Development Guide

Release 1.0

RDA

Mar 16, 2018

CONTENTS:

CHAPTER

ONE

INTRODUCTION

This document will give some basic introductions for software development.

The RDA modem software is built up based on a dual CPU architecture. One CPU runs SX RTOS and is responsible for all applications and the whole software stack, including GSM/NB protocol stack. It is called XCPU in short. Another CPU is purely doing some algorithms for wireless communications, the software running on it is inner a main loop without OS. This one is called BCPU in short. The two CPUs communicate through a mailbox.

CHAPTER

TWO

GENERAL NOTES ABOUT RDA IOT MODEM PROGRAMMING

2.1 Application startup flow

This note explains various steps which happen before `main` function is called.

The high level view of startup process is as follows:

1. After code of ROM run finished, CPU will jump to address in flash with offset 0x10, where the first-stage bootloader start.
2. The first stage bootloader is responsible for ota upgrading. If no ota needs, program will jump to second-stage bootloader.
3. Second-stage bootloader initializes hardware resources and the whole platforms.
4. Finally, we reach to `main`. At this point the second CPU and RTOS scheduler can be started. Interrupts are also unmasked and the system can respond to interrupts.

2.2 First stage bootloader

The first stage bootloader works mainly for FOTA. Its code is located at directory `platform/chip/bootloader/` and `toolpool/blfota/` and compiled separately.

The generated .lod file named with a prefix `blfota`.

To make it, please cd to `toolpool/blfota/` and run `make`.

The first stage bootloader is optional and can be disabled at menuconfig.

2.3 Second stage bootloader

Second-stage bootloader initializes some hardware resources, such as power, clock and storage. The text/data/bss etc. sections are initialized as well.

After that, the HAL(hardware abstract layer) is initialized and some other features in platform start to work, including log output, clock system, power management, peripheral devices and IOs.

2.4 Main function

`main()` works as a entry for the OS and see source code `platform/base/sap/src/sap_main.c` for more details.

CHAPTER

THREE

BUILD SYSTEMS

The build system consists of two parts:

3.1 COMPILER VARs and RULEs

Compiler variables:

- Defines all variables for makefile, see `env/compilation/compilevars.mk` for more details.
- `Kconfig` is used to configure the system and related configurations are saved into `target/.../config.mk`.
- User can also find some variables defined in file `platform/chip/defs/8909_base.def` and `target/.../target.def`.
- In addition, makefile can append compiler used flags by adding them into `${GLOBAL_EXPORT_FLAG}` `${TARGET_EXPORT_FLAG}` `${CHIP_EXPORT_FLAG}` `${LOCAL_EXPORT_FLAG}`.

Compiler rules:

The core rules of Makefile are defined in `env/compilation/compilerules.mk`.

Note, above two mk are included in all Makefile of the project specifically. `compilevars.mk` is included at header and `compilerules.mk` in the end.

```
include ${SOFT_WORKDIR}/env/compilation/compilevars.mk

# Makefile content

include ${SOFT_WORKDIR}/env/compilation/compilerules.mk
```

3.2 Linker scripts

The link script for application side is `env/link/modem2G_master.ld` The link script for bcpu side is `env/link/modem2G_bcpu_master.ld`

CHAPTER

FOUR

CODE TREE INTRODUCTION

4.1 Source code tree

```
├── application          - application development(modem not concerned)
│   ├── adaptation
│   ├── coolmmi
│   ├── CoolSimulator
│   ├── diag
│   ├── include
│   ├── Makefile
│   ├── mmi_cfg
│   ├── mmi_customer
│   ├── NetworkSimulator
│   ├── ReadMe.txt
│   ├── src
│   ├── systeminc
│   ├── target_res
│   ├── thirdpartylibs
│   └── unit_test
├── at                   - AT command processing framework
│   ├── ate
│   ├── ats
│   ├── c.sh
│   ├── include
│   ├── internal_doc
│   ├── iot
│   ├── Makefile
│   ├── src
│   ├── thirdpartylibs
│   └── utility
├── env                  - build related environment settings
│   ├── compilation
│   ├── launch.sh
│   ├── lib
│   ├── link
│   ├── menuconfig.sh
├── Kconfig
├── Makefile
├── newfcfg              - scripts for menuconfig
```

```
│   ├── application
│   ├── at
│   ├── custom
│   ├── Kconfig.debug
│   ├── Kconfig.default
│   └── platform
├── platform            - all system platform related code
│   ├── base            - OS related
│   ├── chip            - very low level modules for hardware,
│                         including rom, bootloader, HAL, registers etc.
│   ├── edrv            - device drivers, including audio, camera, lcd, gpio, pmu, rf␣
↪etc.
│   ├── include
│   ├── Makefile
│   ├── service         - service layer serves as a bridge between application and␣
↪drivers,
│                         including communication, media, tcp/ip, file system related␣
↪modules.
│   └── system          - this directory contains calibration algorithm, protocol␣
↪stacks and
│                         some post processing blocks. (most delivered as binary to␣
↪customers)
├── platform_test       - test cases
│   ├── calib
│   ├── cos
│   ├── csw
│   ├── hal
│   ├── pal
│   ├── qc
│   ├── reftest
│   ├── rom
│   ├── service
│   ├── spp
│   ├── stack
│   ├── svc
│   ├── sx
│   └── vpp
├── target               - specific configurations for various platforms and products
│   ├── 8809_modem
│   ├── 8908a_nb_modem_p3
│   ├── 8909_2g_evb_watch
│   ├── 8909_2g_modem_jt
│   ├── 8909_nb_modem_jt
│   ├── include
│   ├── Makefile
│   ├── src
│   ├── target_common.def
└── toolpool             - some useful tools are here, including downloading program
                           called ``ramrun``, ``fota``upgrading program and so on.
    ├── 8955_ddr_check
    ├── 8955_factory_ate
    ├── 8955_factory_test
    ├── bl_ddr_check
    ├── blfota
    ├── blfpc
    ├── bl_uart_check
    ├── spiffs_tool
    └── usrgenerator
```

CHAPTER

FIVE

SYSTEM

5.1 Overview

SX The RTOS used in the modem system is not popular, named "SX", which contains most of basic features like other sort of RTOS. Those contain task, queue, mutex, semaphore, timer and dynamic memory management. We don't spend too much time talking about the os deeply, please check the source code in `platform/base/sx/` for more details.

COS As mentioned above, "SX" is not a popular RTOS and its APIs are not clear and easy understanding for application developers. Therefore, "COS" was created to work as a OS abstraction layer, wrapping all the OS related functions and providing more understandable interfaces. The header file `platform/include/cos.h` list all API of "COS".

NOTE: **some deprecated APIs are not list below and please do not ues them either.**

5.1.1 OS API

Task API

```
HANDLE COS_CreateTask(PTASK_ENTRY pTaskEntry,
              PVOID pParameter,
              PVOID pStackAddr,
              UINT16 nStackSize,
              UINT8 nPriority,
```

```
                UINT16 nCreationFlags,
                UINT16 nTimeSlice,
                PCSTR pTaskName);

VOID COS_StartTask(TASK_HANDLE *pHTask, PVOID pParameter);

VOID COS_StopTask(TASK_HANDLE *pHTask);

BOOL COS_DeleteTask(HANDLE hTask);

BOOL COS_SuspendTask(HANDLE hTask);

BOOL COS_ResumeTask(HANDLE hTask);

HANDLE COS_GetCurrentTaskHandle(void);
```

Event

```
typedef struct _COS_EVENT
{
        UINT32 nEventId;
        UINT32 nParam1;
        UINT32 nParam2;
        UINT32 nParam3;
} COS_EVENT;

#define COS_WAIT_FOREVER 0xFFFFFFFF
#define COS_NO_WAIT 0x0

#define COS_EVENT_PRI_NORMAL 0
#define COS_EVENT_PRI_URGENT 1

// NOTE: timeout is not implemented
BOOL COS_SendEvent(HANDLE hTask, COS_EVENT *pEvent, UINT32 nTimeOut, UINT16 nOption);

BOOL COS_WaitEvent(HANDLE hTask, COS_EVENT *pEvent, UINT32 nTimeOut);

UINT32 COS_TaskEventCount(HANDLE hTask);

BOOL COS_IsEventAvailable(HANDLE hTask);
```

Callback

```
// ============================================================================
// COS_CALLBACK_FUNC_T
// ----------------------------------------------------------------------------
/// Function pointer type for a typical callback.
// ============================================================================
typedef VOID (*COS_CALLBACK_FUNC_T)(void *param);

// ============================================================================
// Task Callback
// ----------------------------------------------------------------------------
/// Run callback in task at next event loop, with one parameters.
```

```
/// They are implemented with EVENT, and the callback will be executed in
/// COS_WaitEvent. They are easier to be used that COS_SendEvent due to it is
/// not needed to define EVENT ID, and it is not needed to expose functions
/// to event loop.
///
/// In case that the callback is droppable, and it maybe sent frequently,
/// COS_TaskCallbackNotif will remove previous ones in queue before send.
/// ATTENTION: remove previous ones will take time, don't call it in ISR.
// ============================================================================
BOOL COS_IsTaskCallbackSet(HANDLE hTask, COS_CALLBACK_FUNC_T callback, void *param);
BOOL COS_TaskCallback(HANDLE hTask, COS_CALLBACK_FUNC_T callback, void *param);
BOOL COS_TaskCallbackNotif(HANDLE hTask, COS_CALLBACK_FUNC_T callback, void *param);
BOOL COS_StopTaskCallback(HANDLE hTask, COS_CALLBACK_FUNC_T callback, void *param);

// ============================================================================
// SYNC callback
// ----------------------------------------------------------------------------
/// Run callback in task at next event loop, with one parameters. And it will
/// wait the callback is executed. When the target task is current task,
/// the callback will be called directly.
///
/// callback can be NULL. It can be used to wait all previous events are handled
/// in the target task.
// ============================================================================
BOOL COS_TaskCallbackSync(HANDLE hTask, COS_CALLBACK_FUNC_T callback, void *param);
BOOL COS_TaskWaitEventHandled(HANDLE hTask);

// ============================================================================
// Callback Timer
// ----------------------------------------------------------------------------
/// Run callback in task after specified period. They are implemented with TIMER,
/// and the callback will be executed in COS_WaitEvent. They are easier to be
/// used that COS_StartTimer due to it is not needed to define timer ID, and it
/// is not needed to expose functions to event loop.
///
/// Flags:
/// * FORCEDLY/CHECK: normally, existed timer will be removed before start.
///         When FORCEDLY is set, existed timer won't be checked. When CHECK is
///         is set, start timer only if the timer is not existed. These 2 flags
///         shouldn't be set simultaneously.
/// * PM1/PM3: normally, timer will be ignored in PM1 and PM3. If timer timeout
///         is really needed, these 2 flags will enable timer timeout even in
///         PM1/PM3. These 2 flags can be set simultaneously.
// ============================================================================
#define COS_TIMER_FLAG_START_FORCEDLY (1 << 0)
#define COS_TIMER_FLAG_START_CHECK (1 << 1)
#define COS_TIMER_FLAG_PM1 (1 << 8)
#define COS_TIMER_FLAG_PM3 (1 << 9)

BOOL COS_IsCallbackTimerSet(HANDLE hTask, COS_CALLBACK_FUNC_T callback, void *param);
BOOL COS_StartCallbackTimer(HANDLE hTask, UINT32 ms, COS_CALLBACK_FUNC_T callback,␣
↪void *param);
BOOL COS_StartCallbackFlagTimer(HANDLE hTask, UINT32 ms, UINT32 flags, COS_CALLBACK_
↪FUNC_T callback, void *param);
BOOL COS_StopCallbackTimer(HANDLE hTask, COS_CALLBACK_FUNC_T callback, void *param);
```

Function Timer

```
// ============================================================================
// Function Timer
// ----------------------------------------------------------------------------
/// Function timer will be called inside OS timer IRQ. So, it must be "short".
// ============================================================================
BOOL COS_IsFunctionTimerSet(COS_CALLBACK_FUNC_T callback, void *param);
VOID COS_StartFunctionTimer(UINT32 ms, COS_CALLBACK_FUNC_T callback, void *param);
VOID COS_StartFunctionTimerForcedly(UINT32 ms, COS_CALLBACK_FUNC_T callback, void␣
↪*param);
VOID COS_StartFunctionTimerCheck(UINT32 ms, COS_CALLBACK_FUNC_T callback, void␣
↪*param);
VOID COS_StopFunctionTimer(COS_CALLBACK_FUNC_T callback, void *param);
```

Common Timer

```
#define COS_TIMER_MODE_SINGLE 0
#define COS_TIMER_MODE_PERIODIC 1

//
// TimerID ranges
//
#define COS_MMI_TIMER_ID_BASE 0x000
#define COS_MMI_TIMER_ID_END_ 0x3FF
#define COS_CSW_TIMER_ID_BASE 0x400
#define COS_CSW_TIMER_ID_END_ 0x4FF
#define COS_BT_TIMER_ID_BASE 0x500
#define COS_BT_TIMER_ID_END_ 0x5FF
#define COS_MDI_TIMER_ID_BASE 0x600
#define COS_MDI_TIMER_ID_END_ 0x6FF

#define CFW_AT_UTI_RANGE 32

BOOL COS_SetTimerEX(HANDLE hTask,
                                        UINT16 nTimerId,
                                        UINT8 nMode,
                                        UINT32 nElapse);
BOOL COS_KillTimerEX(HANDLE hTask, UINT16 nTimerId);
UINT32 COS_QueryTimerEX(HANDLE hTask, UINT16 nTimerId);
BOOL COS_ForceDeliverTimerEx(HANDLE hTask, UINT16 nTimerId);
```

Semaphore

```
// ============================================================================
// COS_SEMA
// ----------------------------------------------------------------------------
/// COS_SEMA is a (limited) counting semaphore.
// ============================================================================
struct _COS_SEMA;
typedef struct _COS_SEMA COS_SEMA;
#define COS_SEMA_UNINIT \
        {                   \
        }
```

```
BOOL COS_SemaInited(COS_SEMA *sema);
VOID COS_SemaInit(COS_SEMA *sema, UINT32 init);
VOID COS_SemaTake(COS_SEMA *sema);
BOOL COS_SemaTryTake(COS_SEMA *sema, UINT32 ms);
VOID COS_SemaRelease(COS_SEMA *sema);
VOID COS_SemaDestroy(COS_SEMA *sema);
```

Mutex

```
// ============================================================================
// COS_MUTEX
// ----------------------------------------------------------------------------
/// COS_MUTEX is just a mutex.
// ============================================================================
struct _COS_MUTEX;
typedef struct _COS_MUTEX COS_MUTEX;
#define COS_MUTEX_UNINIT \
        {                \
        }

BOOL COS_MutexInited(COS_MUTEX *mutex);
VOID COS_MutexInit(COS_MUTEX *mutex);
VOID COS_MutexLock(COS_MUTEX *mutex);
BOOL COS_MutexTryLock(COS_MUTEX *mutex, UINT32 ms);
VOID COS_MutexUnlock(COS_MUTEX *mutex);
VOID COS_MutexDestroy(COS_MUTEX *mutex);
```

Cache

```
void COS_CleanDCache();
void COS_CleanICache();
void COS_CleanALLCache();
```

Other

```
// Current task will sleep for specific time
BOOL COS_Sleep(UINT32 nMillisecondes);

// Enter/exit critical section. Must be **paired** and
// do least necessary things during the critical section.
HANDLE COS_EnterCriticalSection(VOID);
BOOL COS_ExitCriticalSection(HANDLE hSection);
```

5.1.2 1 Sample Code (Examples)

This page list some typical examples for fresh developers.

1.1 How to create a task?

```
#include <cos.h>

#define SAMPLE_TASK_STACK_SIZE (1024)
// task priority must < 255: the smaller, the higher
#define SAMPLE_TASK_PRIORITY (253)

// first we define a task entry point
static void sample_task_entry()
{
    HANDLE task = COS_GetCurrentTaskHandle();

    // often the entry body is a dead loop
    while(1)
    {
        COS_EVENT event = {};

        // wait an event coming
        COS_WaitEvent(task, &event, COS_WAIT_FOREVER);

        ... ...
    }
}

HANDLE sample_task_handle = COS_CreateTask(sample_task_entry, NULL, NULL,
                                           SAMPLE_TASK_STACK_SIZE, SAMPLE_TASK_
↪PRIORITY,
                                           COS_CREATE_DEFAULT, 0, "sample");

// Attention: unless we use flag COS_CREATE_SUSPENDED as the parameter at task␣
↪creating,
// the task will run automatically after created.
// If we want to start the task later in specific time, pass flag COS_CREATE_SUSPENDED
// and then call COS_StartTask((TASK_HANDLE *)sample_task_handle, NULL)
```

1.2 How to send event between tasks?

Event is the communication method for tasks. The event is defined by `COS_EVENT`, which consists of one event id and three parameters, four 32bit unsigned integers in all. NOTE: timeout mechanism is not implemented, so always pass *COS_WAIT_FOREVER* when send event

```
#include <cos.h>

#define SAMPLE_TASK_STACK_SIZE (1024)
// task priority must < 255: the smaller, the higher
#define SAMPLE_TASK_PRIORITY (253)

#define EVENT_ID_TASK1_TO_TASK2 0x1
#define EVENT_ID_TASK2_TO_TASK1 0x2

HANDLE sample_task1_handle;
HANDLE sample_task2_handle;
```

```
static void sample_task1_entry()
{
    HANDLE task = COS_GetCurrentTaskHandle();

    while(1)
    {
        COS_EVENT event = {EVENT_ID_TASK1_TO_TASK2, 0, 0, 0};

        // send event to other task
        COS_SendEvent(sample_task2_handle, &event, COS_WAIT_FOREVER, COS_EVENT_PRI_
↪NORMAL);

        // wait an event coming
        COS_WaitEvent(task, &event, COS_WAIT_FOREVER);

        ... ...

    }
}

static void sample_task2_entry()
{
    HANDLE task = COS_GetCurrentTaskHandle();

    while(1)
    {
        COS_EVENT event = {};

        // wait an event coming
        COS_WaitEvent(task, &event, COS_WAIT_FOREVER);

        ... ...

        event = {EVENT_ID_TASK2_TO_TASK1, 0, 0, 0};

        // send event to other task
        COS_SendEvent(sample_task1_handle, &event, COS_WAIT_FOREVER, COS_EVENT_PRI_
↪NORMAL);
    }
}

sample_task1_handle = COS_CreateTask(sample_task1_entry, NULL, NULL,
                                     SAMPLE_TASK_STACK_SIZE, SAMPLE_TASK_PRIORITY,
                                     COS_CREATE_DEFAULT, 0, "sample_task1");

sample_task2_handle = COS_CreateTask(sample_task2_entry, NULL, NULL,
                                     SAMPLE_TASK_STACK_SIZE, SAMPLE_TASK_PRIORITY,
                                     COS_CREATE_DEFAULT, 0, "sample_task2");
```

5.1.3 SX TIMER

- *Overview*

Overview

SX timer has some uncommon characteristics, in contrast with other RTOS timers.

It divides into two paths:

- `SXR_REGULAR_TIMER`
- `SXR_FRAMED_TIMER`

The former works as ordinary timer, triggered by hardware `OS timer`. The latter is based on the concept of `GSM frame` in GSM protocol. It is more understandable if `SXR_FRAMED_TIMER` is not mixed together.

SX timer can be divided into two categories by handling methods when expire:

- `mailbox timer`
- `function timer`

`mailbox timer` will send event to specific `mailbox` according to `mailbox id`. `function timer` will call a specific function when expire.

TimerEnv

Like other components in SX, the information of timer is stored in a globally pre-allocated memory (i.e. global variables in fact)

- `PeriodToNext`:

  The name is bit misleading, actually it means the relative gap with the former(earlier) timer. For example, there are three active timers and their `PeriodToNext` are 2/3/4. That means timer1 will expire in 2 ticks, timer2 in 5 ticks and timer3 will expire in 9 ticks.

- `Ctx`: with different meaning according to timer types
  - `mailbox timer`: event to be sent when expired, so the size of `Ctx` must be smaller than event's size (4 words).
  - `function timer`: Ctx[0] is the called function pointer. Ctx[1] is the parameter.
- `Id`:
  - bit[15] identify mailbox timer
  - bit[14] identify function timer
  - following bits meaning is different according to its type:

    `mailbox timer`:

    * bit[13] identify expecting wake up in PM1,
    * bit[12] identify expecting wake up in PM3,
    * bit[11:0] mailbox ID

`function timer`:

All means ID and the ID can be used for deleting batch of timers. `SXR_FRAMED_TIMER` use this function, while `SXR_REGULAR_TIMER` not.

Some constraints must be taken into consideration:

- `mailbox timer` can only be searched or deleted by the event
- Only `mailbox timer` supports wake up in PM1/PM3
- Now all bits of `Id` in `function timer` are utilized. Incompatibilities with current code may come if you want to extend the `Id`.

Timer Delete

The logic to delete or find a timer is not intuitive. Here's the definition:

```
typedef struct
{
    u32 Ctx;
    u32 Msk;
    u16 Id;
    u8 Idx;
} sxr_Timer_t;
```

Looks like by two ways from the implementations: matching the ID or Ctx.

when `sxr_Timer_t.Id == 0` matching `Ctx`:

```
(TimerEnv->Ctx[Idx] & Msk) == Ctx
```

when `sxr_Timer.Id != 0` matching `Id`:

```
(TimerEnv->Id & Msk) == Ctx && TimerEnv->Id != 0
```

One limitation of matching `Ctx` is only one word is checked.

Another structure `sxr_TimeEx_t` can match `Id` and `Ctx` at the same time, and more words in `Ctx` can be checked for matching.

Overflow

Overflow is caused by data size limitation:

- SX uses 32bits tick, the maximum is 3 days.
- COS uses 32bits ms, the maximum is 49.7 days. (Now it still uses tick and the max is the same)
- OS Timer uses 24bits tick, the maximum is 17min(1024sec). Large timeout needs multiple interrupts.

PM1, PM3

There's no PM3 in 2G system and the time of PM1 is always not longer than 500ms. So timer expire is not considered. In other word, the OS timer interrupt is masked before entering PM1 and expired timers are checked and processed after PM1 ends.

When it turns to NBIoT system, the period of staying in PM1 requested by protocol might be much longer and the protocol stack may require some timers still need to be handled during PM1 when expire. For this situation, the OS timer interrupt cannot be masked after coming to PM1. In addition, those timers which need to be handled during

PM1 have to be found out and update OS timer to the expire time. It can make sure system can be waken up if that timer expires before LPS wake up. Expired timers then can be processed after exit PM1.

For PM3, the OS timer cannot wake up the system. So those timers must be changed to RTC and let RTC wake the system up.

5.1.4 Clock Framework

Overview

Clock is one of the critical components in SOC and it is also very complex. Different modules may work on different clock rate. The clock rate has closest relationship with system power, performance and stability.

The hardware architecture of clock is commonly designed as a tree. The crystal oscillator is the root and other clocks can be generated by other hardware components, such as pll, doubler, divider etc.

The hardware design varies from one product to another, so it brings some challenges to software developers. A totally general clock framework is not feasible on the modem project, because it needs more overhead which is not negligible for such strict real time required system.

The clock framework on RDA modem platform borrows from linux kernel v3.10 and is implemented to meet some of the features and requirements from other modules.

Main Design Goals

- Use one set of APIs for multiple hardware platforms
- Clearly demonstrate the relationship between clocks
- Manage software application and hardware configuration of clock separately

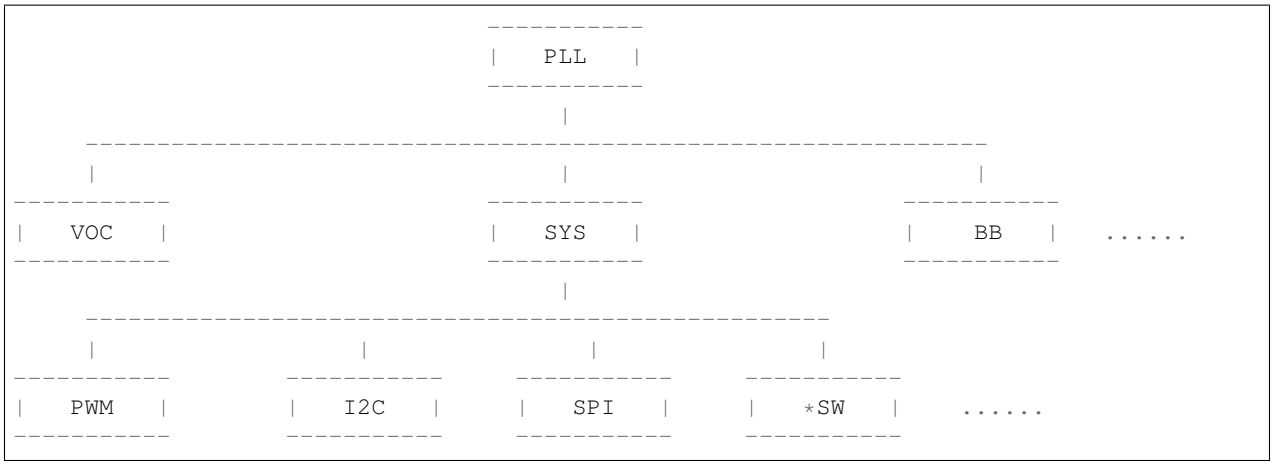
Source Code Tree

```
include
├── hal_clk.h      - header file
src
├── clkdev.c       - clock list management
├── clkdev.h
├── clk.h          - linux styled clock api
├── clock_impl.c   - clock framework implementation
├── clock_impl.h   - internal structure definitions
├── hal_clk_8955.c      - clock objects management and HAL wrapper on 8955
├── 8909/hal_clk_8909.c - clock objects management and HAL wrapper on 8909
```

Key points

- Each clock node uses the same data structure, `struct clk`. All nodes must be created and registered in initializing stage. Runtime creating or deleting is not supported.
- Each node can have one parent node and can be runtime changed. Can have multiple children node.
- A count is used to save node's enabled status when call clock's enable/disable, so calls of enable/disable must be paired. Children node's enable/disable will recursively enable/disable its parent node as well.
- Make sure the node is enabled before altering its clock rate. Once a parent's rate changed, the change will be propagated to its children.
- If a child node is disabled, parent node will scan its children nodes and decide whether to switch a lower rate.
- On RDA platform, `system clock` (aka, clk sys) is an critical node as the clocks of many peripheral components are derived from it. For simplicity, the child node of `clk sys` actually change system clock rate when set its rate. It's a special case, as some of `clk sys`'s children need to work at a specific system clock rate. One node, named `clk sw`, is for software need to change system clock rate according to different user scenarios.

A typical clock tree grown from PLL looks like this:



- Use flag GATE_MANUAL to mark a clock is using manual clock gating. The gate controlling registers will be set at the time clock is enabled or disabled.
- Callback handling: every clock node can assign a callback and it is called when its parent clock rate change. The node must be enabled firstly with one exception when flag IMPLICIT_ENABLE is set, i.e. the callback is always called.
- Some clock may need to switch between a fast clock and a slow clock source. Now the clock node can set two parents, i.e. fast_parent and slow_parent.

API reference

```
// Clock framework initialization, must be called before call any other api
bool hal_ClkInit();

// Get a registered clock object by FOURCC.
//
// @return: NULL if not found and something is wrong
HAL_CLK_T *hal_ClkGet(const uint32_t four_cc);

// Not used since we don't support dynamic clock create or delete
```

```
// and no memory need freed either.
void hal_ClkPut(HAL_CLK_T *clk);

// Enable a clock, and its parent, parent's parent will also be enabled
// recursively if exist. A counter is used internally recording the enabled
// but not disabled times. Therefore, ``Disable`` must be paired with
// ``Enable`` or system cannot go to deep sleep, and this should be guaranteed
// by the caller.
//
// @return: false if enable one clock failed
bool hal_ClkEnable(HAL_CLK_T *clk);

// Disable a clock, must be paired with ``Enable``. Internal count will
// minus one until 0. Can be called even count is 0, but not recommend.
void hal_ClkDisable(HAL_CLK_T *clk);

// Check if a clock is enabled.
//
// @return: true means enabled. false means not enabled yet.
bool hal_ClkIsEnabled(HAL_CLK_T *clk);

// Set the rate of one clock. The clock must be enabled before set rate.
bool hal_ClkSetRate(HAL_CLK_T *clk, uint32_t rate);

// Get the rate of one clock.
uint32_t hal_ClkGetRate(HAL_CLK_T *clk);:we

// dump clock tree
// @param: verbose ``true`` show all clocks, ``false`` just show enabled clock
void hal_clk_dump(bool verbose);
```

Following API are used by applications who need to change system clock rate

```
// Request a system clock rate. FOURCC is passed for identification. One
// application can call multiple times to request different rate on various
// scenarios and it doesn't have to be paired with call of ``release``.
// But ``release`` must be called when not used in the end, or system
// sleep will fail.
bool hal_SwRequestClk(const uint32_t fourCC, uint32_t rate);

// Release the request when not used.
bool hal_SwReleaseClk(const uint32_t fourCC);

// Check if one application is using system clock
bool hal_SwUserIsActive(const uint32_t fourCC);

// Get the requested system clock rate from one application
uint32_t hal_SwGetClkRate(const uint32_t fourCC);
```

CHAPTER

SIX

INDICES AND TABLES

- genindex
- search