a dedicated thread to read packets from IPC channel, and process the packets.

Each level of RPC packet headers are designed to be 8 bytes aligned. So, RPC packet data can be safely casted into struct pointer, even `int64_t` or `double` data type are used in struct.

The common packet header is:

opcode	4 bytes	Type of the RPC packet
size	4 bytes	Size of RPC packet, including `opcode`

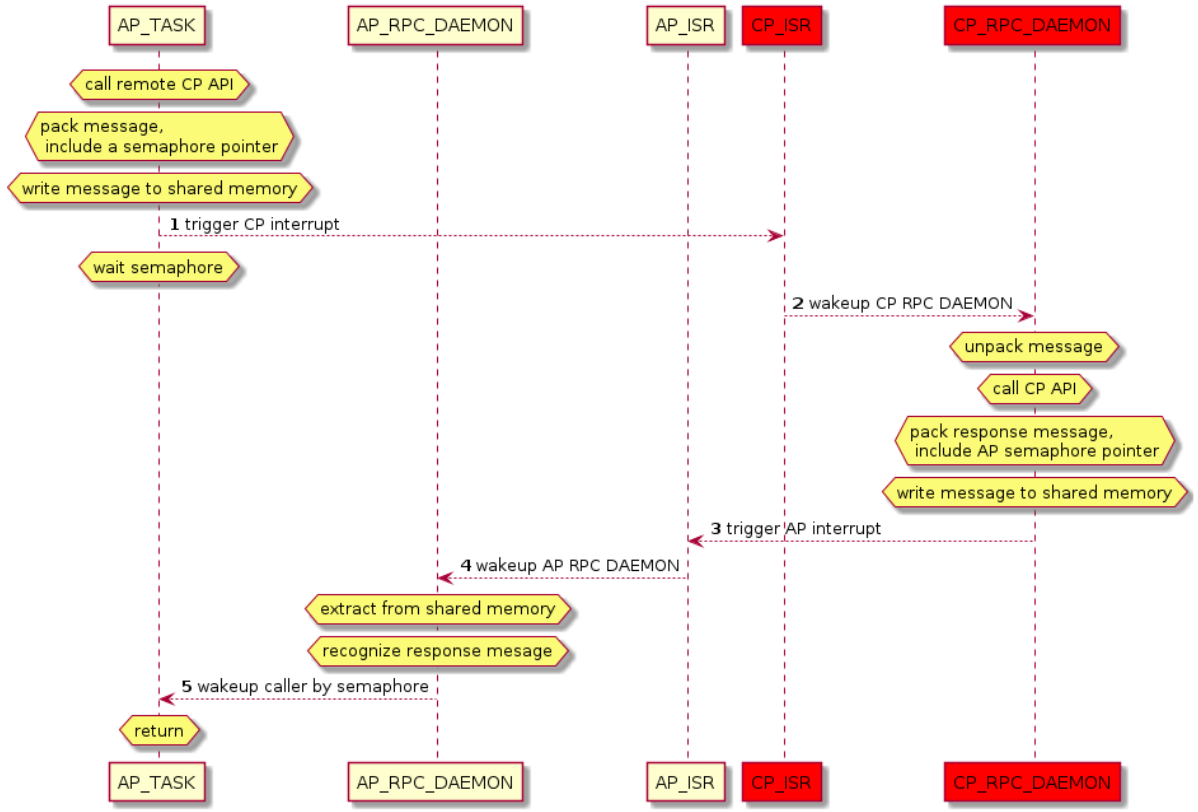
There are 3 kinds of `opcode` defined:

CALL The RPC packet contains *packed* input parameters to call a function on receiver.

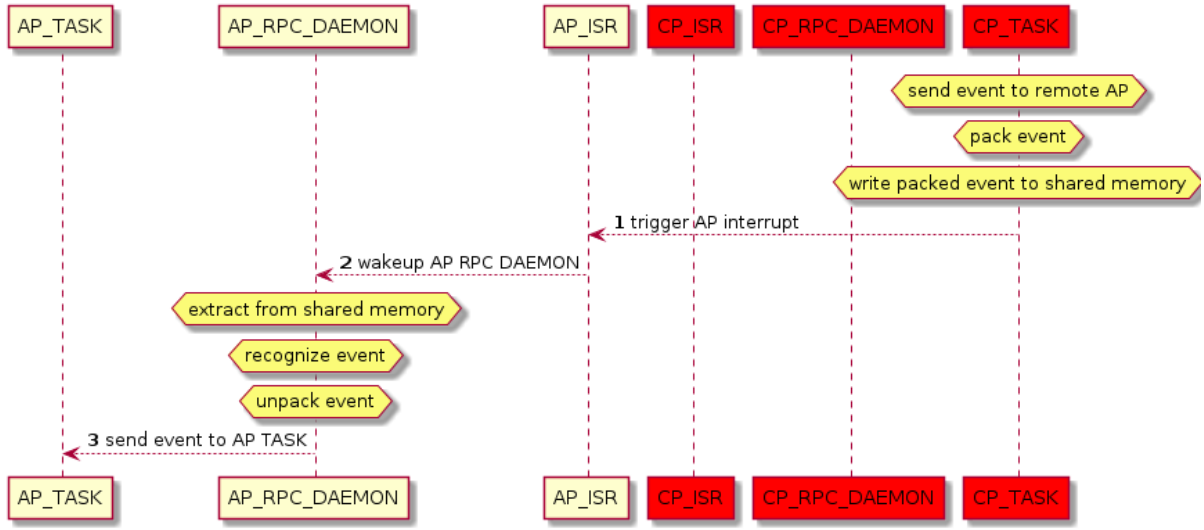
RESPONSE The RPC packet contains results and *packed* output parameters.

EVENT The RPC packet contains *packed* event.

The procedure of RPC function call is:



The procedure of remote event is:



14.2 Function Call

The format of *CALL* packet header:

api_tag	4bytes	Tag to indicate function to be called.
caller_sync	4bytes	Should passed to *RESPONSE*
caller_rsp_ptr	4bytes	Should passed to *RESPONSE*
seq	2bytes	(not important) sequence number
rsp_size	2bytes	*RESPONSE* packet size

The format of *RESPONSE* packet header:

api_tag	4bytes	Copied back
caller_sync	4bytes	Copied back, release it on received
caller_rsp_ptr	4bytes	Copied back, copy response content there
seq	2bytes	Copied back
rpc_error_code	2bytes	0, or ENOENT(2) if function not found

14.3 Event

Event is 4 words message, and the first word is event ID. During event send and receive, these 4 words themselves are transferred by value, not the event struct pointer. So, when the carried information can be represented by 3 words, malloc and free are not needed.

ID	4 bytes
param1	4 bytes
param2	4 bytes
param3	4 bytes

When the carried information can't be represented by 3 words, pointer can be transferred to receiver. Sender and receiver should have the same understanding of the content of event. Usually, when pointer is carried, the memory is allocated by sender and freed by receiver.

When there are no pointers in event, which is called *plain event*, it is trivial to send event to remote processor, just send the 4 words.

When there are pointers in event, it is needed to *pack* the content rather than send the pointer itself. Usually, one processor should access, especially free pointer of another processor. Also, receiver should *unpack* the content, allocate memory and copy the content to local pointer. Then the event receiver can process in the same way for local processor event and remote processor event. In this case, sender processor should implement an event *packer*, and receiver processor should implement an event *unpacker*.

14.4 Code Generation

A lot of codes are needed for *pack* and *unpack*, and it is tedious to write them manually. So, `rpcgen.py` is developed for automatic code generation. The input in XML to describe remote function call and remote event, and the following codes are generated:

Function call sender:

```
uint32_t FunctionOnCP(uint32_t param) {
    pack_input_parameters();
    send_through_IPC_channel();
    wait_response();
    return result_from_response;
}
```

Function call receiver:

```
void RPC_FunctionOnCP(void *in, void *out) {
    unpack();
    opar->result = FunctionOnCP(ipar->param);
}
```

Event sender:

```
void SEND_REMOTE_EVENT_ID(rpcChannel *p, const void *event) {
    pack_event();
    rpcSendPackedEvent(p, packed_event);
}
```

Event unpacker:

```
void RECV_REMOTE_EVENT_ID(rpcEventHeader_t *event) {
    unpack_event();
}
```

Function receiver dispatch:

```
void *rpcFindFunctionByTag(unsigned tag);
```

Event sender dispatch:

```
void *rpcFindEventSender(unsigned id);
```

Event unpacker dispatch:

```
void *rpcFindEventUnpacker(unsigned id);
```

Note: Only limited format of event sender and unpacker can be automatic generated. If they can't be automatic generated, it is needed to implement sender and unpacker manually.

14.5 XML by Example

14.5.1 XML Header

Example:

```
<rpc module="rpc_a2c" sender="ap" receiver="cp">
```

Generated file names are constructed by *module*, *sender* and *receiver*.

sender is the initiator of the transaction. For example, when AP will call a function implemented on CP, then AP is the *sender* and CP is the *receiver*. When CP will send an event to AP, then CP is the *sender* and AP is the *receiver*.

- `rpc_a2c_api.h`: generated function prototype, which can be shared by sender and receiver.
- `rpc_a2c_par.h`: generated struct used by function sender and receiver.
- `rpc_a2c_ap.c`: generated codes, which should be used by sender (ap).
- `rpc_a2c_cp.c`: generated codes, which should be used by receiver (cp).

14.5.2 API

Example:

```
<api name="function_name_on_receiver" condition="CONDITION">
    <SKIP reason="some reason">
    <param_in name="in_value" type="uint32_t"/>
    <param_in name="in_array" type="uint8_t" array_size="16"/>
    <param_in name="in_ptr" type="DATA_STRUCT" pointer="true"/>
    <param_out name="out_value" type="uint32_t"/>
    <param_out name="out_array" type="uint8_t" array_size="20"/>
    <param_inout name="inout_array" type="uint16_t" array_size="4"/>
    <return type="uint32_t"/>
</api>
```

name: Function name

condition: This API will exist conditionally. All generated codes related to this API will be enclosed inside `#if CONDITION`.

SKIP: This API will be ignored by generator. *reason* is a kind of comment.

param_in: Indicate an input parameter, which shall be sent from sender to receiver, in *CALL* RPC package.

param_out: Indicate an output parameter, which shall be sent from receiver to sender, in *RESPONSE* RPC package.

param_out: Indicate an input and output parameter, which shall be sent from sender to receiver, in *CALL* RPC packages, also it will be sent from receiver to sender, in *RESPONSE* RPC package.

return: Return value. When the function doesn't return value, it should be absent.

type: The data type.

array_size: Indicate the parameter is a fixed length array. It can be used in both param_in and param_out.

pointer: Indicate the input parameter is a pointer in prototype. It can only be used in param_in.

Note: RPC code generation can't handle pointers in *DATA_STRUCT*.

The generated function prototype will be:

```
uint32_t function_name_on_receiver(uint32_t in_value,
            uint8_t in_array[16], DATA_STRUCT *in_ptr,
            uint32_t *out_value, uint8_t out_array[20]);
```

14.5.3 Plain input parameter

Example:

```
uint32_t in_value
<param_in name="in_value" type="uint32_t"/>
```

14.5.4 Input struct using pointer

Example:

```
DATA_STRUCT *in_value
<param_in name="in_value" type="DATA_STRUCT" pointer="true"/>

const DATA_STRUCT *in_value
<param_in name="in_value" type="DATA_STRUCT" pointer="true" const="true"/>
```

14.5.5 Input fixed length array

Example:

```
uint8_t in_value[10]
<param_in name="in_value" type="uint8_t" array_size="10"/>
```

14.5.6 Input variable size

Example:

```
uint8_t *in_mem, uint8_t in_size
<param_in name="in_mem" type="uint8_t *" var_size="in_size"/>
<param_in name="in_size" type="uint8_t"/>
```

var_size should be another input parameter name.

14.5.7 Input variable length struct

Example:

```
type struct {
    size_t size; // size of var_data
    ......
    char var_data[0];
} DATA_STRUCT;
DATA_STRUCT *in_value
<param_in name="in_value" type="DATA_STRUCT *"
    svar_size="sizeof(DATA_STRUCT) + in_value->size"/>
```

svar_size can be any C expression, with constants and member of current parameter.

Note: For *var_size* and *svar_size*, one extra byte filled with zero will be send to *receiver*, in case it means \0 terminated string length, and receiver expect \0 after the size.

14.5.8 Input \0 terminated string

Example:

```
uint8_t *str
<param_in name="str" type="uint8_t *" cstring="true"/>
```

14.5.9 Plain output and inout parameter

Example:

```
uint32_t *out_value
<param_out name="out_value" type="uint32_t"/>
uint32_t *inout_value
<param_inout name="inout_value" type="uint32_t"/>
```

14.5.10 Output and inout fixed length array

Example:

```
uint8_t out_value[10]
<param_out name="out_value" type="uint8_t" array_size="10"/>
uint8_t inout_value[10]
<param_inout name="inout_value" type="uint8_t" array_size="10"/>
```

14.5.11 Output peer's pointer

Example:

```
uint8_t **peer_ptr
<param_out name="peer_ptr" type="uint8_t *">
    <comment text="peer's pointer">
</param_out>>
```

Peer's pointer can be returned as normal. However, this usage model is not recommended. Only in case it is hard to re-factor the codes, it can be considered. Also:

- The content of the pointer is *const* on peer. Otherwise, it is possible to get wrong content due to cache coherency.
- It is known that peer's memory is accessible (for example, not inhibited by MMU).
- Memory map on both side are the same.

14.5.12 Output and inout variable length

NOT SUPPORTED.

14.5.13 Event

Example:

```
<event name="EV_PLAIN_ID" id="10001" format="plain"/>
<event name="EV_POINTER_ID" id="10002" format="pointer"
    ptr1_size="sizeof(DATA_STRUCT)"/>
<event name="EV_MANUAL_ID" id="10003" format="manual"/>
```

plain event are not needed to be listed in XML. Actually, they will be ignored silently.

pointer event sender and unpacker will be automatic generated. The generated code looks like:

```
void SEND_EV_POINTER_ID(rpcChannel *p, const void *event) {
    rpcSendPointerEvent(p, event, sizeof(DATA_STRUCT), 0);
}
bool RECV_EV_POINTER_ID(rpcEventHeader_t *event) {
    return rpcUnpackPointerEvent(event, sizeof(DATA_STRUCT), 0);
}
```

For *pointer event*, when `ptr1_size` is non-zero, and `param1` is not NULL, `param1` will be pack and unpack as a pointer. When `ptr2_size` is non-zero, and `param2` is not NULL, `param2` will be pack and unpack as a pointer.

14.6 Event Router

RPC daemon itself doesn't know what to do at receiving event from peer. `rpcRegisterEvents` should be called to register event handler. However, this event handler should just send the event to another thread, rather than process the event directly.

14.7 Dead Lock

Due to all RPC packets are handled in one RPC daemon thread, some cases will cause dead lock, and they should be avoided.

- If a function in CP will be called by AP, the function itself can't call AP functions. However, sending events to AP won't cause dead lock.
- Event router can't call remote function.

Note: The reason of dead lock is *RESPONSE* packets are handled in RPC daemon thread also. In RPC daemon thread itself, there are no opportunities to handle *RESPONSE* packet, and remote call can't finish.

It is possible to handle *RESPONSE* in another thread or ISR, and it can eliminate the dead lock. However, the complexity isn't worth.

14.8 Command Queue

There are **no** command queue in RPC design. Due to commands are handled in one thread of peer, command queue won't help too much.

Taking example that AP will call CP's function, when CP function is blocking, not only the AP caller is blocked, any other AP thread to use RPC channel will be blocked.

Note:

- Any function to be called by peer should be simple enough.
- Delay sensitive functions shouldn't use RPC channel. It is always possible that peer's function executing will take time, for example, preempted by high priority thread.

14.9 API Reference

Typedefs

typedef struct rpcChannel **rpcChannel_t**
Opaque type for RPC channel

typedef void (***rpcEventRouter_t**) (void *ctx, **const** void *event)
Function type to route (send out) event

typedef void (***rpcFunction_t**) (void *in, void *out)
Function type of RPC wrapper for local implemented API

typedef void (***rpcEventSender_t**) (*rpcChannel_t* *ch, **const** void *event)
Function type to pack event

typedef bool (***rpcEventUnpacker_t**) (*rpcEventHeader_t* *event)
Function type to unpack event

Functions

rpcChannel_t ***rpcChannelOpen** (void)
open RPC channel

Though only one RPC channel is designed now (that is the reason there are no parameter at open), RPC channel pointer is used in all RPC APIs.

RPC channel can be shared. So, it will return the same pointer at further open.

There are no "close" API. In current design, RPC channel won't be closed.

Return the RPC channel pointer

void **rpcSendEvent** (*rpcChannel_t* *ch, **const** void *event)
send an event to peer

The event is generic event (4 words), and the first word MUST be event ID. Sending method will depend on event ID, and event packer will be called inside.

Parameters

- `ch`: the RPC channel
- `event`: the event to be send

bool **rpcRegisterEvents** (*rpcChannel_t* *ch, uint32_t *start*, uint32_t *end*, *rpcEventRouter_t* *router*, void *router_ctx*)
register event router

RPC daemon itself doesn't know how to route the event from peer. Application should register router for specified event range.

The router should only send the event to corresponding thread. The router itself shouldn't handle the event.

When range and callback are existed, no duplicated router will be added. It is application's duty to decide whether overlapped ranges are permitted.

Return

- true registration success
- false registration failed, due to invalid parameters or there are too many routers.

Parameters

- `ch`: the RPC channel
- `start`: event range start (inclusive)
- `end`: event range end (inclusive)
- `router`: event router
- `router_ctx`: context for router callback

int **rpcSendCall**(*rpcChannel_t *ch*, *rpcCallHeader_t *call*)

send function call packet to peer

RPC packets are 8 bytes aligned. When the size in input header is not 8 bytes aligned, it will be changed to 8 bytes aligned. That is, event->h.size may be changed inside.

Most likely, it won't be called directly by application. It will be called only in RPC daemon and RPC stubs.

Return

- 0 success
- others error. Only when peer can't find the function, it will return error (-ENOENT).

Parameters

- `ch`: the RPC channel
- `call`: the constructed call header

void **rpcSendPackedEvent**(*rpcChannel_t *ch*, *rpcEventHeader_t *event*)

send a packed event to peer

When all content needed to send to peer are packed after the header, it is "packed event". event->h.size is the data size needed to send to peer.

RPC packets are 8 bytes aligned. When the size in input header is not 8 bytes aligned, it will be changed to 8 bytes aligned. That is, event->h.size may be changed inside.

Most likely, it won't be called directly by application. It will be called only in RPC daemon and RPC stubs.

Parameters

- `ch`: the RPC channel
- `event`: the packed event

void **rpcSendPlainEvent**(*rpcChannel_t *ch*, **const** void *event*)

send a plain event to peer

"plain event" is event without pointer. The 4 words can be send to peer directly. *event* should be osiEvent_t.

Most likely, it won't be called directly by application. It will be called only in RPC daemon and RPC stubs.

Parameters

- `ch`: the RPC channel
- `event`: the event to be send

void **rpcSendPointerEvent**(*rpcChannel_t* *ch, **const** void *event, uint32_t *ptr1_size*, uint32_t *ptr2_size*)

send an event with pointer to peer

"pointer event" is event with pointer. The content of the pointer shall be sent to peer rather than the pointer itself. Also, the pointer should be freed after the event is sent to peer (not after peer handled the event).

When par1 or par2 is not a pointer, or the pointer is NULL, the parameter ptr1_size or ptr2_size must be zero.

Though it is possible to get the allocated size of pointer, and event the manual packer will use this method, it is not recommended programming style here (can be regarded as a hack).

Most likely, it won't be called directly by application. It will be called only in RPC daemon and RPC stubs.

Parameters

- `ch`: the RPC channel
- `event`: the event to be send
- `ptr1_size`: the memory size when par1 is a pointer, otherwise 0
- `ptr2_size`: the memory size when par1 is a pointer, otherwise 0

void **rpcRouteEvent**(*rpcChannel_t* *ch, *rpcEventHeader_t* *event)

route the event from peer

Search registered router, and call all matched routers.

Most likely, it won't be called directly by application. It will be called only in RPC daemon and RPC stubs (such as customized router).

Parameters

- `ch`: the RPC channel
- `event`: the unpacked event

bool **rpcUnpackPointerEvent**(*rpcEventHeader_t* *event, uint32_t *ptr1_size*, uint32_t *ptr2_size*)

unpack pointer event

Unpack pointer event from peer. The event itself is "packed" event. The pointer content will be copied to local malloc memory.

Most likely, it won't be called directly by application. It will be called only in RPC daemon and RPC stubs.

Return -true success -false unpack fail, may due to incorrect event header, or malloc failed.

Parameters

- `event`: the event to be send
- `ptr1_size`: the memory size when par1 is a pointer, otherwise 0

- `ptr2_size`: the memory size when par1 is a pointer, otherwise 0

struct rpcHeader_t
#include <rpc_daemon.h> RPC packet header

Public Members

uint32_t **opcode**
operation in RPC packet

uint32_t **size**
the whole RPC packet size

struct rpcCallHeader_t
#include <rpc_daemon.h> RPC call packet header

Public Members

rpcHeader_t **h**
common RPC packet header

uint32_t **api_tag**
tag for RPC function

uint32_t **caller_sync**
sync primitive of caller, usually is a semaphore

uint32_t **caller_rsp_ptr**
response pointer of caller

uint16_t **seq**
sequence number, just for debug

uint16_t **rsp_size**
response packet size

struct rpcRespHeader_t
#include <rpc_daemon.h> RPC response packet header

Public Members

rpcHeader_t **h**
common RPC packet header

uint32_t **api_tag**
tag for RPC function, copied from “call”, just for debug

uint32_t **caller_sync**
sync primitive of caller, copied from “call”

uint32_t **caller_rsp_ptr**
response pointer of caller

uint16_t **seq**
sequence number, just for debug

uint16_t **rpc_error_code**
0 or ENOENT(2)

struct rpcEventHeader_t
#include <*rpc_daemon.h*> RPC event packet header

The 4 words after common RPC packet header is just osiEvent_t. It is to reduce header file dependency t expand them.

Public Members

rpcHeader_t **h**
common RPC packet header

uint32_t **id**
event id

uint32_t **par1**
1st word parameter

uint32_t **par2**
2nd word parameter

uint32_t **par3**
3rd word parameter

CHAPTER

FIFTEEN

AT RECEIVER ENGINE

Contents

15.1 Overview

AT engine is designed as one-thread, with event driven. Also, it will support multiple interfaces, multiple channels and multiple SIM.

device/interface UART, USB serial are typical interfaces. Also, it is possible to implement other kinds of interface, by implement `atDevice_t` APIs.

channel Considering *CMUX*, even on one interface, there are several DLC. Channel is DLC in CMUX mode, or interface not in CMUX mode. For AT programming, there are no differences.

Each channel can be in one of the 3 modes:

- command mode: data parsing will follow AT command syntax.
- data mode: PPP will enter data mode. Data parsing will follow PPP syntax (HDLC packets).
- CMUX mode: data parsing will follow CMUX syntax, either basic option, or advanced option.

command channel mode When a channel is in command mode, there are 3 sub-modes:

- command line mode: data parsing will try to parse command line.
- prompt mode: the mode defined in *CMGS* and *CMGW*.
- bypass mode: all data will be sent to registered bypass callback.

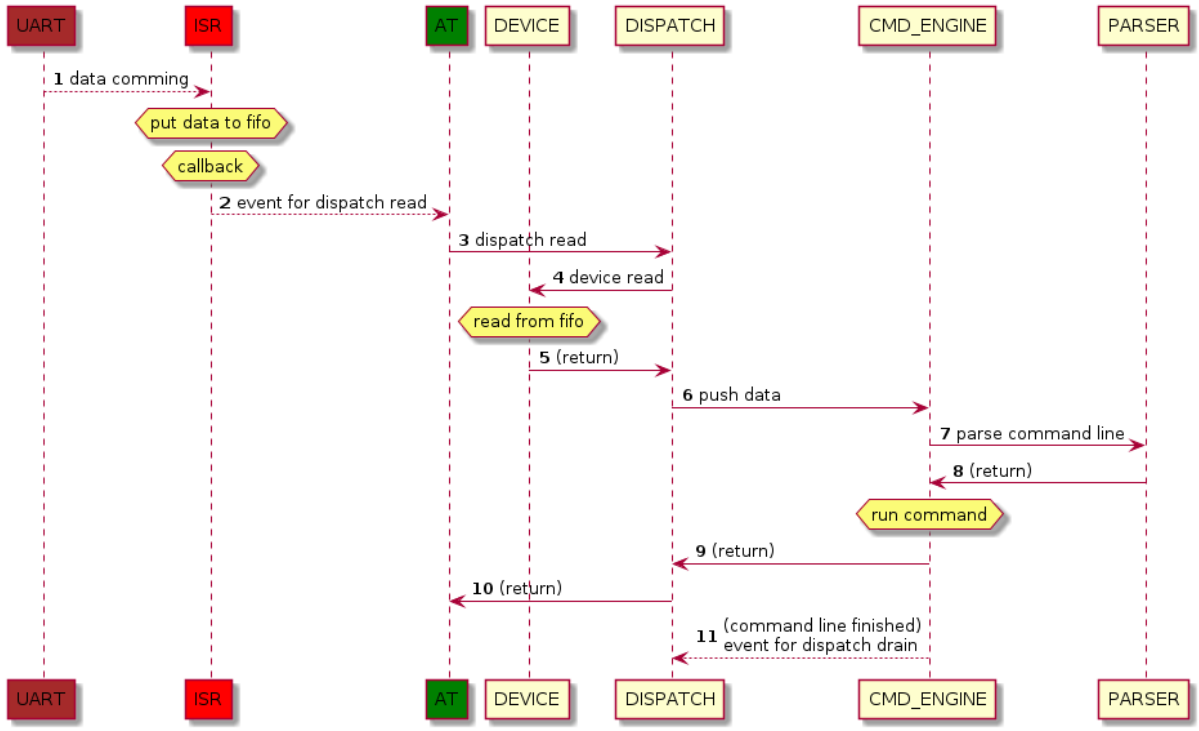
dispatch Each channel will have a *dispatch*, it manages channel mode switch, and data flow to and from the corresponding engine.

engine For each mode of a channel, there is an engine. So, there are 3 engines:

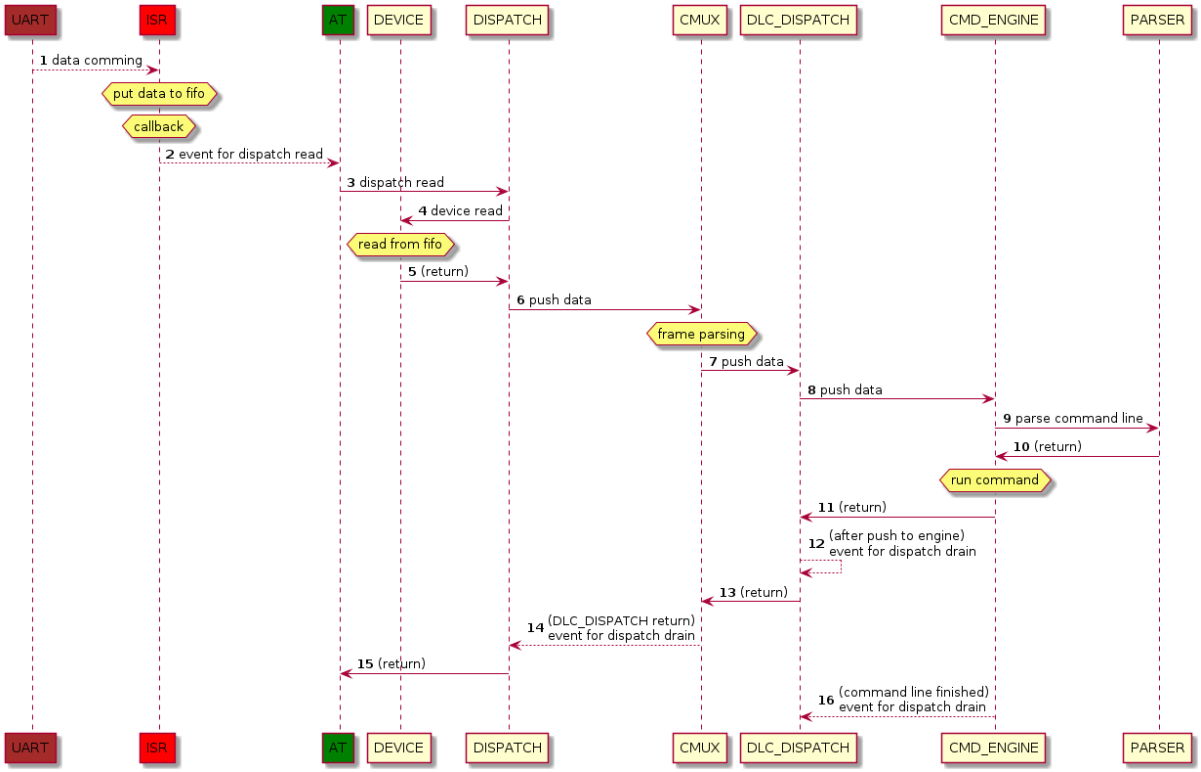
- command engine
- data engine
- cmux engine

15.2 AT Engine Process Flow

A typical AT command engine processing flow:



A typical AT command engine processing flow for CMUX DLC:



15.3 AT Settings

AT settings are packed with *nanopb*, and stored as file on file system. So, it won't break compatibility to add and remove setting items, only if the ID of each item in *proto* file is unique.

There are several setting files on file system:

/nvm/at_cfg.nv AT global settings.

/nvm/at_profile0.nv /nvm/at_profile1.nv AT profile settings. Multiple profiles are supported. The profile count can be configured by `CONFIG_ATR_PROFILE_COUNT`, and the default is 2.

/nvm/at_autosave.nv Auto-save settings. For some AT commands, the corresponding should be saved automatically when the command is executed.

/nvm/at_tcpip.nv TCPIP settings.

15.4 AT Command Line Parsing

The unit for AT command parsing is AT command line, rather than single AT command:

```
AT<one_or_more_commands>\r\n
A/
```

When a command line is parsed correctly, the command line will be parsed into several commands. And AT command engine will execute the commands one by one. If any command failed, the remaining commands will be ignored.

AT command line parser will only identify command name, and split the command parameters. It won't validate the parameters. So, every command handler should validate all parameters according to command definition.

There are a bunch of command parameter APIs to extract various types of parameter.

15.5 AT Command Parameter

In V.250 specification, there are only 2 types of parameters:

- integer
- string

However, there are many customized parameter type. So, AT command line parser will only *split* the parameter. During AT command handling, the handler should known the syntax and meaning of the parameters, then all proper AT parameter API to get the desired value.

`paramok` is a trick for easier programming. For AT parameter APIs, it is an inout argument. When it is false at input, parameter API will do nothing, and when the parameter failed, it will be set to false. So, when there are many parameters, we can check it after several parameter extraction calls.

For string and DTMF parameters, there are in-place parsing, such as parsing escape pattern in string. So, the caller is not needed to allocate dynamic memory for the parsed result. The side effect is the original data is changed, and other parameter API call on the same parameter may fail.

Note: It is rare to call multiple parameter APIs for one parameter. When needed, the in-place change should be considered.

15.6 AT Response

AT command response format is defined in V.250. And it will be affected by various AT settings. So, AT command handler shouldn't output free text. Rather, proper AT response API should be called.

There are 2 types of AT responses:

- information text
- result code

And there are 3 types of result codes:

- final: OK/ERROR/BUSY/NO_CARRIER/NO_ANSWER/NO_DAILTONE

- intermediate: CONNECT
- unsolicited

Final result code is integrated with AT engine state. On final result code, the current command will be finished. Also, the current command data structure will be destroyed. So, current command shouldn't be accessed after final result code is responded.

Without final result code, AT engine will be in the state of handling current command. And it won't process next command of current command line, or fetch next command line.

Unsolicited result code (URC) may or may not bind to specified AT engine. There are 3 cases:

- If the URC is the direct result or previous command, the URC will be reported to the AT engine of the previous command.
- If the URC is bind to specified SIM, URC API with SIM as parameter shall be called. The URC will be output to all channels in command state, and bind to that specified SIM.
- If the URC is not related to SIM, The URC will be output to all channels in command state.

AT response APIs will follow V.250 about prepend or append `\r\n`. The caller should just pass the content to response APIs, and the APIs shall handle `\r\n` according to AT settings.

15.7 Add an AT command

To add an AT command:

- add one line in `at_cmd_table.gperf`
- implement the handler

The mapping from AT command name text to handler is managed by *gperf*. At adding a command, one line should be added between `%%`. Examples:

```
P, atCmdHandleP          // arbitrary comment
S100, atCmdHandleS100    // arbitrary comment
+CMD, atCmdHandleCMD     // arbitrary comment
```

The AT parser will parse 3 kinds of command name:

- basic command: optional `&` and one character
- S-parameters: `S<parameter_number>`
- extended command: one of `+ * ^ % $` and one or more characters

`at_cmd_table.gperf` will be pre-processed by C/C++ preprocessor. So, C/C++ style comment is permitted.

AT command name is case insensitive.

The prototype of AT command handler:

```
void atCmdHandleCMD(atCommand_t *cmd);
```

If the command handler report final result code, this command is called synchronous. Otherwise, it is called asynchronous. For asynchronous command, the final result code **must** be reported in asynchronous callback.

15.8 AT Command Asynchronous Context

For asynchronous command handler, it is common requirement to keep information specified to that command. This is called *asynchronous context*. The pointer shall be stored in `atCommand_t::async_ctx`. When the command is finished, `atCommand_t::async_ctx_destroy` will be called to free resources. If the asynchronous context is a plain pointer, `atCommandAsyncCtxFree` can be set to `atCommand_t::async_ctx_destroy`. Otherwise, it is needed to implement the callback (dtor).

It is not recommended to destroy `atCommand_t::async_ctx` manually.

Global variable is discouraged for asynchronous command context, because it can't handle the case that the same command is issued in multiple channels simultaneously.

Use this standard programming pattern will be helpful to avoid memory leak.

15.9 AT and SIM

This SDK is always support multiple SIM. Each channel will be bind to *one and only one* SIM. And it is permitted that multiple channels are binded to the same SIM. So, when CFW API are called from *AT command*, the SIM parameter should be get from channel property.

For event handling from CFW, SIM is carried in CFW event. It should be used in further CFW API calls.

AT channel SIM binding can be in several states:

- DEFAULT: AT channel may be created before SIM recognize.
- SIM0/SIM1: AT channel is bind to one SIM.

For channels of interface, SIM binding will be in *DEFAULT* state. That is, the channel is binded to the first recognized SIM. For channels created by CMUX, it will inherit SIM property of parent channel.

Note: Onother typical usage model is to arrange CMUX channels manually. Some of them will bind to SIM0, and some others will bind to SIM1. For this case, customer should call API or send AT command to change the SIM binding manually.

15.10 Speech Call

The platforms can support at most one speech call connection, including conference call. So, AT engine will keep the global speech call state:

- AT_CHANNEL_CC_NONE
- AT_CHANNEL_CC_DIALING

- AT_CHANNEL_CC_ONLINE

When multiple channels are supported, there are some conventions:

- When there are no speech call, any channel can send *ATD* command.
- After *ATD* command is finished, further AT commands can be accepted. However, before the speech call connection is established or failed, further *ATD* can not be accepted.
- When there are extension in *ATD* command, further *ATD* can not be accepted before the extension is send through tone.
- There is a global speech call channel, the channel send *ATD* or *ATA* command. When there are speech call, further speech call related commands can only be accepted in the speech call channel. Only after the speech call is finished, other channels can accept speech call commands.
- The speech call channel can not be switched to PPP mode.

15.11 Memory Free Later

There is a memory recycler in AT thread. `atMemFreeLater` and `atMemUndoFreeLater` can be used for the memory recycler in AT thread.

The memory recycler will be emptied in the end of AT thread event loop.

15.12 AT Engine API Reference

Typedefs

typedef struct atEngine **atEngine_t**
opaque data structure of AT engine

typedef struct *atDevice* **atDevice_t**
data structure of AT device

typedef struct atDispatch **atDispatch_t**
opaque data structure of AT dispatch

typedef struct atCmdEngine **atCmdEngine_t**
opaque data structure of AT command mode engine

typedef struct atDataEngine **atDataEngine_t**
opaque data structure of AT data mode engine

typedef struct atCmuxEngine **atCmuxEngine_t**
opaque data structure of AT mux mode engine

typedef struct atCommand **atCommand_t**
data structure of AT command

typedef enum *atDeviceFormat* **atDeviceFormat_t**
@ brief AT device character framing format

Refer to V.250 +ICF. The enum value *matches* format parameter in +ICF command.

typedef enum *atDeviceParity* **atDeviceParity_t**
@ brief AT device character framing parity

Refer to V.250 +ICF. The enum value *matches* format parameter in +ICF command.

typedef enum *atDeviceRXFC* **atDeviceRXFC_t**
@ brief AT device receive flow control

Refer to V.250 +IFC. The enum value *matches* format parameter in +IFC command. Not supported options are not listed.

typedef enum *atDeviceTXFC* **atDeviceTXFC_t**
@ brief AT device transfer flow control

Refer to V.250 +IFC. The enum value *matches* format parameter in +IFC command. Not supported options are not listed.

typedef enum *atCmdPromptEndMode* **atCmdPromptEndMode_t**
enum type of command mode engine prompt finish mode

In prompt mode, BACKSPACE will be chcked. When BACKSPACE is input in the middle, the previous character will be removed from the buffer.

typedef void (***atCmdPromptCB_t**)(void *ctx, *atCmdPromptEndMode_t* end_mode, size_t size)
callback function type after prompt mode is finished

Parameters

- `ctx`: provided callback context
- `end_mode`: prompt end mode, refer to `atCmdPromptEndMode_t`
- `size`: received data size

typedef int (***atCmdBypassCB_t**)(void *ctx, **const** void *data, size_t size)
callback function type in command bypass mode

Return

- consumed bytes, it can be less than `size`
- 0 if nothing consumed

Parameters

- `ctx`: provided callback context
- `data`: received data pointer
- `size`: received data size

typedef int (***atDataBypassCB_t**)(void *ctx, **const** void *data, size_t size)
callback function type in data bypass mode

Return

- consumed bytes, it can be less than `size`
- 0 if nothing consumed

Parameters

- `ctx`: provided callback context
- `data`: received data pointer
- `size`: received data size

typedef void (***atCommandTimeoutHandler_t**) (*atCommand_t* *cmd)
AT command timeout handler function type.

Parameters

- `cmd`: current AT command in handling

typedef void (***atCommandAbortHandler_t**) (*atCommand_t* *cmd)
AT command abort handler function type.

Parameters

- `cmd`: current AT command in handling

typedef void (***atCommandAsyncCB_t**) (*atCommand_t* *cmd, **const** *osiEvent_t* *event)
callback function for async AT command

For async command, the callback shall be registered though *cfwRequestUTI* to wait CFW event with specified UTI, or through *atSetPendingIdCmd* to wait speficied event ID. When the matching event is arrived, this callback will be called (and the registration will be removed automatically).

typedef enum *atModeSwitchCause* **atModeSwitchCause_t**

Enums

enum atDeviceFormat
@ brief AT device character framing format

Refer to V.250 +ICF. The enum value *matches* format parameter in +ICF command.

Values:

AT_DEVICE_FORMAT_AUTO_DETECT
auto detect

AT_DEVICE_FORMAT_8N2
8 Data; 2 Stop

AT_DEVICE_FORMAT_811
8 Data; 1 Parity; 1 Stop

AT_DEVICE_FORMAT_8N1
8 Data; 1 Stop

AT_DEVICE_FORMAT_7N2
7 Data; 2 Stop

AT_DEVICE_FORMAT_711
7 Data; 1 Parity; 1 Stop

AT_DEVICE_FORMAT_7N1
7 Data; 1 Stop

enum atDeviceParity
@ brief AT device character framing parity

Refer to V.250 +ICF. The enum value *matches* format parameter in +ICF command.

Values:

AT_DEVICE_PARITY_ODD
Odd.

AT_DEVICE_PARITY_EVEN
Even.

AT_DEVICE_PARITY_MARK
Mark.

AT_DEVICE_PARITY_SPACE
Space.

enum atDeviceRXFC
@ brief AT device receive flow control

Refer to V.250 +IFC. The enum value *matches* format parameter in +IFC command. Not supported options are not listed.

Values:

AT_DEVICE_RXFC_NONE
None.

AT_DEVICE_RXFC_HW = 2
Circuit 133 (Ready for Receiving)

enum atDeviceTXFC
@ brief AT device transfer flow control

Refer to V.250 +IFC. The enum value *matches* format parameter in +IFC command. Not supported options are not listed.

Values:

AT_DEVICE_TXFC_NONE
None.

AT_DEVICE_TXFC_HW = 2
Circuit 106 (Clear to Send/Ready for Sending)

enum atCmdPromptEndMode
enum type of command mode engine prompt finish mode

In prompt mode, BACKSPACE will be chcked. When BACKSPACE is input in the middle, the previous character will be removed from the buffer.

Values:

AT_PROMPT_END_CTRL_Z
ended with CTRL-Z

AT_PROMPT_END_ESC
ended with ESCAPE

AT_PROMPT_END_OVERFLOW
provided buffer overflow

enum atModeSwitchCause
Values:

AT_MODE_SWITCH_DATA_START

AT_MODE_SWITCH_DATA_END

AT_MODE_SWITCH_DATA_ESCAPE

AT_MODE_SWITCH_DATA_RESUME

Functions

static void **atDeviceSetDispatch**(*atDevice_t* *th, *atDispatch_t* *recv)
set AT device dispatch

Parameters

- `th`: AT device pointer, must be valid
- `recv`: AT dispatch pointer, must be valid

static *atDispatch_t* ***atDeviceGetDispatch**(*atDevice_t* *th)
get AT device dispatch

Return AT dispatch pointer

Parameters

- `th`: AT device pointer, must be valid

atDevice_t ***atDeviceUartCreate**(*atDeviceUartConfig_t* *cfg)
create UART AT device

After create, the device is in *close* state. `atDeviceOpen` should be called before access the device.

Return

- UART AT device pointer

- NULL if out of memory

Parameters

- `cfg`: UART AT device configuration, must be valid

atDevice_t ***atDeviceUserialCreate**(uint32_t *name*)
create serial AT device

This used to create USB CDC/ACM AT device.

Return

- serial AT device pointer
- NULL if out of memory

Parameters

- `name`: serial device name, such as DRV_NAME_USRL_COM0

atDevice_t ***atDeviceDiagCreate**()
create diag at device

This used to create diag AT device

Return

- NULL fail
- other the at device

void **atDeviceDelete**(*atDevice_t* **th*)
delete the AT device

When `th` is NULL, nothing will be done.

Parameters

- `th`: AT device to be deleted

bool **atDeviceOpen**(*atDevice_t* **th*)
open the AT device for read and write

Return

- true on success

Parameters

- `th`: AT device, must be valid

void **atDeviceClose**(*atDevice_t* **th*)
close the AT device

Parameters

- `th`: AT device, must be valid

int **atDeviceWrite**(*atDevice_t* *th, **const** void *data, size_t *size*)
write data to AT device

AT device will try to write all data. When the output buffer is full, it will wait.

Usually, AT device will define a *reasonable* timeout. At timeout, the written size may be less than specified size.

When `size` is 0, nothing will be done.

Return

- written byte count
- -1 if parameter is invalid, or device error

Parameters

- `th`: AT device, must be valid
- `data`: data pointer to be written, must be valid if `size` is not zero
- `size`: data size

int **atDeviceRead**(*atDevice_t* *th, void *data, size_t *size*)
read data from AT device

It will just read from the device receive buffer. When the buffer is empty, return 0. Even the receive buffer is not empty, the return size may be less than the specified size.

When `size` is 0, nothing will be done.

Return

- read byte count
- -1 if parameter is invalid, or device error

Parameters

- `th`: AT device, must be valid
- `data`: mempry pointer for read, must be valid if `size` is not zero
- `size`: memory size

int **atDeviceReadAvail**(*atDevice_t* *th)

int **atDeviceWriteAvail**(*atDevice_t* *th)

void **atDeviceSetFormat**(*atDevice_t* *th, size_t *baud*, *atDeviceFormat_t* *format*, *atDeviceParity_t* *parity*)

bool **atDeviceSetFlowCtrl**(*atDevice_t* *th, *atDeviceRXFC_t* *rxfc*, *atDeviceTXFC_t* *txfc*)

void **atDeviceSetAutoSleep**(*atDevice_t* *th, int *timeout*)
set AT device auto sleep

Refer to `drvUartSetAutoSleep`

Parameters

- `th`: AT device, must be valid
- `timeout`: auto sleep wait time after transfer done. It can be 0 but not recommended. Negative value to disable auto sleep feature.

atDispatch_t ***atDispatchCreate**(*atDevice_t* *device*)
create a dispatch of device

Return the created dispatch pointer

Parameters

- `device`: AT device pointer, must be valid

void **atDispatchDelete**(*atDispatch_t* *th*)
delete a dispatch

When `th` is NULL, nothing will be done.

Parameters

- `th`: the dispatch pointer, must be valid

void **atDispatchRead**(*atDispatch_t* *th*)
read data from AT device

Usually, it is called (directly or through thread callback) when there are input data in AT device.

Parameters

- `th`: the dispatch pointer, must be valid

void **atDispatchReadLater**(*atDispatch_t* *th*)
send event to AT thread itself to consume more data

AT thread is working in non-blocking asynchronous mode. When downstream is ready, it is needed to notify upstream to consume more data. Event is used to avoid too many levels of function calls.

Parameters

- `th`: the dispatch pointer, must be valid

atDevice_t ***atDispatchGetDevice**(*atDispatch_t* *th*)
get AT device of the dispatch

When the dispatch is binded to CMUX DLC, the device of the parent CMUX engine will be returned.

Return the AT device pointer

Parameters

- `th`: the dispatch pointer, must be valid

atCmdEngine_t ***atDispatchGetCmdEngine** (*atDispatch_t *th*)
get the command mode engine the dispatch

Command mode engine of dispatch will be always created. This will return the command mode engine, no matter which dispatch mode is.

Return the command mode engine

Parameters

- `th`: the dispatch pointer, must be valid

atDataEngine_t ***atDispatchGetDataEngine** (*atDispatch_t *th*)
get the data mode engine the dispatch

Data mode engine of dispatch will be created when dispatch is changed to data mode. So, the returned pointer may be NULL if not in data mode.

Return the data mode engine

Parameters

- `th`: the dispatch pointer, must be valid

atCmuxEngine_t ***atDispatchGetParentCmuxEngine** (*atDispatch_t *th*)
get the paremt cmux mode engine

When the dispatch is not created for cmux DLC, it will return NULL.

Return the parent cmux mode engine

Parameters

- `th`: the dispatch pointer, must be valid

bool **atDispatchIsCmdMode** (*atDispatch_t *th*)
check whether in command mode

Return

- true if in command mode
- false if not in command mode

Parameters

- `th`: the dispatch pointer, must be valid

bool **atDispatchIsDataMode** (*atDispatch_t *th*)
check whether in data mode

Return

- true if in data mode
- false if not in data mode

Parameters

- `th`: the dispatch pointer, must be valid

void **atDispatchSetCmdMode**(*atDispatch_t* *th)
set dispatch to command mode

Parameters

- `th`: the dispatch pointer, must be valid

void **atDispatchSetDataMode**(*atDispatch_t* *th)
set dispatch to data mode

Parameters

- `th`: the dispatch pointer, must be valid

void **atDispatchEndDataMode**(*atDispatch_t* *th)
finish data mode and set dispatch to command mode

Parameters

- `th`: the dispatch pointer, must be valid

bool **atDispatchInDataEscape**(*atDispatch_t* *th)
check whether dispatch in data escape mode

Parameters

- `th`: the dispatch pointer, must be valid

void **atDispatchSetCmuxMode**(*atDispatch_t* *th, **const** *atCmuxConfig_t* *cfg)
set dispatch to cmux mode, with specified configuration

Parameters

- `th`: the dispatch pointer, must be valid
- `cfg`: cmux configuration

osiSlistHead_t ***atDispatchGetList**(void)
get the global AT dispatch list

Return the global AT dispatch list head

void **atCmdWrite** (*atCmdEngine_t* *th, **const** void *data, size_t *size*)

write data to AT command mode engine

When the parent dispatch is not in command mode, nothing will be done.

It will wait all data are written (maybe in buffer of AT device).

Though the data should be printable characters in typical cases, non-printable characters are permitted. It can be used to output binary data if needed.

Parameters

- `th`: command mode engine, must be valid
- `data`: data pointer to be write, can't be NULL is size is not 0
- `size`: data size

void **atCmdSetLineMode** (*atCmdEngine_t* *th)

set command mode engine to command line mode

Command line mode is the mode to receive AT commands.

Parameters

- `th`: command mode engine, must be valid

bool **atCmdSetPromptMode** (*atCmdEngine_t* *th, *atCmdPromptCB_t* *cb*, void *cb_ctx, void *buff, size_t *buff_size*)

set command mode engine to command line mode

Prompt mode is the mode defined in `CMGS` and `CMGW`.

Though the main purpose of this mode is for `CMGS` and `CMGW`, it can be used in other command handling, if the behavior is the same.

In prompt mode, the input won't come to command line buffer, rather the input will be written to the provided buffer.

Return

- true on success
- false on invalid parameter, the mode isn't changed

Parameters

- `th`: command mode engine, must be valid
- `cb`: callback after the prompt input is finished
- `cb_ctx`: callback context
- `buff`: buffer pointer for input data, can't be NULL
- `buff_size`: buffer size

bool **atCmdSetBypassMode**(*atCmdEngine_t* *th, *atCmdBypassCB_t* cb, void *cb_ctx)
set command mode engine to bypass mode

In bypass mode, the input will be passed as parameter of callback. Also, the input won't be echoed no matter the setting of *ATE*.

After bypass mode input is finished, `atCmdSetLineMode` should be called explicitly.

Typical usage is to receive binary data.

Return

- true on success
- false on invalid parameter

Parameters

- `th`: command mode engine, must be valid
- `cb`: callback when there are input data
- `cb_ctx`: callback context

void **atCmdDeviceSetFormatNeeded**(*atCmdEngine_t* *th)
notify AT device format is changed by AT command

When AT device format is changed, the new setting can't be applied to AT device immediately. Rather, AT device format should be changed after the command line is finished. This is for AT command handler to notify that AT device format change.

AT device format change commands:

- +IPR
- +ICF
- +IFC

Parameters

- `th`: command mode engine, must be valid

void **atCmdDeviceSetIfcNeeded**(*atCmdEngine_t* *th)

void **atCmdClearRemains**(*atCmdEngine_t* *th)
ignore remain commands in the command line

Parameters

- `th`: command mode engine, must be valid

void **atCmdFinalHandle**(*atCmdEngine_t* *engine, bool ok)
handle one command finish

Parameters

- `engine`: command mode engine, must be valid
- `ok`: whether the command is success or fail

void **`atCmdCommandFinished`**(*atCmdEngine_t* *th*)

notify one command in command line is finished

One command line may contain multiple AT commands. After each command handling is finished, this should be called. And command mode engine will schedule to handle the next command, if available.

Parameters

- `th`: command mode engine, must be valid

bool **`atCmdSetTimeoutHandler`**(*atCmdEngine_t* *th*, uint32_t *timeout*, *atCommandTimeoutHandler_t* *handler*)

set command timeout handler

Set AT command handler timeout handler. At timeout, `handler` will be called.

It is not needed to unregister the timeout handler manually. At `atCmdFinalHandle`, the timeout handler will be cleared automatically.

When `handler` is NULL, the previously registered timeout handler will be cleared, through it is not needed in typical cases.

This *must* be called in AT command handler.

Usually, the timeout handler shall do the command clean up, and response a final result code. For example:

```
static void _timeout(atCommand_t *cmd)
{
    // clean up ...
    atCmdRespCmdError(cmd->engine, cme_error_code);
}

void AT_COMMAND_HANDLE(atCommand_t *cmd)
{
    // ......
    atCmdSetTimeoutHandler(cmd->engine, ms, _timeout);
}
```

When the AT command will switch channel to data or CMUX mode, the timeout handler may be called when the channel is not in command mode. Usually, it is not wanted. And it is suggested not to set timeout handler if the command will switch mode.

Return

- true on success
- false if there are no AT command in handling

Parameters

- `th`: command mode engine, must be valid

- `timeout`: timeout in milliseconds
- `handler`: timeout handler

bool **atCmdSetAbortHandler**(*atCmdEngine_t* *th, *atCommandAbortHandler_t* handler)

set command handler abort handler

By default, when an AT command is in handling, further input will be hold. Only after the current AT command finished, further input will be parsed.

When a valid `handler` is registered, `handler` will be called on *any* input character. Usually, final result code shall be responsed in `handler` to abort the current command handling.

When command engine is not in command line mode (such as prompt mode), abort handler won't be called in further input.

It is not needed to unregister the abort handler manually. At `atCmdFinalHandle`, the abort handler will be cleared automatically.

When `handler` is NULL, the previously registered abort handler will be cleared.

Usually, the timeout handler shall do the command clean up, and response a final result code. For example:

```
static void _aborted(atCommand_t *cmd)
{
    // clean up ...
    atCmdRespCmdError(cmd->engine, cme_error_code);
}

void AT_COMMAND_HANDLE(atCommand_t *cmd)
{
    // ......
    atCmdSetAbortHandler(cmd->engine, _aborted);
}
```

Return

- true on success
- false if there are no command in handling

Parameters

- `th`: command mode engine, must be valid
- `handler`: abort handler

bool **atCmdIsLineMode**(*atCmdEngine_t* *th)

check whether in command line mode

Return

- true if in command line mode
- false if not in command line mode

Parameters

- th: command mode engine, must be valid

bool **atCmdIsPromptMode** (*atCmdEngine_t* *th)
check whether in prompt mode

Return

- true if in prompt mode
- false if not in prompt mode

Parameters

- th: command mode engine, must be valid

bool **atCmdIsBypassMode** (*atCmdEngine_t* *th)
check whether in bypass mode

Return

- true if in bypass mode
- false if not in bypass mode

Parameters

- th: command mode engine, must be valid

void **atCmdSetSim** (*atCmdEngine_t* *th, uint8_t *sim*)
set the binded SIM

Parameters

- th: command mode engine, must be valid
- sim: SIM number

uint8_t **atCmdGetSim** (*atCmdEngine_t* *th)
get the binded SIM

Return the binded SIM number

Parameters

- th: command mode engine, must be valid

atDispatch_t ***atCmdGetDispatch** (*atCmdEngine_t* *th)
get the parent dispatch

Return the parent dispatch

Parameters

- th: command mode engine, must be valid

atChannelSetting_t *`atCmdChannelSetting` (*atCmdEngine_t *th*)
get the pointer of per channel settings

It returns the per channel settings, and it can be modified directly.

Return per channel settings pointer

Parameters

- `th`: command mode engine, must be valid

bool `atCmdIsFirstInfoText` (*atCmdEngine_t *th*)
check whether it is the first infomation text

Depends on AT channel setting, there are extra characters for the first information text.

Return

- true if it is the first information text, that is there are no information text is output before.
- false if not

Parameters

- `th`: command mode engine, must be valid

void `atCmdSetFirstInfoText` (*atCmdEngine_t *th*, bool *is_first*)
set the first infomation text flag

Parameters

- `th`: command mode engine, must be valid
- `is_first`: the first information text flag

bool `atCmdListIsEmpty` (*atCmdEngine_t *th*)
check whether the command list is empty

A command line will contain multiple AT commands. This will check whether the command list is empty.

Return

- true if the command list is empty
- false if not

Parameters

- `th`: command mode engine, must be valid

uint8_t `atCmdChannelIndex` (*atCmdEngine_t *th*)
get the channel index

Channel index is an internal index for each channel. Application shouldn't assume the meaning of the channel index. Also, there are no API to find channel by index.

Return the internal channel index

Parameters

- `th`: command mode engine, must be valid

void **atDataWrite**(*atDataEngine_t* *th, **const** void *data, size_t size)
write data to data mode engine

When the parent dispatch is not in data mode, nothing will be done.

It will wait all data are written (maybe in buffer of AT device).

Parameters

- `th`: data mode engine, must be valid
- `data`: data pointer to be write, can't be NULL is size is not 0
- `size`: data size

void **atDataSetPPPMode**(*atDataEngine_t* *th, void *ppp)
set data mode engine to PPP mode

Parameters

- `th`: data mode engine, must be valid
- `ppp`: the PPP context pointer

void **atDataSetBypassMode**(*atDataEngine_t* *th, *atDataBypassCB_t* cb, void *cb_ctx)
set data mode engine to bypass mode

Return

- true on success
- false on invalid parameter

Parameters

- `th`: data mode engine, must be valid
- `cb`: callback when there are input data
- `cb_ctx`: callback context

bool **atDataIsPPPMode**(*atDataEngine_t* *th)
check whether in PPP mode

Return

- true if in PPP mode
- false if not in PPP mode

Parameters

- th: data mode engine, must be valid

bool **atDataIsBypassMode** (*atDataEngine_t *th*)
check whether in bypass mode

Return

- true if in bypass mode
- false if not in bypass mode

Parameters

- th: data mode engine, must be valid

atDispatch_t ***atDataGetDispatch** (*atDataEngine_t *th*)
get the parent dispatch

Return the parent dispatch

Parameters

- th: data mode engine, must be valid

void ***atDataEngineGetPppSession** (*atDataEngine_t *th*)
get the PPP context

Return the PPP context pointer

Parameters

- th: data mode engine, must be valid

void **atDataClearPPPSession** (*atDataEngine_t *th*)
clear the pppsession pointer

Parameters

- th: data mode engine, must be valid

atCmuxConfig_t **atCmuxGetConfig** (*atCmuxEngine_t *th*)
get the cmux mode engine current configuration

Return the cmux mode engine current configuration

Parameters

- th: cmux mode engine, must be valid

bool **atSetPendingIdCmd** (*atCommand_t *cmd*, uint32_t *id*, *atCommandAsyncCB_t handler*)
register an ID pending command

When the event with specified ID arrived, the callback will be called.

Return

- true if success
- false on error. Refer to *osiEventDispatchRegister*

Parameters

- `cmd`: the AT command context
- `id`: event ID
- `handler`: the callback to be called

void **atAlarmInit**()
init at device as an alarm owner

Parameters

- `device`: the AT device

void **atEngineSchedule**(*osiCallback_t cb*, void *cb_ctx)

void **atEngineModeSwitch**(*atModeSwitchCause_t cause*, *atDispatch_t *d*)

void **atEngineSetDeviceAutoSleep**(bool *enabled*)

void **atMemFreeLater**(void *ptr)

void **atMemUndoFreeLater**(void *ptr)

void **atEngineStart**(void)

osiThread_t ***atEngineGetThreadId**(void)

bool **atEventRegister**(uint32_t *id*, *osiEventHandler_t handler*)

bool **atEventsRegister**(uint32_t *id*, ...)

void **AT_SetAsyncTimerMux**(*atCmdEngine_t *cmd*, uint32_t *timeout*)

bool **atCmdEngineIsValid**(*atCmdEngine_t *cmd*)

struct atDevice
#include <at_engine.h> AT device data structure.

Public Members

void (***destroy**)(*atDevice_t* *th)
delete function

bool (***open**)(*atDevice_t* *th)
open function

void (***close**)(*atDevice_t* *th)
close function

int (*`write`) (*atDevice_t* *th, `const` void *data, size_t size)
write function

int (*`read`) (*atDevice_t* *th, void *data, size_t size)
read function

int (*`read_avail`) (*atDevice_t* *th)

int (*`write_avail`) (*atDevice_t* *th)

void (*`set_format`) (*atDevice_t* *th, size_t baud, *atDeviceFormat_t* format, *atDeviceParity_t* parity)
set format function

bool (*`set_flow_ctrl`) (*atDevice_t* *th, *atDeviceRXFC_t* rxfc, *atDeviceTXFC_t* txfc)
set flow control

void (*`set_auto_sleep`) (*atDevice_t* *th, int timeout)
set auto sleep timeout

atDispatch_t *`recv`
the dispatch

`struct atCmuxConfig_t`
#include <at_engine.h> CMUX configuration.

Public Members

uint8_t `transparency`
0: basic, 1: advanced

uint8_t `subset`
0: UIH, 1: UI, 2: I

uint8_t `port_speed`
transmission rate

int `max_frame_size`
maximum frame size

uint8_t `ack_timer`
acknowledgement timer in units of ten milliseconds

uint8_t `max_retrans_count`
maximum number of re-transmissions

uint8_t `resp_timer`
response timer for the multiplexer control channel in units of ten milliseconds

uint8_t `wakeup_resp_timer`
wake up response timer in seconds

uint8_t `window_size`
window size, for Advanced option with Error-Recovery Mode

struct atDeviceUartConfig_t
#include <at_engine.h> UART AT device configuration.

Public Members

uint32_t **name**
uart name, such as DRV_NAME_UART1

size_t **baud**
baud rate

atDeviceFormat_t **format**
character framing format

atDeviceParity_t **parity**
character framing parity

bool **rts_enable**
hw flow control, rts enable

bool **cts_enable**
hw flow control, cts enable

15.13 AT Parameter API Reference

Functions

uint32_t **atParamUint**(*atCmdParam_t* *param, bool *paramok)
extract uint parameter

When *paramok is false, it will do nothing. On failure, *param will be set to false on return.

Return

- uint parameter
- 0 on failed
  - *paramok is false at input
  - *param* is empty
  - *param* is not integer number

Parameters

- `param`: parameter pointer
- `paramok`: in/out parameter parsing ok flag

uint32_t **atParamDefUint**(*atCmdParam_t* *param, uint32_t *defval*, bool *paramok)
extract optional uint parameter

When *paramok is false, it will do nothing. On failure, *param will be set to false on return.

When *param* is empty, *defval* is returned. Otherwise, it is the same as *atParamUint*.

Return

- uint parameter
- 0 on failed
  - *paramok is false at input
  - *param* is not integer number

Parameters

- `param`: parameter pointer
- `defval`: default value
- `paramok`: in/out parameter parsing ok flag

uint32_t **atParamUintInRange**(*atCmdParam_t* *param, uint32_t *minval*, uint32_t *maxval*, bool *paramok)
extract uint parameter, and check range

When *paramok is false, it will do nothing. On failure, *param will be set to false on return.

Return

- uint parameter
- 0 on failed
  - *paramok is false at input
  - *param* is empty
  - *param* is not integer number
  - *param* is not in range

Parameters

- `param`: parameter pointer
- `minval`: minimum valid value, inclusive
- `maxval`: maximum valid value, inclusive
- `paramok`: in/out parameter parsing ok flag

uint32_t **atParamDefUintInRange**(*atCmdParam_t* *param, uint32_t *defval*, uint32_t *minval*, uint32_t *maxval*, bool *paramok)
extract optional uint parameter, and check range

When *paramok is false, it will do nothing. On failure, *param will be set to false on return.

When *param* is empty, *defval* is returned. Otherwise, it is the same as *atParamUintInRange*. The *defval* is not required in the range.

Return

- uint parameter
- 0 on failed
  - **paramok* is false at input
  - *param* is not integer number
  - *param* is not in range

Parameters

- `param`: parameter pointer
- `defval`: default value
- `minval`: minimum valid value, inclusive
- `maxval`: maximum valid value, inclusive
- `paramok`: in/out parameter parsing ok flag

uint32_t **`atParamUintInList`**(*atCmdParam_t* **param*, **`const`** uint32_t **list*, unsigned *count*, bool **paramok*)

extract uint parameter, and check in list

When **paramok* is false, it will do nothing. On failure, **param* will be set to false on return.

Return

- uint parameter
- 0 on failed
  - **paramok* is false at input
  - *param* is empty
  - *param* is not integer number
  - *param* is not in list

Parameters

- `param`: parameter pointer
- `list`: array of valid values
- `count`: valid value count
- `paramok`: in/out parameter parsing ok flag

uint32_t **atParamDefUintInList** (*atCmdParam_t* *param, uint32_t *defval*, **const** uint32_t *list, unsigned *count*, bool *paramok*)

extract optional uint parameter, and check in list

When *paramok* is false, it will do nothing. On failure, *param* will be set to false on return.

When *param* is empty, *defval* is returned. Otherwise, it is the same as *atParamUintInList*. The *defval* is not required in the list.

Return

- uint parameter
- 0 on failed
  - *paramok* is false at input
  - *param* is not integer number
  - *param* is not in list

Parameters

- `param`: parameter pointer
- `defval`: default value
- `list`: array of valid values
- `count`: valid value count
- `paramok`: in/out parameter parsing ok flag

int **atParamInt** (*atCmdParam_t* *param, bool *paramok*)

extract int parameter

When *paramok* is false, it will do nothing. On failure, *param* will be set to false on return.

Return

- int parameter
- 0 on failed
  - *paramok* is false at input
  - *param* is empty
  - *param* is not integer number

Parameters

- `param`: parameter pointer
- `paramok`: in/out parameter parsing ok flag

int **atParamDefInt** (*atCmdParam_t* *param, int *defval*, bool *paramok*)

extract optional int parameter

When *paramok* is false, it will do nothing. On failure, *param* will be set to false on return.

When *param* is empty, *defval* is returned. Otherwise, it is the same as *atParamUint*.

Return

- int parameter
- 0 on failed
  - **paramok* is false at input
  - *param* is not integer number

Parameters

- `param`: parameter pointer
- `defval`: default value
- `paramok`: in/out parameter parsing ok flag

int **atParamIntInRange**(*atCmdParam_t* **param*, int *minval*, int *maxval*, bool **paramok*)

extract int parameter, and check range

When **paramok* is false, it will do nothing. On failure, **param* will be set to false on return.

Return

- int parameter
- 0 on failed
  - **paramok* is false at input
  - *param* is empty
  - *param* is not integer number
  - *param* is not in range

Parameters

- `param`: parameter pointer
- `minval`: minimum valid value, inclusive
- `maxval`: maximum valid value, inclusive
- `paramok`: in/out parameter parsing ok flag

int **atParamDefIntInRange**(*atCmdParam_t* **param*, int *defval*, int *minval*, int *maxval*, bool **paramok*)

extract optional int parameter, and check range

When **paramok* is false, it will do nothing. On failure, **param* will be set to false on return.

When *param* is empty, *defval* is returned. Otherwise, it is the same as *atParamUintInRange*. The *defval* is not required in the range.

Return

- int parameter
- 0 on failed
  - **paramok* is false at input
  - *param* is not integer number
  - *param* is not in range

Parameters

- `param`: parameter pointer
- `defval`: default value
- `minval`: minimum valid value, inclusive
- `maxval`: maximum valid value, inclusive
- `paramok`: in/out parameter parsing ok flag

int **atParamIntInList**(*atCmdParam_t* *param, **const** int *list, unsigned *count*, bool *paramok)

extract int parameter, and check in list

When **paramok* is false, it will do nothing. On failure, **param* will be set to false on return.

Return

- int parameter
- 0 on failed
  - **paramok* is false at input
  - *param* is empty
  - *param* is not integer number
  - *param* is not in list

Parameters

- `param`: parameter pointer
- `list`: array of valid values
- `count`: valid value count
- `paramok`: in/out parameter parsing ok flag

int **atParamDefIntInList**(*atCmdParam_t* *param, int *defval*, **const** int *list, unsigned *count*, bool *paramok)

extract optional int parameter, and check in list

When **paramok* is false, it will do nothing. On failure, **param* will be set to false on return.

When *param* is empty, *defval* is returned. Otherwise, it is the same as *atParamUintInList*. The *defval* is not required in the list.

Return

- int parameter
- 0 on failed
  - **paramok* is false at input
  - *param* is not integer number
  - *param* is not in list

Parameters

- `param`: parameter pointer
- `defval`: default value
- `list`: array of valid values
- `count`: valid value count
- `paramok`: in/out parameter parsing ok flag

const char ***atParamStr** (*atCmdParam_t* **param*, bool **paramok*)
extract string parameter

When **paramok* is false, it will do nothing. On failure, **param* will be set to false on return.

String parameter is parameter started and ended with double quotation. The escape defined in V.250 is \HH. For example, \30 for '0'.

It is possible that the internal storage will be changed during this function call. So, don't call other parameter APIs for the same parameter after it is called. However, this function itself can be called again.

The output pointer will be valid till the parameter itself is deleted.

The output string is ended with null byte for end of string.

Return

- parsed string
- NULL on failed
  - **paramok* is false at input
  - *param* is empty
  - *param* is not string (start and end with double quotation)

Parameters

- `param`: parameter pointer
- `paramok`: in/out parameter parsing ok flag

const char ***atParamOptStr** (*atCmdParam_t* **param*, bool **paramok*)
extract string parameter

When **paramok* is false, it will do nothing. On failure, **param* will be set to false on return.

String parameter is parameter started and ended with double quotation. The escape defined in V.250 is \HH. For example, \30 for '0'.

It is possible that the internal storage will be changed during this function call. So, don't call other parameter APIs for the same parameter after it is called. However, this function itself can be called again.

The output pointer will be valid till the parameter itself is deleted.

The output string is ended with null byte for end of string.

Return

- parsed string
- NULL on failed
  - **paramok* is false at input
  - *param* is empty
  - *param* is not string (start and end with double quotation)

Parameters

- `param`: parameter pointer
- `paramok`: in/out parameter parsing ok flag

const char ***atParamDefStr** (*atCmdParam_t* **param*, **const** char **defval*, bool **paramok*)
extract optional string parameter

When **paramok* is false, it will do nothing. On failure, **param* will be set to false on return.

When *param* is empty, *defval* is returned. Otherwise, it is the same as *atParamStr*.

It is possible that the internal storage will be changed during this function call. So, don't call other parameter APIs for the same parameter after it is called. However, this function itself can be called again.

Return

- parsed string
- NULL on failed
  - **paramok* is false at input
  - *param* is not string (start and end with double quotation)

Parameters

- `param`: parameter pointer
- `defval`: default value
- `paramok`: in/out parameter parsing ok flag

const char ***atParamRawText** (*atCmdParam_t* **param*, bool **paramok*)
extract raw parameter

When **paramok* is false, it will do nothing. On failure, **param* will be set to false on return.

The raw parameter is not parsed. For example, the double quotation of string parameter will be kept, and the escape in string parameter is untouched.

The output string is ended with null byte as end of string.

Return

- parsed string
- NULL on failed
  - **paramok* is false at input
  - *param* is empty

Parameters

- `param`: parameter pointer
- `paramok`: in/out parameter parsing ok flag

double **`atParamDouble`** (*atCmdParam_t* **param*, bool **paramok*)
extract floating point parameter

When **paramok* is false, it will do nothing. On failure, **param* will be set to false on return.

Return

- floating point parameter
- NULL on failed
  - **paramok* is false at input
  - *param* is empty
  - *param* is not double

Parameters

- `param`: parameter pointer
- `paramok`: in/out parameter parsing ok flag

`const` char ***`atParamDtmf`** (*atCmdParam_t* **param*, bool **paramok*)
extract DTMF parameter

When **paramok* is false, it will do nothing. On failure, **param* will be set to false on return.

There are 2 kinds of DTMF parameter:

- one DTMF character without double quotation
- multiple DTMF chaeacters with double quotation In both case, the return value is DTMF character string with null byte as the string end.

It will not check whether DTMF characters inside the string is valid DTMF character. Application should validate them.

It is possible that the internal storage will be changed during this function call. So, don't call other parameter APIs for the same parameter after it is called. However, this function itself can be called again.

Return

- parsed DTMF string
- NULL on failed
  - **paramok* is false at input
  - *param* is empty
  - *param* is not DTMF

Parameters

- `param`: parameter pointer
- `paramok`: in/out parameter parsing ok flag

uint32_t **atParamUintByStrMap**(*atCmdParam_t* **param*, **const** *osiValueStrMap_t* **vsmap*, bool **paramok*)

extract string parameter and return mapped uint

When **paramok* is false, it will do nothing. On failure, **param* will be set to false on return.

It is just a wrapper to:

- extract string parameter
- map to uint

Return

- uint parameter
- NULL on failed
  - **paramok* is false at input
  - *param* is empty
  - *param* is not in map

Parameters

- `param`: parameter pointer
- `vsmap`: integer/string map, ended with NULL string value
- `paramok`: in/out parameter parsing ok flag

uint32_t **atParamDefUintByStrMap**(*atCmdParam_t* **param*, uint32_t *defval*, **const** *osiValueStrMap_t* **vsmap*, bool **paramok*)

extract optional string parameter and return mapped uint

When **paramok* is false, it will do nothing. On failure, **param* will be set to false on return.

When *param* is empty, *defval* is returned. Otherwise, it is the same as *atParamUintByStrMap*. The *defval* is not required in the map.

Return

- uint parameter
- NULL on failed
  - **paramok* is false at input
  - *param* is not in map

Parameters

- `param`: parameter pointer
- `defval`: default value
- `vsmap`: integer/string map, ended with NULL string value
- `paramok`: in/out parameter parsing ok flag

uint32_t **atParamHexStrUint** (*atCmdParam_t* **param*, bool **paramok*)

extract uint parameter by hex string

When **paramok* is false, it will do nothing. On failure, **param* will be set to false on return.

The valid parameter is the output of:

```
printf("\"%x\"", value)
```

For example, parameter "12345" will be parsed as 0x12345.

Return

- uint parameter
- NULL on failed
  - **paramok* is false at input
  - *param* is empty
  - *param* is not the needed format

Parameters

- `param`: parameter pointer
- `paramok`: in/out parameter parsing ok flag

bool **atParamTrimTail** (*atCmdParam_t* **param*, uint32_t *len*)

trim tailing characters at the end of parameter

This shall be called before the parameter isn't parsed. `atParamRawText` won't parse the parameter, so it is safe to call it after `atParamRawText`.

It will change the internal storage of parameter directly. So, only call it when absolutely needed, and you know what you are doing.

Return

- true on success
- false on failed
  - `param` type is not raw text
  - the length of parameter is shorter than `len`

Parameters

- `param`: parameter pointer
- `paramok`: in/out parameter parsing ok flag

bool **atParamIsEmpty**(*atCmdParam_t* *param)

check whether the parameter is empty

Both not specified and skipped parameter are empty. For example:

- AT+CMD=123
- AT+CMD=123,
- AT+CMD=123,,456

In all of the above cases, params[1] is empty.

Return

- true if the parameter is empty
- false otherwise

Parameters

- `param`: parameter pointer

struct atCmdParam_t

#include <*at_param.h*> parsed AT command parameter

Public Members

uint8_t **type**

parameter type, used by AT engine internally

uint16_t **length**

value length

char **value**[1]

value, the real size is variable

15.14 AT Response API Reference

Functions

void **atCmdRespInfoText** (*atCmdEngine_t* *engine, **const** char *text)
response info text

Event info text is empty, it will be regarded as one line of info text. And then \r\n will be output still.

Parameters

- engine: AT command engine, can't be NULL
- text: info text, can't be NULL

void **atCmdRespInfoNText** (*atCmdEngine_t* *engine, **const** char *text, size_t *length*)
response info text with length

Parameters

- engine: AT command engine, can't be NULL
- text: info text, can't be NULL if length is not zero
- length: info text length

void **atCmdRespOK** (*atCmdEngine_t* *engine)
response OK

This is final result. When it is called, AT engine will finish current command. So, don't access current command pointer after this is called.

Parameters

- engine: AT command engine, can't be NULL

void **atCmdRespError** (*atCmdEngine_t* *engine)
response ERROR (code 4)

This is final result. When it is called, AT engine will finish current command. So, don't access current command pointer after this is called.

Parameters

- engine: AT command engine, can't be NULL

void **atCmdRespErrorCode** (*atCmdEngine_t* *engine, int *code*)
response error with specified code

This is final result. When it is called, AT engine will finish current command. So, don't access current command pointer after this is called.

Parameters

- `engine`: AT command engine, can't be NULL
- `code`: error code

void **atCmdRespIntermCode**(*atCmdEngine_t* *engine, int code)
response intermediate code

Parameters

- `engine`: AT command engine
- `code`: intermediate code

void **atCmdRespFinish**(*atCmdEngine_t* *engine)
response finish the command without response

Parameters

- `engine`: AT command engine

void **atCmdRespUrcCode**(*atCmdEngine_t* *engine, int code)
response URC code

Parameters

- `engine`: AT command engine, can't be NULL
- `code`: URC code

void **atCmdRespCmeError**(*atCmdEngine_t* *engine, int errcode)
response CME error

This is final result. When it is called, AT engine will finish current command. So, don't access current command pointer after this is called.

Parameters

- `engine`: AT command engine, can't be NULL
- `errcode`: CME error code

void **atCmdRespCmsError**(*atCmdEngine_t* *engine, int errcode)
response CMS error

This is final result. When it is called, AT engine will finish current command. So, don't access current command pointer after this is called.

Parameters

- `engine`: AT command engine
- `errcode`: CMS error code

void **atCmdRespUrcText** (*atCmdEngine_t* *engine, **const** char *text)
response URC text to specified engine

Parameters

- `engine`: AT command engine, can't be NULL
- `text`: URC text, can't be NULL

void **atCmdRespUrcNText** (*atCmdEngine_t* *engine, **const** char *text, size_t length)
response URC text with length to specified engine

Parameters

- `engine`: AT command engine, can't be NULL
- `text`: URC text, can't be NULL if length is not zero
- `length`: URC text length

void **atCmdRespIntermText** (*atCmdEngine_t* *engine, **const** char *text)
response intermediate text

Parameters

- `engine`: AT command engine, can't be NULL
- `text`: intermediate text, can't be NULL

void **atCmdRespDefUrcCode** (int code)
response URC code to default engine(s)

This is called to report event not related to specifed channel. And it will be output to all channels in command state.

Parameters

- `code`: URC code

void **atCmdRespDefUrcText** (**const** char *text)
response URC text to default engine(s)

This is called to report event not related to specifed channel. And it will be output to all channels in command state.

Parameters

- `text`: URC text, can't be NULL

void **atCmdRespDefUrcNText** (**const** char *text, size_t length)
response URC text with length to default engine(s)

This is called to report event not related to specifed channel. And it will be output to all channels in command state.

Parameters

- `text`: URC text, can't be NULL if length is not zero
- `length`: URC text length

void **atCmdRespSimUrcCode** (uint8_t *sim*, int *code*)
response SIM related URC code

This is called to report event not related to specifed channel, but related to SIM. And it will be output to all channels in command state, and bind to the SIM.

Parameters

- `sim`: SIM number
- `code`: URC code

void **atCmdRespSimUrcText** (uint8_t *sim*, **const** char **text*)
response SIM related URC text

This is called to report event not related to specifed channel, but related to SIM. And it will be output to all channels in command state, and bind to the SIM.

Parameters

- `sim`: SIM number
- `text`: URC text, can't be NULL

void **atCmdRespSimUrcNText** (uint8_t *sim*, **const** char **text*, size_t *length*)
response SIM related URC text with length

This is called to report event not related to specifed channel, but related to SIM. And it will be output to all channels in command state, and bind to the SIM.

Parameters

- `sim`: SIM number
- `text`: URC text, can't be NULL if length is not zero
- `length`: URC text length

void **atCmdRespInfoTextBegin** (*atCmdEngine_t* **engine*, **const** char **text*)
response info text, start of line

When it is hard to combine all data of one info text line, the following sequence can be used:

- `atCmdRespInfoTextBegin`
- `atCmdRespOutputText`, multiple calls are allowed
- `atCmdRespInfoTextEnd`

Parameters

- `engine`: AT command engine, can't be NULL
- `text`: output text, can't be NULL

void **atCmdRespInfoNTextBegin** (*atCmdEngine_t* *engine, **const** char *text, size_t length)
response info text with length, start of line

Parameters

- `engine`: AT command engine, can't be NULL
- `text`: output text, can't be NULL if length is non zero
- `length`: output text length

void **atCmdRespInfoTextEnd** (*atCmdEngine_t* *engine, **const** char *text)
response info text, end of line

When it is hard to combine all data of one info text line, the following sequence can be used:

- `atCmdRespInfoTextBegin`
- `atCmdRespOutputText`, multiple calls are allowed
- `atCmdRespInfoTextEnd`

Parameters

- `engine`: AT command engine, can't be NULL
- `text`: output text, can't be NULL

void **atCmdRespInfoNTextEnd** (*atCmdEngine_t* *engine, **const** char *text, size_t length)
response info text with length, end of line

Parameters

- `engine`: AT command engine, can't be NULL
- `text`: output text, can't be NULL if length is non zero
- `length`: output text length

void **atCmdRespOutputText** (*atCmdEngine_t* *engine, **const** char *text)
output text

When it is hard to combine all data of one info text line, the following sequence can be used:

- `atCmdRespInfoTextBegin`
- `atCmdRespOutputText`, multiple calls are allowed
- `atCmdRespInfoTextEnd`

Though it is possible to output arbitrary text with this function, and AT engine won't prepend or append `\r\n`, it is not recommended. V.250 has detailed requirement for all kinds of response.

Parameters

- `engine`: AT command engine, can't be NULL
- `text`: output text, can't be NULL

void **atCmdRespOutputNText** (*atCmdEngine_t* *engine, **const** char *text, size_t length)
output text with length

Parameters

- `engine`: AT command engine, can't be NULL
- `text`: output text, can't be NULL if length is non zero
- `length`: output text length

void **atCmdRespOutputPrompt** (*atCmdEngine_t* *engine)
output prompt

The prompt is `\r\n> `.

Parameters

- `engine`: AT command engine, can't be NULL

void **atCmdRespOKText** (*atCmdEngine_t* *engine, **const** char *text)
response OK with non-standard text

This is final result. When it is called, AT engine will finish current command. So, don't access current command pointer after this is called.

Parameters

- `engine`: AT command engine, can't be NULL
- `text`: non-standard OK text, can't be NULL

void **atCmdRespErrorText** (*atCmdEngine_t* *engine, **const** char *text)
response ERROR with non-standard text

This is final result. When it is called, AT engine will finish current command. So, don't access current command pointer after this is called.

Parameters

- `engine`: AT command engine, can't be NULL
- `text`: non-standard OK text, can't be NULL

void **atCmdRingInd** (uint8_t *sim*)
ring indicate

It shall be called when there is an incoming call. The detailed output is affected by various AT settings.

Due to incoming call is bind to one SIM, the output will be send to all channels in command state, and bind with the SIM.

Parameters

- `sim`: SIM number

CHAPTER

SIXTEEN

FIRMWARE UPDATE

Contents

This SDK provides the feature of differential firmware update. Separated PC tools will be provided to generate differential pack between 2 versions. The firmware update feature will be able to update current version to a new version based on the differential pack.

When generating differential pack, current version running on target is called *old version*. And the version which will be updated to is called *new version*. For differential firmware update, current version will be used at applying differential pack. So, it is necessary that the current version running on target must *exactly* match the *old version* used at generating differential pack.

Inside `fupdateSetReady`, current version will be thoroughly checked with the information embedded in differential pack. It is not a simple version string comparison, rather the content checksum will be checked. When the parameter of `fupdateSetReady` or version string doesn't exist in update pack, the version string check will be skipped.

Differential firmware update will be performed in bootloader. So, firmware update will be integrated into both of bootloader and application.

16.1 Firmware Update in Application

It is suggested to check firmware update status at the beginning of boot. For example:

```
if (fupdateGetStatus == FUPDATE_STATUS_FINISHED) {
    char *old_version;
```

(continues on next page)

(continued from previous page)

```
    char *new_version;
    if (fupdateGetVersion(&old_version, &new_version)) {
        OSI_LOGXI(OSI_LOGPAR_SS, 0, "FUPDATE %s -> %s", old_version, new_version);
        free(old_version);
        free(new_version);
    }
    fupdateInvalidate(true);
}
```

At firmware update:

```
// get differential pack in any method
vfs_file_write(gFupdatePackFileName, pack_data, pack_size);
if (fupdateSetReady(curr_version)) {
    OSI_LOGI(0, "start firmware update");
    osiShutdown(OSI_SHUTDOWN_RESET);
} else {
    OSI_LOGI(0, "invalid FUPDATE pack !!");
}
```

16.2 Firmware Update in Bootloader

There is only one API to integrate firmware update in bootloader:

```
fupdateResult_t result = fupdateRun();
if (result == FUPDATE_RESULT_FAILED)
    reboot_and_try_again();
```

16.3 FUPDATE_RESULT_CANNT_START

Ideally, this shouldn't happen. Typical reason is that there are issues in the created update pack, or system power supply is unstable during bootloader check. In this case, the existed content is untouched and it is safe for normal boot.

16.4 Files for Firmware Update

There are 3 files related to firmware update:

- gFupdatePackFileName: differential pack data
- gFupdateStageFileName: keep record of firmware update stage
- gFupdateTempFileName: temporal file used during firmware update

`gFupdatePackFileName` can be read and write with normal vfs APIs. However, never access the other 2 files directly.

These 3 variable are defined with default value inside firmware update module, as weak symbol. They can be override. And it is necessary to keep them identical between application and bootloader.

16.5 API Reference

Typedefs

typedef enum *fupdateStatus* **fupdateStatus_t**
firmware update status

typedef void (***fupdateProgressCallback_t**) (int block_count, int block)
progress callback function type

During firmware update in bootloader, this callback will be called after each blocked is processed.

Parameters

- `block_count`: total block count
- `block`: current finished block

typedef enum *fupdateResult* **fupdateResult_t**
firmware update result

Enums

enum fupdateStatus
firmware update status

Values:

FUPDATE_STATUS_NOT_READY
not ready

FUPDATE_STATUS_READY
ready

FUPDATE_STATUS_FINISHED
update finished

enum fupdateResult
firmware update result

Values:

FUPDATE_RESULT_NOT_READY
not ready

FUPDATE_RESULT_CANNT_START
failed to start

FUPDATE_RESULT_FAILED
started and failed

FUPDATE_RESULT_FINISHED
finished

Functions

fupdateStatus_t **fupdateGetStatus** (void)
get firmware update status

The status is just get from update status file. And it won't perform thorough update pack file content check.

Return update status

bool **fupdateGetVersion** (char **_old_version_, char **_new_version_)
get firmware update version information

The old version and new version are embedded inside update pack.

The version strings are specified at update pack creation. It is suggestion just use them as trace or report information.

`old_version` and `new_version` are allocated inside. Caller should free them after used. When there are no version string inside update pack, the returned pointer will be NULL.

Return

- true on success
- false on error
  - out of memory, or invalid parameter
  - status is `FUPDATE_STATUS_NONE`

Parameters

- `old_version`: return pointer for old version
- `new_version`: return pointer for new version

void **fupdateInvalidate** (bool *removePack*)
invalidate firmware update

Invalidate firmware update status. `gFupdateStageFileName` will be removed, `gFupdateTempFileName` will be removed, and `gFupdatePackFileName` can be removed optionally.

After previous update is finished, it is recommended to call this, and remove update pack file, to release spaces in file system.

It will be ensured that `fupdateGetStatus` will return `FUPDATE_STATUS_NONE` after this is called.

Parameters

- `removePack`: true to remove update pack file

bool **`fupdateIsPackValid`**(**`const`** char *_curr_version_)
check whether firmware update file is valid

It will perform thorough update pack file content. So, it will take times.

When `curr_version` is not NULL, it will try to check current version with old version string embedded in update pack. When old version string in update pack is not NULL also, and `curr_version` doesn't match old version in update pack, the update pack will be regarded as invalid.

It is optional to embed version string inside update pack.

Return

- true if the update pack file content is valid
- false if not

Parameters

- `curr_version`: current version string, NULL for not to check

bool **`fupdateSetReady`**(**`const`** char *_curr_version_)
set firmware update to ready status

Inside this function, the update pack file will be thoroughly checked.

When the update pack file is valid:

- `gFupdateStageFileName` will be updated to indicate ready status;
- `gFupdateTempFileName` will be removed;

When the update pack file is invalid:

- `gFupdateStageFileName` will be removed;
- `gFupdateTempFileName` will be removed;

It must be called after the update pack file is written and closed. If the pack file is modified after `fupdateSetReady` is called, bootloader will still try to update. Though bootloader can detected update pack invalid and not start real update, it will waste boot time.

No matter whether `fupdateIsPackValid` is called, the update pack file will be verified inside. So, it will take even longer.

When `curr_version` is not NULL, it will try to check current version with old version string embedded in update pack. When old version string in update pack is not NULL also, and `curr_version` doesn't match old version in update pack, the update pack will be regarded as invalid.

It is optional to embed version string inside update pack.

Return

- true if update pack file is valid

- false if not

Parameters

- `curr_version`: current version string, NULL for not to check

fupdateResult_t **fupdateRun** (*fupdateProgressCallback_t progress*)

perform firmware update

It must be called in bootloader. Before `fupdateRun` is called, file system should be initializaed.

- file system for `CONFIG_FS_FOTA_DATA_DIR`
- file system to be patched

File system layout and mount points must be the same as application.

`FUPDATE_RESULT_NOT_READY` means that `gFupdateStageFileName` doesn't exists, or not indicates `FUPDATE_STATUS_READY`. Bootloader shall boot application normally.

`FUPDATE_RESULT_CANNT_START` means that `gFupdateStageFileName` indicates `FUPDATE_STATUS_READY`, however the update pack is not valid or not enough memory can be allocated for upgraded. It shouldn't happen, but the current firmware isn't changed. Bootloader can boot application normally. And it is suggested to notify application to invalidate update status.

`FUPDATE_RESULT_FAILED` means that current firmware is changed, and encounter unrecoverable error. Typical case is that power supply is unstable. Due to current firmware is changed, bootloader shall **NEVER** boot application normally. It is suggested that bootloader shall reboot and try again.

`FUPDATE_RESULT_FINISHED` means that current firmware is successfully updated. Bootloader can boot application normally, or even better reboot. At reboot, `fupdateRun` will return `FUPDATE_RESULT_NOT_READY`.

When `progress` callback is not NULL, it will be called after each block is finished.

Parameters

- `progress`: progress callback

Variables

const char ***gFupdatePackFileName**

firmware update pack file name

The default value is `FUPDATE_PACK_FILE_NAME`. And it can be defined as other value to override the default value. When overrided, it should be the same as bootloader.

const char ***gFupdateStageFileName**

firmware update stage file name

The default value is `FUPDATE_STAGE_FILE_NAME`. And it can be defined as other value to override the default value. When overrided, it should be the same as bootloader.

const char ***gFupdateTempFileName**

firmware update temporal file name

The default value is FUPDATE_TEMP_FILE_NAME. And it can be defined as other value to override the default value. When overrided, it should be the same as bootloader.

CHAPTER

SEVENTEEN

IOMUX

Contents

17.1 Overview

It is common cases that some pins can be configured as several functions. This module provides unified API to set pin function.

The simplest pin to function map is *1:N*. That is one pin can be configured as several functions, and each function can be configured to only one pin. And this module assumes this usage model.

More complex usage model is *M:N*. That is one pin can be configured as several functions, and each function can be configured to several pins. In this case, the usage model should be simplified.

IOMUX information is stored in a `csv` file. `tools/iomuxgen.py` will parse the file, and generate `hal_iomux_pindef.h` and `hal_iomux_pincfg.h`. `hal_iomux_pindef.h` defines all functions.

> Example of function enum is `PINFUNC_GPIO_0`. `PINFUNC_` is the enum prefix, `GPIO_0` is the function name.

`hal_iomux_pincfg.h` contains information about how to set register to enable the function.

For *M:N* model, `x--` shall be added to unneeded pin functions, to simplify it as *1:N* model.

> `PINFUNC_GPIO_<n>` should be equal to `PINFUNC_GPIO_0 + <n>`.

During simplification, some functions are removed. It is possible that the default version can't fit specific project. In that case, it is needed to change the csv, and re-generate `hal_iomux_pindef.h` and `hal_iomux_pincfg.h`.

Other pin properties, such as *pull-down* and *drive strength* aren't supported in this module. If needed, it is needed to write register directly.

17.2 API Reference

Defines

HAL_IOMUX_REQUEST_END
special value to indicate end of batch request

Functions

void **halIomuxInit**(void)
IOMUX module initialization.

void **halIomuxRequest**(unsigned *pinfunc*)
request pin function

Change IOMUX setting. When the *pinfunc* is invalid, it will be ignored silently.

Parameters

- `pinfunc`: pin function

void **halIomuxRequestBatch**(unsigned *pinfunc*, ...)
batch request pin function

Change multiple IOMUX settings. The variadic variables should be ended with *HAL_IOMUX_REQUEST_END*.

Parameters

- `pinfunc`: pin function

void **halIomuxRelease**(unsigned *pinfunc*)
reset IOMUX to default value

Change IOMUX setting to default to avoid power leakage. If current pin is not the *pinfunc*, do nothing.

Parameters

- `pinfunc`: pin function

void **halIomuxReleaseBatch**(unsigned *pinfunc*, ...)
batch reset IOMUX to default value

Reset multiple IOMUX settings. The variadic variables should be ended with *HAL_IOMUX_REQUEST_END*.

Parameters

- `pinfunc`: pin function

CHAPTER

EIGHTEEN

HARDWARE SPINLOCK

Contents

18.1 Overview

Hardware spinlock is to avoid conflicts when multiple CPUs access shared resources.

A typical example is ADI bus for PMIC. For each ADI bus read, there are several ADI bus master register access. Without protection, ADI bus access will get unexpected result.

Another example is multiple CPUs will change bits, even different bits in one hardware register. Read-modify-write is needed to change bits. Without protection, it is possible that one CPU will overwrite other CPU's write.

In 8910, there are 32 hardware spinlocks. Hardware spinlock themselves aren't attached to any shared resources in hardware. Protection is achieved by software convention. And all CPUs in the system shall follow the same convention.

Hardware spinlock is binded with critical section in software implementation. That is, when the hardware spinlock is acquired, system will enter critical section also. It can avoid to hold hardware spinlock for too long due to interrupt context switch.

Hardware spinlock APIs can be called in ISR.

18.2 API Reference

Defines

HAL_HWSPINLOCK_ID_ADIBUS
reserved HW spinlock ID for ADI bus access

HAL_HWSPINLOCK_ID_IPC
reserved HW spinlock ID for IPC

Functions

uint32_t **halHwspinlockAcquire**(uint32_t *id*)
acquire HW spinlock and enter critical

Return critical flag

Parameters

- `id`: HW spinlock id

void **halHwspinlockRelease**(uint32_t *critical*, uint32_t *id*)
release HW spinlock and exit critical

Parameters

- `critical`: critical flag
- `id`: HW spinlock id

CHAPTER

NINETEEN

GPIO

Contents

19.1 Overview

GPIO can work in 2 modes:

- input: can read the input level, and enable input interrupt
- output: can write the output level

The input interrupt can be set to edge trigger, or level trigger. Level interrupt is seldom used. Edge interrupt can be enabled at rising edge, falling edge, or both.

debounce is a hardware feature to eliminate glitch. Usually, if the GPIO is connected to a button (which will be pressed manually), *debounce* is recommended.

It is possible that not all GPIO can support input interrupt.

8910:

- There are 32 GPIOs totally.
- All GPIOs can support input interrupt.

19.2 API Reference

Typedefs

typedef struct drvGpio **drvGpio_t**
opaque data struct for GPIO instance

typedef void (***drvGpioIntrCB_t**)(void *ctx)
GPIO interrupt callback

It will be called when GPIO interrupt occurs. Due to it is executed in ISR, the callback should follow ISR programming guides.

Enums

enum drvGpioMode_t
Values:

DRV_GPIO_INPUT

DRV_GPIO_OUTPUT

Functions

void **drvGpioInit**(void)
GPIO module initialization.

It just initialize GPIO module, and won't touch any GPIO. It should be called before any *drvGpioOpen*.

drvGpio_t ***drvGpioOpen**(uint32_t *id*, *drvGpioConfig_t* **cfg*, *drvGpioIntrCB_t* *cb*, void **cb_ctx*)
open a GPIO

IOMUX will be set to GPIO mode at open.

Each GPIO sould be opened only once. When it is already opened, this API will return NULL.

If the specified GPIO can't support the specified mode, this API will return NULL.

If the specified GPIO can't support input interrupt, and interrupt is enabled, this API will return NULL.

The returned instance is dynamic allocated, caller should free it after *drvGpioClose* is called.

Return

- GPIO instance pointer
- NULL if parameter is invalid

Parameters

- `id`: GPIO id. The range may be different among chips.
- `cfg`: GPIO configuration
- `cb`: callback at interrupt
- `cb_ctx`: context pointer for callback

void **drvGpioClose**(*drvGpio_t* **d*)
close a GPIO

IOMUX will be kept, and the GPIO will be set to input mode.

Parameters

- `d`: GPIO instance

bool **drvGpioReconfig**(*drvGpio_t* *d, *drvGpioConfig_t* *cfg)
reconfigure the opened GPIO

When the configuration is invalid, the current configuration will be kept.

Return

- true if GPIO configuration is valid
- false if GPIO configuration is invalid

Parameters

- `d`: GPIO instance
- `cfg`: GPIO configuration

bool **drvGpioRead**(*drvGpio_t* *d)
read the level of GPIO

It can be called for both GPIO input and output. For GPIO output, it is the level set by software.

Return

- true for level high
- false for level low

Parameters

- `d`: GPIO instance

void **drvGpioWrite**(*drvGpio_t* *d, bool *level*)
write level for GPIO

When it is called for GPIO input, it will do nothing.

When it is called for GPIO output, set the output level.

Parameters

- `d`: GPIO instance
- `level`: GPIO level to be set,
  - true for level high
  - false for level low

struct drvGpioConfig_t
#include <*drv_gpio.h*> GPIO configuration

Public Members

drvGpioMode_t **mode**
GPIO mode.

bool **out_level**
level to be set for GPIO output

bool **intr_enabled**
interrupt enabled, only for GPIO input

bool **intr_level**
true for level interrupt, false for edge interrupt

bool **debounce**
debounce enabled

bool **rising**
rising edge or level high interrupt enabled

bool **falling**
falling edge or level low interrupt enabled

CHAPTER

TWENTY

ADI BUS FOR PMIC

Contents

20.1 Overview

20.2 API Reference

Defines

HAL_ADI_CHANGE1 (r, t, f1, v1)

HAL_ADI_BUS_OVERWITE (value)
mask indicates write rather than read-modify-write

HAL_ADI_BUS_CHANGE_END
indicates end of variadic variables of `halAdiBusBatchChange`

Functions

void **halAdiBusInit** (void)
initialize ADI bus

uint32_t **halAdiBusRead** (**volatile** uint32_t **reg*)
read register through ADI bus

Return read value

Parameters

- `reg`: register physical address

void **halAdiBusWrite** (**volatile** uint32_t *_reg_, uint32_t _value_)
read register through ADI bus

Parameters

- `reg`: register physical address
- `value`: value to be written

void **halAdiBusChange** (**volatile** uint32_t *_reg_, uint32_t _write_mask_, uint32_t _write_value_)
change register

When `write_mask` is not all 1s, the operation is read-modify-write.

```
new_value = (old_value & ~writ_mask) | (write_value & write_mask)
```

Parameters

- `reg`: register physical address
- `write_mask`: mask for change, 1 for change, 0 for no change
- `write_value`: value to be changed

void **halAdiBusBatchChange** (**volatile** uint32_t *_reg_, uint32_t _write_mask_, uint32_t _write_value_, ...)
batch change registers

Batch change several registers through ADI bus. When `write_mask` is not all 1s, the operation is read-modify-write.

```
new_value = (old_value & ~writ_mask) | (write_value & write_mask)
```

`reg` value of 0 indicates the end of variadic variables.

Parameters

- `reg`: register physical address
- `write_mask`: mask for change, 1 for change, 0 for no change
- `write_value`: value to be changed

CHAPTER

TWENTYONE

PMIC INTERRUPT

Contents

21.1 Overview

PMIC is an external chip. PMIC registers are accessed through ADI bus. All PMIC interrupts are connected to host CPU though a GPIO. So, on host CPU, all PMIC interrupts are handled in a GPIO ISR callback.

PMIC EIC (external interrupt control) is a module inside PMIC. Multiple sources are connected to EIC. All EIC interrupts will appear in PMIC interrupt controller as one interrupt.

Note: Though EIC can support both debounce mode and bypass mode, debounce mode shall be used.

All EIC interrupts are *level* interrupt, and the *level* is configurable. In debounce mode, interrupt will only appear after *manual trigger*. Usually software will just need to know interrupt of *changes*, conceptually close to *edge* interrupt. The API style of EIC interrupt is different from *edge* interrupt. Software shall call `drvPmicEicTrigger` explicitly for the opposite level.

Most likely, this module won't be called by high level application, input parameters will be checked, but may return silently on invalid parameters.

21.2 API Reference

Typedefs

typedef void (***drvPmicIntrCB_t**)(void *ctx)
callback function for PMIC interrupt

It is called in ISR, and it is needed to follow ISR programming guide.

Enums

enum drvPmicIntrType_t
PMIC interrupt types

Values:

DRV_PMIC_INTR_ADC
PMIC ADC interrupt.

DRV_PMIC_INTR_RTC
PMIC RTC interrupt.

DRV_PMIC_INTR_WDG
PMIC watchdog interrupt.

DRV_PMIC_INTR_FGU
PMIC fuel gauge unit interrupt.

DRV_PMIC_INTR_EIC
PMIC EIC interrupt.

DRV_PMIC_INTR_AUD
PMIC audio interrupt.

DRV_PMIC_INTR_TMR
PMIC timer interrupt.

DRV_PMIC_INTR_CAL
PMIC oscillator calibration interrupt.

DRV_PMIC_INTR_COUNT

enum drvPmicEicType_t
PMIC EIC interrupt types

Values:

DRV_PMIC_EIC_CHGR_INT
charge indicator

DRV_PMIC_EIC_PBINT
power on 1

DRV_PMIC_EIC_PBINT2
power on 2

DRV_PMIC_EIC_AUDIO_HEAD_BUTTON
??

DRV_PMIC_EIC_CHGR_CV
??

DRV_PMIC_EIC_AUDIO_HEAD_INSERT
??

DRV_PMIC_EIC_VCHG_OVI
??

DRV_PMIC_EIC_AUDIO_HEAD_INSERT2
??

DRV_PMIC_EIC_BATDET_OK
??

DRV_PMIC_EIC_EXT_RSTN
??

DRV_PMIC_EIC_EXT_XTL_EN0
??

DRV_PMIC_EIC_AUDIO_HEAD_INSERT3
??

DRV_PMIC_EIC_AUDIO_HEAD_INSERT_ALL
??

DRV_PMIC_EIC_EXT_XTL_EN1
??

DRV_PMIC_EIC_EXT_XTL_EN2
??

DRV_PMIC_EIC_EXT_XTL_EN3
??

DRV_PMIC_EIC_COUNT

Functions

void **drvPmicIntrInit**(void)
PMIC interrupt module initialization.

It should be called after ADI bus is initialized, and before any other modules which will use PMIC interrupt and PMIC EIC interrupts.

void **drvPmicIntrEnable**(unsigned *intr*, *drvPmicIntrCB_t* *cb*, void **ctx*)
enable PMIC interrupt and set callback

When there already exist callback for the specified PMIC interrupt, the previous one will be replaced.

Parameters

- `intr`: PMIC interrupt type (*drvPmicType_t*)
- `cb`: PMIC interrupt callback
- `ctx`: PMIC interrupt cabback context

void **drvPmicIntrDisable**(unsigned *intr*)
disable PMIC interrupt

Parameters

- `intr`: PMIC interrupt type (*drvPmicType_t*)

void **drvPmicEicSetCB**(unsigned *eic*, *drvPmicIntrCB_t cb*, void **ctx*)
set PMIC EIC interrupt callback

Set the callback of PMIC EIC intterupt. The EIC interrupt won't be started automatically.

In callback, it is permitted to call EIC APIs, such as re-trigger it in opposite polarity.

Parameters

- `eic`: PMIC EIC interrupt type (*drvPmicEicType_t*)
- `cb`: PMIC EIC interrupt callback
- `ctx`: PMIC EIC interrupt callback context

void **drvPmicEicTrigger**(unsigned *eic*, unsigned *debounce*, bool *level*)
enable PMIC EIC interrupt

After EIC interrupt arrived, the interrupt will be disabled automatically. When the interrupt is needed, *drvPmicEicTrigger* (with the same or opposite polarity).

When *debounce* is too small, it will be set to the minimum value suitable for underlay hardware. Typically, it will be several milliseconds.

Parameters

- `eic`: PMIC EIC interrupt type (*drvPmicEicType_t*)
- `debounce`: debounce time in milliseconds
- `level`: trigger level, true for high

bool **drvPmicEicGetLevel**(unsigned *eic*)
PMIC EIC source current level.

Return

- true for high level
- false for low level, or invalid type

Parameters

- `eic`: PMIC EIC interrupt type (*drvPmicEicType_t*)

void **drvPmicEicDisable**(unsigned *eic*)
disable PMIC EIC interrupt

Parameters

- `eic`: PMIC EIC interrupt type (*drvPmicEicType_t*)

CHAPTER

TWENTYTWO

SPI FLASH

Contents

22.1 Overview

SPI flash can support XIP (eXecute In Place), and this SDK will support XIP.

When XIP is used, the most important issue for SPI flash is to avoid read access when flash program or erase isn't finished. SPI flash program and erase is very slow. It is unacceptable to wait program and erase finish with interrupt disabled. Rather, when there are *important* things to do during wait program and erase finish, software will send suspend command and then flash is accessible. After system is idle, software will send resume command and then flash will continue the interrupted program or erase.

To enable this procedure, RTOS interrupt handler shall call `drvSpiFlashEraseProgramSuspend` at the beginning. The interrupt handler should be located in RAM (either PSRAM or SRAM). Location of ISR for each interrupt, including the functions called is unpredictable.

Most of the APIs won't be called by application directly. Rather, application shall high level APIs based on SPI flash, such as file system.

22.2 API Reference

Typedefs

typedef struct drvSpiFlash **drvSpiFlash_t**
opaque data structure for SPI flash

Functions

void **drvSpiFlashEarlyConfig**(uint32_t *name*)
config SPI flash in early stage of boot

It will be executed at early stage of boot, it is possible:

- RAM isn't initialized, only SRAM can be accessed;
- RTOS isn't initialized;

Parameters

- `name`: SPI flash device name

drvSpiFlash_t ***drvSpiFlashOpen**(uint32_t *name*)
open SPI flash

SPI flash instance is singleton for each flash. So, for each device name the returned instance pointer is the same. And there are no *close* API.

At the first open, the flash will be initialized (for faster speed).

Return

- SPI flash instance pointer
- NULL if the name is invalid

Parameters

- `name`: SPI flash device name

uint32_t **drvSpiFlashGetID**(*drvSpiFlash_t* **d*)
get flash ID

The contents of the ID:

- id[23:16]: capacity
- id[15:8]: memory type
- id[7:0]: manufacturer

When `d` is NULL, the default SPI flash will be used.

Return

- flash ID

Parameters

- `d`: SPI flash instance pointer, must be valid

uint32_t **drvSpiFlashCapacity**(*drvSpiFlash_t* **d*)
get flash capacity in byte

When `d` is NULL, the default SPI flash will be used.

Return

- flash capacity in byte

Parameters

- `d`: SPI flash instance pointer, must be valid

void **drvSpiFlashWriteLock** (*drvSpiFlash_t* *d)
lock (disable) flash program or erase

When there are flash writing in progress, it will wait the current operation finish.

When `d` is NULL, the default SPI flash will be used.

Parameters

- `d`: SPI flash instance pointer, must be valid

void **drvSpiFlashWriteUnlock** (*drvSpiFlash_t* *d)
unlock (enable) flash program or erase

When `d` is NULL, the default SPI flash will be used.

Parameters

- `d`: SPI flash instance pointer, must be valid

const void ***drvSpiFlashMapAddress** (*drvSpiFlash_t* *d, uint32_t *offset*)
map flash offset to accessible address

The returned pointer should be accessed as read only.

When `d` is NULL, the default SPI flash will be used.

Return

- accessible pointer
- NULL if *offset* is invalid

Parameters

- `d`: SPI flash instance pointer, must be valid
- `offset`: flash offset

uint32_t **drvSpiFlashOffset** (*drvSpiFlash_t* *d, **const** void **address*)
convert mapped address to flash offset

When the address is invalid, return -1U.

When `d` is NULL, the default SPI flash will be used.

Return

- flash offset

- -1U if *address* is not mapped flash address

Parameters

- `d`: SPI flash instance pointer, must be valid
- `address`: mapped flash accessible pointer

bool **drvSpiFlashFastRead**(*drvSpiFlash_t* *d, uint32_t *offset*, void *data, size_t *size*)
read flash by FAST_READ command

This function is for debug only. **Don't** use it in production codes.

The method to read flash is use *drvSpiFlashMapAddress* to get a pointer, and read flash by the pointer directly. Though it is named as *fast read*, it doesn't mean that it is faster than direct access by mapped pointer.

When `d` is NULL, the default SPI flash will be used.

Return

- true on success
- false on invalid parameters

Parameters

- `d`: SPI flash instance pointer, must be valid
- `offset`: flash offset
- `data`: memory for read
- `size`: read size

bool **drvSpiFlashWrite**(*drvSpiFlash_t* *d, uint32_t *offset*, **const** void *data, size_t *size*)
write to flash

This will just use flash PROGRAM command to write. Caller should ensure the write sector or block is erased before. Otherwise, the data on flash may be different from the input data.

offset and *size* are not needed to be sector or block aligned.

It will wait flash write finish. On success return, all data are written to flash.

It won't read back to verify the result.

When `d` is NULL, the default SPI flash will be used.

Return

- true on success
- false on invalid parameters

Parameters

- `d`: SPI flash instance pointer, must be valid
- `offset`: flash offset

- `data`: memory for write
- `size`: write size

bool **drvSpiFlashErase** (*drvSpiFlash_t* *d, uint32_t *offset*, size_t *size*)

erase flash sectors or blocks

Both *offset* and *size* must be sector (4KB) aligned.

When *size* is larger than sector or block, it will loop inside until all requested sector or block are erased.

It will wait flash erase finish. On success return, the whole region is erased.

It won't read back to verify the result.

When `d` is NULL, the default SPI flash will be used.

Return

- true on success
- false on invalid parameters

Parameters

- `d`: SPI flash instance pointer, must be valid
- `offset`: flash offset
- `size`: erase size

void **drvSpiFlashChipErase** (*drvSpiFlash_t* *d)

erase the whole flash chip

In normal case, it shouldn't be used by application.

When `d` is NULL, the default SPI flash will be used.

Parameters

- `d`: SPI flash instance pointer, must be valid

uint16_t **drvSpiFlashReadStatus** (*drvSpiFlash_t* *d)

read flash status register

When `d` is NULL, the default SPI flash will be used.

Return

- flash status register

Parameters

- `d`: SPI flash instance pointer, must be valid

void **drvSpiFlashWriteStatus** (*drvSpiFlash_t* *d, uint16_t *status*)
read flash non-volatile status register

When `d` is NULL, the default SPI flash will be used.

Parameters

- `d`: SPI flash instance pointer, must be valid
- `status`: value to be written to flash status register

void **drvSpiFlashWriteVolatileStatus** (*drvSpiFlash_t* *d, uint16_t *status*)
read flash volatile status register

When `d` is NULL, the default SPI flash will be used.

Parameters

- `d`: SPI flash instance pointer, must be valid
- `status`: value to be written to flash status register

void **drvSpiFlashEraseProgramSuspend** (void)
send suspend command if needed

During flash erase or program, the flash can't be accessed directly. It is needed to send suspend command before flash access.

This API is intended to be called in interrupt handler, and before ISR is called. In this API, it will be checked whether flash is in program or erase, and send suspend command if in program or erase. And this API is returned, flash becomes accessible.

Due to it is intended to be called in interrupt handler, there are no protection inside.

CHAPTER

TWENTYTHREE

RTC AND ALARM

Contents

23.1 Overview

23.2 API Reference

Defines

DRV_RTC_ALARM_INFO_SIZE
alarm opaque information size

Typedefs

typedef void (***drvRtcAlarmCB_t**)(**struct** *drvRtcAlarm* *alarm, void *ctx)
Callback function type to be called on alarm expiration.

In callback, it is prohibited to call alarm related APIs. Ecah owner should set there own callback first.

typedef struct *drvRtcAlarm* **drvRtcAlarm_t**
alarm information

Enums

enum drvRtcAlarmType_t
alarm type

Values:

DRV_RTC_ALARM_ONE_TIME
one time alarm

DRV_RTC_ALARM_WDAY_REPEATED
repeated alarm in selected day of the week.

Functions

void **drvRtcInit**(void)
initialize RTC driver

At initialization, NVM file may be read. And then it MUST be called after file system is initialized. Also, it will use ADI bus, and then it should be called after ADI bus is initialized.

bool **drvRtcSetAlarm**(uint32_t *owner*, uint32_t *name*, **const** void **info*, uint32_t *info_size*, int64_t *sec*, bool *replace*)
set one time alarm

Set a one time alarm with specified time. The time is offset in second from 1970-01-01 UTC.

When the alarm already exists, it will be replaced with the new parameters.

Return

- true on success
- false on failure
  - invalid parameter
  - alarm exists, and replace is false
  - out of memory

Parameters

- `owner`: alarm owner
- `name`: alarm name
- `info`: alarm opaque information
- `info_size`: alarm opaque information size
- `sec`: time second from 1970-01-01 UTC
- `replace`: true: replace existed alarm with the same owner and name

bool **drvRtcSetRepeatAlarm**(uint32_t *owner*, uint32_t *name*, **const** void **info*, uint32_t *info_size*, uint8_t *wday_mask*, uint32_t *sec_in_day*, int *timezone*, bool *replace*)
set repeated alarm

Set a repeated alarm with specified time.

Return

- true on success

- false on failure
  - invalid parameter
  - alarm exists, and replace is false
  - out of memory

Parameters

- `owner`: alarm owner
- `name`: alarm name
- `info`: alarm opaque information
- `info_size`: alarm opaque information size
- `wday_mask`: day of the week for the alarm. bit 0 for Sunday, bit 6 for Saturday.
- `sec_in_day`: alarm time in the day
- `timezone`: timezone (second offset from UTC) of the alarm
- `replace`: true: replace existed alarm with the same owner and name

bool **drvRtcRemoveAlarm**(uint32_t *owner*, uint32_t *name*)
remove an alarm

Return

- true on success
- false on failure

Parameters

- `owner`: alarm owner
- `name`: alarm name

int **drvRtcGetAlarmCount**(uint32_t *owner*)
get alarm count for specific owner

Return alarm count

Parameters

- `owner`: alarm owner

int **drvRtcGetAlarms**(uint32_t *owner*, *drvRtcAlarm_t* **alarms*, uint32_t *count*)
get alarm information for specific owner

The returned alarms will be ordered by expiration time.

Return count of alarm

Parameters

- `owner`: alarm owner
- `alarms`: pointer to alarm array for output (caller allocated)
- `count`: max count of alarms

int **drvRtcGetAllAlarmCount**(void)
get alarm count

Return alarm count

int **drvRtcGetAllAlarms**(*drvRtcAlarm_t* *alarms, uint32_t *count*)
get all alarm information

The returned alarms will be ordered by expiration time.

Return count of alarm

Parameters

- `alarms`: pointer to alarm array for output (caller allocated)
- `count`: max count of alarms

void **drvRtcRemoveAllAlarms**(void)
remove all alarms

bool **drvRtcAlarmOwnerSetCB**(uint32_t *owner*, void *context, *drvRtcAlarmCB_t* *cb*)
set alarm process callback for specific owner

Caller must set a alarm owner callback before set an alarm, or the alarm won't response.

Return

- true on success
- false on fail

Parameters

- `owner`: alarm owner
- `context`: caller context
- `cb`: callback (NULL is allowed)

void **drvRtcUpdateTime**(void)
update RTC time from system epoch time

int64_t **drvRtcGetClosestAlarm**(void)
get closest alarm in epoch time

Return

- closest alarm in epoch time in millisecond
- INT64_MAX if there are no alarms

struct drvRtcAlarmWdayRepeated_t
#include <drv_rtc.h> Parameters for day of the week repeated alarm

Public Members

uint8_t **wday_mask**
selected day of the week

uint32_t **sec_in_day**
time in day, in seconds

int32_t **timezone**
timezone of the alarm

struct drvRtcAlarm
#include <drv_rtc.h> alarm information

Public Members

uint32_t **owner**
alarm owner

uint32_t **name**
name of the alarm

drvRtcAlarmType_t **type**
type of the alarm

drvRtcAlarmWdayRepeated_t **wday_repeated**
day of the week repeated parameters

int64_t **expire_sec**
next expiration time

uint8_t **info**[**DRV_RTC_ALARM_INFO_SIZE**]
opaque information

CHAPTER

TWENTYFOUR

UART DRIVER

Contents

24.1 Overview

Uart can work at fifo mode and dma mode, this driver work at dma mode automatically, however, if dma resource is lacked, uart will work at fifo mode.

The driver set a buffer for tx/rx data, therefore the data will not be overlap if hardware resource is not ready most time. Caller should set a proper fifo size for both direction of datas.

There are three types of event the driver will notify the caller:

- DRV_UART_EVENT_RX_ARRIVED (new data is coming)
- DRV_UART_EVENT_RX_OVERFLOW (rx overflow, some data may be lost)
- DRV_UART_EVENT_TX_COMPLETE (all data in tx buffer are sent)

Caller may do something if some events are received.

24.2 API Reference

Typedefs

typedef struct drvUart **drvUart_t**

UART struct.

Only a declare here, caller have no necessary to know how it is implemented. And it is a anchor to identify which uart device is to be handled.

typedef void (***drvUartEventCB_t**) (void *param, uint32_t evt)
function type to notify data event

Enums

enum [anonymous]
UART data event.

Values:

DRV_UART_EVENT_RX_ARRIVED = (1 << 0)
Received new data.

DRV_UART_EVENT_RX_OVERFLOW = (1 << 1)
Rx fifo overflowed.

DRV_UART_EVENT_TX_COMPLETE = (1 << 2)
All data had been sent.

enum drvUartDataBits_t
UART data bits.

Values:

DRV_UART_DATA_BITS_7 = 7

DRV_UART_DATA_BITS_8 = 8

enum drvUartStopBits_t
UART stop bits.

Values:

DRV_UART_STOP_BITS_1 = 1

DRV_UART_STOP_BITS_2 = 2

enum drvUartParity_t
UART parity check mode.

Values:

DRV_UART_NO_PARITY
No parity check.

DRV_UART_ODD_PARITY
Parity check is odd.

DRV_UART_EVEN_PARITY
Parity check is even.

DRV_UART_SPACE_PARITY
Parity check is always 0 (space)

DRV_UART_MARK_PARITY
Parity check is always 1 (mark)

Functions

drvUart_t ***drvUartCreate** (uint32_t *name*, *drvUartCfg_t* **cfg*)
create an UART driver

Create an UART driver, caller should provide an valid UART FOURCC device name and an valid UART config.

Return

- NULL Fail to create an UART driver
- non-null Create an UART driver

Parameters

- `name`: FOURCC UART name (DRV_NAME_UART{1, 2, 3})
- `cfg`: UART config

bool **drvUartOpen** (*drvUart_t* **uart*)
open UART driver

Open the UART driver, done it can send/receive data via the UART.

Return

- true success
- false fail

Parameters

- `uart`: the UART driver

void **drvUartClose** (*drvUart_t* **uart*)
close UART driver

Close the UART driver, stop all data transfer. But do not release resource.

Note All data in tx/rx buffer will be purged.

Parameters

- `uart`: the UART driver

void **drvUartDestroy** (*drvUart_t* **uart*)
destroy the UART driver

Parameters

- `uart`: the UART driver

int **drvUartSend**(*drvUart_t* *uart, **const** void *data, size_t size)
send data via the UART

Send data via UART, the data may not send to hardware directly because the UART hw may be busy, and the data not sent to hw will be cached to a software tx buffer(see, drvUartCfg_t->tx_buf_size).

Note data may not send to hardware right away

Therefore the return value include the data had been sent and the data had been cached.

Return

- (-1) Parameter error
- OTHERS (>=0) The number of bytes actually sent (or cached)

Parameters

- `uart`: the UART driver
- `data`: data buffer to be sent
- `size`: data buffer size

int **drvUartSendAll**(*drvUart_t* *uart, **const** void *data, size_t size, uint32_t timeout_ms)
send data via the UART without cache in n milliseconds

Note data will not cached to tx fifo

Return

- (-1) Parameter error
- OTHERS (>=0) The number of bytes actually sent

Parameters

- `uart`: the UART driver
- `data`: data buffer to be sent
- `size`: data buffer size
- `timeout_ms`: timeout milliseconds

int **drvUartReceive**(*drvUart_t* *uart, void *buf, size_t size)
receive data from the UART

Receive data from the UART driver. Actually caller got data from a software rx fifo but not hw fifo directly.

After drvUartOpen the driver will store data from UART hw rx fifo automatically and trigger a DRV_UART_EVENT_RX_ARRIVED, app who use this driver can call the api after receive the event.

Return

- (-1) Parameter error
- OTHERS (>=0) The number of bytes actually receive from UART

Parameters

- `uart`: the UART driver
- `buf`: buffer to store data
- `size`: buffer size

int **drvUartReadAvail** (*drvUart_t* *uart)
inquire avalable bytes in rx buffer

Return

- (-1) Parameter error
- OTHERS (>=0) Available size in byte

Parameters

- `uart`: the UART driver

int **drvUartWriteAvail** (*drvUart_t* *uart)
inquire avalable space in tx buffer

Return

- (-1) Parameter error
- OTHERS (>=0) Available size in byte

Parameters

- `uart`: the UART driver

void **drvUartSetAutoSleep** (*drvUart_t* *uart, int *timeout*)
set whether UART will auto sleep

When enabled, UART will sleep after all bytes are transfered , with specified timeout.

To disable auto sleep feature, set `timeout` to -1 (or other negative values).

It maybe different among chips, or even different UARTs of one chip whether system will be waken up when there are UART input.

Parameters

- `uart`: the UART driver
- `timeout`: auto sleep wait time after transfer done. It can be 0 but not recommended. Negative value to disable auto sleep feature.

const *drvUartCfg_t* ***drvUartConfig** (*drvUart_t* *uart)
get uart configuration, will never return null

Return

- NULL fail, only when UART is NULL

- otherwise the UART configuration struct point.

Parameters

- `uart`: the UART driver

bool **drvUartReconfig**(*drvUart_t* *uart, *drvUartCfg_t* *cfg)
reconfig the uart

Note caller should make sure the UART had already closed, or it will return false if `cfg` is NULL, it will do nothing and return true

Return

- true sucess
- false fail

Parameters

- `uart`: the UART driver
- `cfg`: the configuration

bool **drvUartWaitTxFinish**(*drvUart_t* *uart, uint32_t *timeout*)
wait tx data send finished

Return

- true data all sent
- false data residual

Parameters

- `uart`: the UART driver
- `timeout`: wait time in millisecond

void **drvUartBlueScreenInit**(void)
prepare UART blue screen

void **drvUartBlueScreenPoll**(void)
poll for UART blue screen

struct drvUartCfg_t
#include <*drv_uart.h*> UART config.

Public Members

uint32_t **baud**
baudrate, 0 for auto baud

drvUartDataBits_t **data_bits**
data bits

drvUartStopBits_t **stop_bits**
stop bits

drvUartParity_t **parity**
parity check mode

bool **auto_baud_lc**
"at" ot "AT" for auto baud. 8910 can detect both "at" and "AT". So, it will be ignored in 8910.

bool **cts_enable**
enable cts or not

bool **rts_enable**
enable rts or not

size_t **rx_buf_size**
rx buffer size

size_t **tx_buf_size**
tx buffer size

uint32_t **event_mask**
event mask, identify which event will be notified

drvUartEventCB_t **event_cb**
event notify callback function

void ***event_cb_ctx**
app private param (like a context)

CHAPTER

TWENTYFIVE

AXIDMA DRIVER

Contents

25.1 Overview

A generic axidma driver. Caller should request a channel, then config the channel to transfer data.

If an address is a hardware register (like a fifo), config this address as a fix_addr. If source address is a fifo, and the data is not coming, must set trigger mode.

8910:

- There are 12 axidma channels, [0:5] for ap, [6:11] for cp

25.2 API Reference

Typedefs

typedef void (***drvAxidmaIsr_t**) (*drvAxidmaIrqEvent_t* evt, void *p)
function type to process AXIDMA channel interrupt

typedef struct drv_axidma_channel **drvAxidmaCh_t**
the AXIDMA channel, call do not need to know its implement

typedef struct *drv_axidma_config* **drvAxidmaCfg_t**
AXIDMA channel config.

Enums

enum drvAxidmaIrqEvent_t
AXIDMA interrupt event.

Values:

AD_EVT_FINISH = (1 << 0)
event transfer done

AD_EVT_PART_FINISH = (1 << 1)
event transfer once done (part size)

AD_EVT_STOP = (1 << 2)
event the channel stop

enum drvAxidmaDataType_t
AXIDMA data type.

Values:

AD_DATA_8BIT = 0

AD_DATA_16BIT = 1

AD_DATA_32BIT = 2

AD_DATA_64BIT = 3

Functions

void **drvAxidmaInit**(void)
AXIDMA module initialization.

CP will rely on the initialization, so it should be called before release CP reset.

drvAxidmaCh_t ***drvAxidmaChAllocate**()
allocate an AXIDMA channel

Return

- NULL no free axidma channel
- OTHERS axidma channel

void **drvAxidmaChRelease**(*drvAxidmaCh_t* **ch*)
release an AXIDMA channel

Parameters

- `ch`: the AXIDMA channel

bool **drvAxidmaChStart**(*drvAxidmaCh_t* **ch*, **const** *drvAxidmaCfg_t* **cfg*)
start the AXIDMA channel

Return

- true success
- flase fail

Parameters

- `ch`: the AXIDMA channel
- `cfg`: start config

void **drvAxidmaChStop** (*drvAxidmaCh_t *ch*)
stop the AXIDMA channel

Stop the AXIDMA channel, all transfer will stop, may triger a STOP event. Do not reset AXIDMA registers, caller can call this then read channel pending data count.

Parameters

- `ch`: the AXIDMA channel

bool **drvAxidmaChBusy** (*drvAxidmaCh_t *ch*)
check if the AXIDMA channel is busy

Return

- true busy
- false not busy

Parameters

- `ch`: the AXIDMA channel

void **drvAxidmaChSetDmamap** (*drvAxidmaCh_t *ch*, uint8_t *req_source*, uint8_t *ack_map*)
set AXIDMA map for a specific channel

Parameters

- `ch`: the AXIDMA channel
- `req_source`: request source
- `ack_map`: ack map

uint32_t **drvAxidmaChCount** (*drvAxidmaCh_t *ch*)
get AXIDMA channel pending count

Get pending data for an AXIDMA channel in bytes. Caller can calculate the AXIDMA actually transfered data size by total size minus this count.

Return

- (Non-negative integer) pending data count in byte

Parameters

- `ch`: the AXIDMA channel

void **drvAxidmaChRegisterIsr**(*drvAxidmaCh_t* *ch, *drvAxidmaIsr_t* isr, void *param)
register a interrupt handler for a specific AXIDMA channel

Parameters

- `ch`: the AXIDMA channel
- `isr`: interrupt routine
- `param`: call private data (like context)

void **drvAxidmaStopAll**(void)
stop all AXIDMA channel

It is only intented to be called at enter panic.

struct drv_axidma_config

Public Members

uint32_t **src_addr**
source address

uint32_t **dst_addr**
destination address

uint32_t **data_size**
data size in byte

uint32_t **part_trans_size**
transfer size once in byte

drvAxidmaDataType_t **data_type**
dma data type

drvAxidmaIrqEvent_t **mask**
react interrupt mask

uint8_t **src_addr_fix**
source address fixed (like hw fifo)

uint8_t **dst_addr_fix**
destination address fixed

uint8_t **sync_irq**
request trigger mode

uint8_t **force_trans**
force trans start

uint8_t **req_sel_level**
high level trigger

CHAPTER

TWENTYSIX

PMIC ADC

Contents

26.1 Overview

The AuxADC is a 32-channel 12bits output ADC,it samples VBAT voltage, and etc.

Note: All 32 channels can be controlled by SW.

26.2 API Reference

Defines

ADC_CHANNEL_INVALID

Enums

enum adc_aux_scale

Values:

ADC_SCALE_1V250 = 0

ADC_SCALE_2V444 = 1

ADC_SCALE_3V233 = 2

ADC_SCALE_5V000 = 3

ADC_SCALE_MAX = 3

enum adc_aux_channel

Values:

ADC_CHANNEL_BAT_DET = 0

ADC_CHANNEL_1 = 1

ADC_CHANNEL_2 = 2

ADC_CHANNEL_3 = 3

ADC_CHANNEL_4 = 4

ADC_CHANNEL_VBATSENSE = 5

ADC_CHANNEL_6 = 6

ADC_CHANNEL_TYPEC_CC1 = 7

ADC_CHANNEL_THM = 8

ADC_CHANNEL_TYPEC_CC2 = 9

ADC_CHANNEL_10 = 10

ADC_CHANNEL_11 = 11

ADC_CHANNEL_12 = 12

ADC_CHANNEL_DCDC_CALOUT = 13

ADC_CHANNEL_VCHGSEN = 14

ADC_CHANNEL_VCHG_BG = 15

ADC_CHANNEL_PROG2ADC = 16

ADC_CHANNEL_17 = 17

ADC_CHANNEL_18 = 18

ADC_CHANNEL_SDAVDD = 19

ADC_CHANNEL_HEADMIC = 20

ADC_CHANNEL_LDO_CALOUT0 = 21

ADC_CHANNEL_LDO_CALOUT1 = 22

ADC_CHANNEL_LDO_CALOUT2 = 23

ADC_CHANNEL_24 = 24

ADC_CHANNEL_25 = 25

ADC_CHANNEL_26 = 26

ADC_CHANNEL_27 = 27

ADC_CHANNEL_28 = 28

ADC_CHANNEL_SELF_OFFSET_MEASURE = 29

ADC_CHANNEL_DP = 30

ADC_CHANNEL_DM = 31

ADC_CHANNEL_MAX = 31

Functions

void **drvAdcInit** (void)
initialize ADC driver it should be called after ADI bus is initialized.

int32_t **drvAdcGetChannelVolt** (uint32_t *channel*, int32_t *scale*)
return the volt for the given channel in mV.

Return -1 failure the volt vlaue.

Parameters

- `channel`: set the channel to measue
- `scale`: set the scale of the channel

int32_t **drvAdcGetRawValue** (uint32_t *channel*, int32_t *scale*)
return the raw adc value for the given channel.

Return -1 failure the raw vlaue.

Parameters

- `channel`: set the channel to measue
- `scale`: set the scale of the channel

CHAPTER

TWENTYSEVEN

I2C DRIVER

Contents

27.1 Overview

Because I2C is a two-wire, bi-directional serial bus that provides a simple and efficient method of data exchange between devices. So I2c only open one time, when used i2c to send or get data,we must wait i2c bus idle.

8910:

- Granite Chip includes 3xI2C masters.
- Only 100Kbps and 400Kbps modes are supported directly.

27.2 API Reference

Typedefs

typedef struct drvI2cMaster **drvI2cMaster_t**
the i2c master indicator

Enums

enum drvI2cBps_t
i2c bps enumeration

Values:

DRV_I2C_BPS_100K
normal 100Kbps

DRV_I2C_BPS_400K
fast 400Kbps

DRV_I2C_BPS_3P5M
high speed 3.5Mbps

Functions

drvI2cMaster_t ***drvI2cMasterAcquire** (uint32_t *name*, *drvI2cBps_t bps*)
acquire the i2c master

Return

- (NULL) fail
- otherwise the i2c master instance

Parameters

- `name`: name of the i2c master
- `bps`: the i2c speed

void **drvI2cMasterRelease** (*drvI2cMaster_t *i2c*)
release the i2c master

Parameters

- `i2c`: the i2c master

bool **drvI2cWrite** (*drvI2cMaster_t *i2c*, **const** *drvI2cSlave_t *slave*, **const** uint8_t *data*, uint32_t *length*)
i2c master send data

Return

- true success
- false fail

Parameters

- `i2c`: the i2c master
- `slave`: the i2c slave
- `data`: data to send
- `length`: data length

bool **drvI2cRead** (*drvI2cMaster_t *i2c*, **const** *drvI2cSlave_t *slave*, uint8_t *buf*, uint32_t *length*)
i2c master get data

Return

- true success
- false fail

Parameters

- `i2c`: the i2c master
- `slave`: the i2c slave
- `buf`: buffer to receive data
- `length`: buffer length

bool **drvI2cWriteRawByte**(*drvI2cMaster_t* *i2c, uint8_t *data*, uint32_t *cmd_mask*)
i2c master send raw byte

Return

- true success
- false fail

Parameters

- `i2c`: the i2c master
- `data`: raw byte to send
- `cmd_mask`: command associated with this byte.

bool **drvI2cReadRawByte**(*drvI2cMaster_t* *i2c, uint8_t *data, uint32_t *cmd_mask*)
i2c master read raw byte

Return

- true success
- false fail

Parameters

- `i2c`: the i2c master
- `data`: room for the byte
- `cmd_mask`: the command mask required for the final phase of the SCCB read cycle.

struct drvI2cSlave_t
#include <*drv_i2c.h*> i2c slave definition

Public Members

uint8_t **addr_device**

uint8_t **addr_data**

CHAPTER

TWENTYEIGHT

IFC

Contents

28.1 Overview

IFC is DMA for slow peripherals.

IFC controller may have several channels. Before using IFC, it is needed to *request* a channel. And after the channel is used, it is needed to *release* the channel.

Note: IFC controller has `get_ch` hardware register. At reading, it will return the next available channel ID. And the channel usage information will be updated in hardware.

When there are no available channels, `get_ch` will return 0xff.

Each IFC channel can be used for any supported hardware modules. The used hardware module is configured in `control.req_src` field. For example, when this field is written to `SYS_ID_TX_UART1`, this channel will be used by UART1 TX DMA. This field must be filled before IFC start.

There are nearly no sanity checks in IFC driver. Caller should take care APIs are called in valid conditions.

28.1.1 8955 and 8909

IFC driver covers *sys_ifc*. *bb_ifc* and *audio_ifc* controller is different from *sys_ifc*, and this driver isn't for them.

28.1.2 8910

IFC driver covers *sys_ifc* and *aon_ifc*.

28.2 Auto Mode

IFC hardware support *auto mode*, which means that when the DMA is done, the channel will be released automatically. This feature won't be used in software.

28.3 Interrupt

IFC module itself won't generate interrupt. Rather, the DMA interrupt will be generated in the module using IFC. For example, when UART with IFC is working on DMA mode, UART module will have `rx_dma_done` and `tx_dma_done` interrupt.

28.4 Channels

Usually, the available hardware channel count is smaller than possible request count. It is possible fail to request hardware channel.

Though it is rare that too many hardware channels are needed simultaneously, this case should be considered. General guidelines:

- Debughost TX will always occupy a channel for trace;
- Debughost RX may occupy a channel for PC commands. Though it is not so important, the possibility of losing data in FIFO mode is very high;
- UART RX will occupy a channel;
- UART TX will use dynamic channel. When there are data to be output, hardware channel shall be requested (actually, waited). And when the transfer is done, the channel shall be released;
- SIM RX and TX will use dynamic channels;

28.5 Cache Coherence

Cache coherence should be considered when using IFC. However, IFC driver is a *thin* wrapper for hardware, cache coherence isn't considered inside the driver. Caller should take care cache coherence.

28.6 Thread Safe

IFC driver is intended to called by other drivers. The APIs are **NOT** thread safe, and **NOT** interrupt safe. Caller should consider thread safe and interrupt safe.

28.7 API Reference

Enums

enum drvIfcDirection_t
IFC direction.

Values:

DRV_IFC_RX
RX, from peripherals to memory.

DRV_IFC_TX
TX, from memory to peripherals.

Functions

bool **drvIfcChannelInit**(*drvIfcChannel_t* *ch, uint32_t *name*, *drvIfcDirection_t* *dir*)
initialize IFC channel data structure

Return

- true on success
- false if the device name or direction is not supported by IFC

Parameters

- `ch`: IFC channel to be initialized
- `name`: device name
- `dir`: IFC direction

bool **drvIfcRequestChannel**(*drvIfcChannel_t* *ch)
request a hardware IFC channel

Return

- true if a hardware channel is requested
- false if there are no available hardware channels

Parameters

- `ch`: IFC channel data structure, must be valid

void **drvIfcWaitChannel**(*drvIfcChannel_t* *ch)
request and wait a hardware IFC channel

It can't be called inside ISR.

Parameters

- `ch`: IFC channel data structure, must be valid

void **drvIfcReleaseChannel**(*drvIfcChannel_t* *ch)
release the hardware channel

Parameters

- `ch`: IFC channel data structure, must be valid

bool **drvIfcReady**(*drvIfcChannel_t* *ch)
whether there is requested hardware channel

Return

- true if a hardware channel is requested
- false if no hardware channels are requested

Parameters

- `ch`: IFC channel data structure, must be valid

void **drvIfcFlush**(*drvIfcChannel_t* *ch)
flush hardware IFC channel

This should be only called for RX channel. At flush, IFC controller will pause to fetch data from hardware module. For example, when *drvIfcFlush* is called for UART RX channel, UART arriving data won't be transfered to memory configured in IFC. Rather, they will be put to UART FIFO.

Parameters

- `ch`: IFC channel data structure, must be valid

void **drvIfcClearFlush**(*drvIfcChannel_t* *ch)
clear flush state of IFC channel

It should be only called on *flushed* IFC channel. After it is called, IFC will continue to fetch data from hardware module.

Parameters

- ch: IFC channel data structure, must be valid

uint32_t **drvIfcGetTC** (*drvIfcChannel_t* *ch)
get current transfer count

TC will be decreased from the initial value. Current transfer count is the remaining byte count for the configured DMA transfer.

Return

- *TC*

Parameters

- ch: IFC channel data structure, must be valid

void **drvIfcStart** (*drvIfcChannel_t* *ch, **const** void *address, uint32_t *size*)
start a transfer

Caller should take care cache coherence before.

Parameters

- ch: IFC channel data structure, must be valid
- address: DMA address
- size: DMA size

void **drvIfcStop** (*drvIfcChannel_t* *ch)
stop a transfer

After stop, *TC* will be set to 0. If the *TC* before stop, it is needed to call drvIfcGetTC before stop.

Parameters

- ch: IFC channel data structure, must be valid

void **drvIfcExtend** (*drvIfcChannel_t* *ch, uint32_t *size*)
extend current transfer

Caller should take care cache coherence before.

When the next DMA address is adjacent to the previous started DMA end address, *drvIfcExtend* can be called to transfer mode bytes. For RX channel, it means receiving more bytes. For TX channel, it means sending more bytes.

It can be called even the DMA is on going.

Parameters

- ch: IFC channel data structure, must be valid
- size: extend DMA size

bool **drvIfcIsFifoEmpty** (*drvIfcChannel_t *ch*)
whether the IFC fifo is empty

Return

- true if the hardware IFC channel FIFO is empty
- false otherwise

Parameters

- `ch`: IFC channel data structure, must be valid

bool **drvIfcIsRunning** (*drvIfcChannel_t *ch*)
whether the hardware is running

Return

- true if the hardware IFC channel is started, and not finished
- false otherwise

Parameters

- `ch`: IFC channel data structure, must be valid

void **drvIfcWaitDone** (*drvIfcChannel_t *ch*)
wait IFC channel is done

Parameters

- `ch`: IFC channel data structure, must be valid

struct drvIfcChannel_t
#include <drv_ifc.h> IFC channel data structure.

Don't access the members directly.

CHAPTER

TWENTYNINE

PPP GUIDES

Contents

29.1 Linux

The followings are verified on Ubuntu 16.04

29.1.1 Prepare

```
$ sudo apt install ppp
```

29.1.2 Permission

```
$ sudo adduser <username> dialout
```

It is needed to logout and re-login to make the setting take effect.

29.1.3 PPP Setting

Create file `/etc/peers/gprs`:

```
/dev/ttyACM0    # this is the tty device name
115200          # baud rate
nolock          # not lock device file
local           # not detect CD, not signal DTR
debug           # enable debug
nocrtscts       # no hardware flow control
nodetach        # run on foreground
noauth          # no authentication
usepeerdns      # ask peer up to 2 DNS servers
nodefaultroute  # not add to system routing table
user ''
connect '/usr/sbin/chat -s -v -f /etc/ppp/chat-gprs-connect'
```

Create file `/etc/ppp/chat-gprs-connect` (mentioned above):

```
TIMEOUT 5
ECHO ON
ABORT '\nBUSY\r'
ABORT '\nERROR\r'
ABORT '\nRINGING\r\n\r\nRINGING\r'
ABORT '\nCOMMAND NO RESPONSE!\r'
'' AT

TIMEOUT 60
SAY "Press CTRL-C to break the connection process.\n"
OK 'ATE0'
OK 'ATD*99***1#'

TIMEOUT 60
SAY "Waiting for connect...\n"
CONNECT ''
SAY "connect Success!\n"
```

Note: *gprs* is just an example. It can be arbitrary name.

29.1.4 Setup PPP

```
$ ppp call gprs
```

29.1.5 Setup Routing Table for Specified Address

In most cases, it is desired to specified only the specified testing address will use PPP interface. It can be archived by setting touring table:

```
$ sudo ip route add xx.xx.xx.xx/32 dev ppp0
```

The setting is temporal. Each time PPP interface is created, it will be applied again.

29.2 Windows

29.2.1 Create Modem

- *Control Panel -> Phone and Modem -> Modem -> Add*;
- Check *Don't detect my modem, I will select it from a list*, then *Next*;
- *(Standard Modem Types) -> Standard 33600 Modem -> Next*;
- Select the COM port, then *Next -> Finish*;
- Select the created mode, *Properties -> Change Settings -> Advanced -> Change Default Preferences -> General*, change *Flow Control* to *None*;
- If the COM port is USB CDC device of module, the baud rate doesn't master. If it connected to module UART, change the baud rate to the baud rate of module UART.

29.2.2 Create Dial-up Network

- *Control Panel -> Network and Sharing Center -> Set up a new connection or network -> Set up a dial-up connection*;
- Set *Dial-up phone number* as `*99***1#` (the last `1` before `#` is to specify CID);
- Set *Connection name* to a friendly name, such as `PPP0`;
- Check *Allow other people to use this connection*;

29.2.3 Tune Dial-up Network

The followings are not absolutely necessary, but it is recommended. Otherwise, there are too many traffics from Windows system, and will affect testing.

- *Control Panel -> Network and Sharing Center -> Change adapter settings*;

- Right click *PPP0*, if *Cancel as Default Connection* appears, check it;
- Right click *PPP0* -> *Properties* -> *Networking*, uncheck all unnecessary protocols and services;
- Right click *PPP0* -> *Properties* -> *Networking*, choose *Use the following DNS server addresses*, and set *Preferred DNS server* to the DNS server of PC NIC;
- Right click *PPP0* -> *Properties* -> *Networking* -> *Internet Protocol Version 4 (TCP/IPv4)* -> *Properties* -> *Advanced*,
  - Uncheck *Use default gateway on remote network*, keep *Disable class based route addition* unchecked;
  - Uncheck *Automatic metric*, and set *Interface Metric* to 9999;
  - *Use IP header compression* can be kept as checked;
- *Start* -> *services.msc*, disable *SSDP Discovery* and reboot PC;

After the settings are changed, it is needed to re-connect PPP0.

With this setting, PPP0 won't be used by default. When it is needed to access specified remote address through PPP0, it is needed to set route manually.

For example, the module IP address is *10.11.22.33*, and the default route will be *10.0.0.0*. And it is needed to access *111.205.140.137* from PPP0:

```
> route delete 10.0.0.0
> route add 111.205.140.137 mask 255.255.255.255 10.11.22.33
```

Administrator is needed to execute the above command.

29.2.4 Baud Rate Higher Than 115200

By default, the maximum baud rate of the created modem is 115200. When higher baud rate is needed, it is needed to edit registry manually.

Open *regedit.exe*, and find the registry of the modem, for example:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Class\{4D36E96D-E325-11CE-BFC1-
↪08002BE10318}\0000
```

Double click *DCB* for editing. the bytes [7:4] is the maximum baud rate in little endian. For example, *00 C2 01 00* is 115200, *00 10 0E 00* is 921600.

CHAPTER

THIRTY

CODING STYLE GUIDE

Contents

30.1 Copyright Header

All source files, include C/C++ source files, header files, Python scripts and CMakeLists.txt should add standard copy right header **if and only if** it is really developed by ourselves. For files from open source community or third party vendors, the original copyright header **MUST** be kept.

30.2 Indent

All C/C++ source files developed by ourselves should be indented by clang-format (in `prebuilt` directory), with `.clang-format` in root directory.

Source files from open source community or third party vendors, the original style **MUST** be kept.

`ninja beautify` will indent the specified C/C++ source codes with clang-format. An example to specify files to be indented in CMakeLists.txt:

```
file(GLOB srcs include/*.h src/*.c src/*.h)
beautify_c_code(${target} ${srcs})
```

Note: `ninja beautify` will be checked in CI.

30.3 Line Length

Too long line is bad for reading. However, it is possible to use long function or variable name, long line is hard to be avoided completely.

- Try best not to exceed 80 characters per line;
- Don't exceed 100 characters, unless there are enough reasons.

30.4 File Name Convention

File name should be lower case, joined by underscore (_). The first word should be the module name. Such as:

```
osi_log.h
drv_uart.h
```

30.5 Function Name Convention

Function name should be lower case module name followed by CamelCase words. The module name should match the module name used in file name. Such as:

```
osiMalloc
drvUartClose
```

30.6 Static Functions

Static function name **MUST** add `prv` prefix. The function name can be shorter. Such as:

```
prvIrqHandler
prvHandleByte
```

Note: Previously, it is required to add underscore (_) as prefix. However, it is found that _ is hard for reading. So, the prefix is changed to `prv`. So, _ is permitted, and `prv` is recommended.

If a function is not indented to be called outside the file, it **must** be declared as static.

Public function name should be easier to be understood by module user (it should be assumed that module user doesn't know too many details about the module implementation), and static function should be easier to be understood by module maintainer (it can be assumed that module maintainer understand more details about the module implementation).

Typically, embedded system will be linked into one binary. Longer public function name can avoid function conflict. However, name conflict doesn't impact static functions.

30.7 Local Variable Name Convention

Variable name should be lower case word with underscore. The name should be helpful to understand the purpose of the variable. Both long name and short name are acceptable.

30.8 Global Variable Name Convention

Global variables should prefix by `g`, and followed by camel cases, both for static or non static.

30.9 Struct Name Convention

`struct` name follow the same name convention with function name. If `typedef` is used, the defined type should add `_t` suffix. Field name should follow local variable name, that is lower case words, joined by underscore (_). Such as:

```
struct drvUart {
    int name;
    int stop_bits;
};
typedef struct drvUart drvUart_t;
```

30.10 enum

Keep in mind that the size of enum is compiler dependent. It is permitted to use fixed length integer type in struct, even the field value should be enum.

enum name follows the same convention as struct. `_e` won't be cleaner than `_t`.

enum value **SHOULD** be upper case word with underscore. Such as:

```
typedef enum {
    DRV_UART_DATA_BITS_7 = 7,
    DRV_UART_DATA_BITS_8 = 8
} drvUartDataBits_t;
```

If it is needed to make sure the size of enum is the same for compilers, add a dummy name with value `0x7fffffff`:

```
enum
{
    CFW_SIM_0 = 0x00,
    CFW_SIM_1 = 0x01,
    CFW_SIM_END = 0xFF,

    CFW_SIM_ENUM_FILL = 0x7FFFFFFF
};
```

30.11 C++ Class, Method and Member Name

C++ class name should be CamelCase, with initial upper case character. C++ class method should be camelCase, with initial lower case character. C++ class member should have prefix `m` with CamelCase. Such as:

```
class SomeClass
{
public:
    void someMethod();

private:
    int mSomeMember;
};
```

30.12 stdint, stdbool

Use types defined in `stdint.h` and `stdbool.h`. In most cases, the followings should be enough:

```
bool
char
int, unsigned
int8_t, uint8_t
int16_t, uint16_t
int32_t, uint32_t
int64_t, uint64_t
ssize_t, size_t
intptr_t, uintptr_t
```

DON'T use the followings:

```
BOOL, CHAR, UINT8, UINT32, u8, u32, ...
```

When the exact bytes of integer is unimportant, `int, unsigned` is simpler and recommended.

30.13 const, void *

For pointer type parameter, `const` **MUST** be added when it is permitted to pass a const pointer. Otherwise, it will a pain for serious programmer to use the module.

When the parameter is just a block of memory, `void *` should be used. Otherwise, caller will add tedious and meaningless type cast.

`read` and `write` are good prototype example:

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

30.14 Object Oriented

Keep *object oriented programming* in mind, even C is using. For many modules APIs, the first parameter should be an instance pointer.

From the beginning, design APIs for multiple instances, except there are strong reason that multiple instances is impossible.

30.15 extern

DON'T Use `extern` to declare function or variables of other modules. The correct method is to include the public header. When other modules are changes, compiler can't detect mismatch of `extern`.

30.16 Global Variables

Global variables is evil, but is inevitable in some cases. Also, it is helpful for debugger. Global variable is evil just because it may break multiple instances. Keep multiple instances in mind, and use global variables if needed.

30.17 Public Header

In public header file, if not absolutely needed, **DON'T** put struct definition in public header file. Rather, forward declaration should be used. Such as:

```
typedef struct osiEvent osiEvent_t;
bool osiEventSend(osiThread_t *thread, const osiEvent_t *event);
```

This can simplify header file dependency. Also, less details is easier to be understood for module user.

For low level codes, and it is **really** want to embed the struct into another larger data struct, it is permitted to put struct details in the end of public header file. And the following comment should be added:

```
// ============================================================================
// IMPLEMENTATION. DON'T USE THEM DIRECTLY
// ============================================================================
```

When struct definition will bring in unnecessary include dependency, **DON'T** put struct definition in public header. Code structure is more important than optimization.

Doxygen style documentation is *required* for public headers.

30.18 extern "C"

For C headers, `extern "C"` **MUST** be added. It is possible in some day, a C++ module will include the header file.

30.19 Parameter Checking

For public API, **all** parameters should be checked. In API document, the behavior of invalid parameter must be documented. In some aspects, exception behavior is more important than normal behavior.

Exception: for object-oriented style API, the first parameter is the object. It is permitted and suggested not to check the object pointer. Like C++, there are no opportunities to check `this` pointer. Redundant checking just means the module is not well designed.

For private functions, it is permitted not to check parameter if it is already checked in public APIs.

Note: All *delete* type APIs should permit NULL pointer, just like free(3).

30.20 Return bool or int

When the reason of failure is important, and there really exist modules will care the reason, use `int`, 0 for success, and negative value for error reason. Otherwise, return `bool`.

If nobody cares about the failure reason, `int` is over-design.

30.21 Warning

Warning is evil and all warnings **must** be cleaned.

`-Wall` is always enabled, `-Werror` is disabled by default. and enabled in CI.

Note: When compiler changed, or compiling options changed, it is possible that more warning will be reported. The minimal requirement is to resolve all warnings reported by current compiler, current options.

CHAPTER

THIRTYONE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

Symbols

A

C

D

F

G

H

I

O

P

R

S