AN13232

RT-Thread Board Port and Application Development

Rev. 0 — 04/2021 Application Note

1 Introduction

RT-Thread is a multitasking application development platform integrating Real-Time Operating System (RTOS) kernel, middleware component and developer community. It is developed with the strength of open source community. RT-Thread is also an Internet of things operating system with rich components, highly scalable, simple development, ultra-low power consumption and high security. RT-Thread has all the key components required for an IoT OS platform, such as GUI, network protocol stack, secure transport, low-power components, and so on.

After 11 years of cumulative development, RT-Thread has owned the largest embedded open source community in China and quickly got global interests. RT-Thread has been widely used in energy, vehicle-mounted, medical, consumer electronics and other industries, deployed on more than 800 million devices.

2 Architecture of RT-Thread

One of the main differences between RT-Thread and many other RTOS, such as FreeRTOS and uC/OS, is that it is a real-time kernel and has a rich set of software components, as shown in Figure 1.

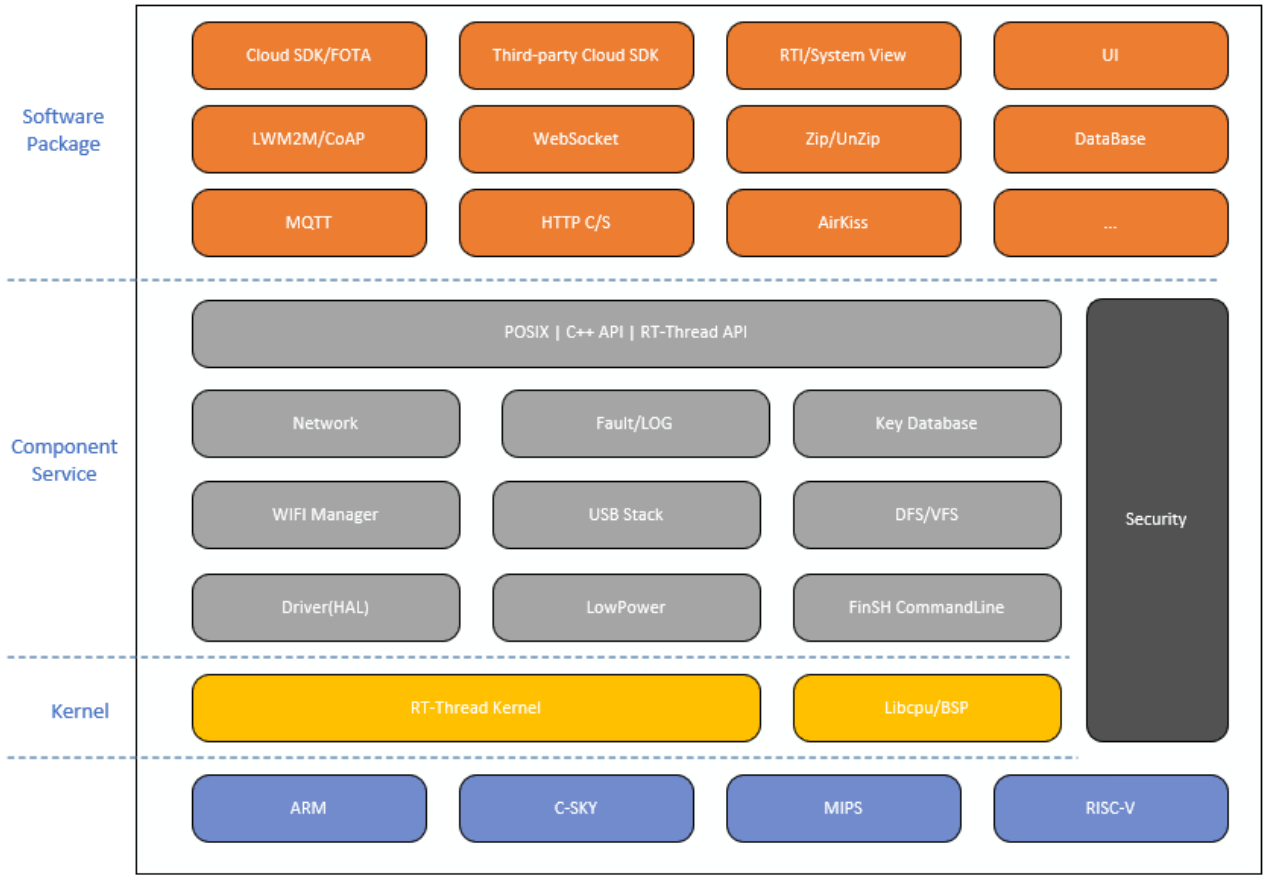

Figure 1. RT-Thread architecture

- Kernel layer: RT-thread kernel is the core part of RT-Thread, including the implementation of objects in the kernel system, such as multi-thread and its scheduling, semaphore, mailbox, message queue, memory management, timer, etc. Libcpu /BSP (chip migration related file/board level support package) is closely related to hardware and consists of peripheral drivers and CPU support.
- Component and Service layer: Components are top-level software based on RT-Thread kernel, such as VFS, FinSH command line interface, network, device framework, and so on. Modular design is adopted to achieve high cohesion and low coupling between components.
- Software package layer:
  - IOT Software package: Paho MQTT, WebClient, mongoose, WebTerminal
  - Scripts Languange: JerryScript, MicroPython
  - Multimedia: muPDF
  - Tools: CmBacktrace, EasyFlash, EasyLogger, SystemView
  - System related Software: RTGUI, Persimmon UI, lwext4, partition, SQLite

3 RT-Thread startup sequence

Figure 2 shows the RT-Thread startup sequence. The colored blocks require special attention, yellow for libCPU porting and green for board porting.

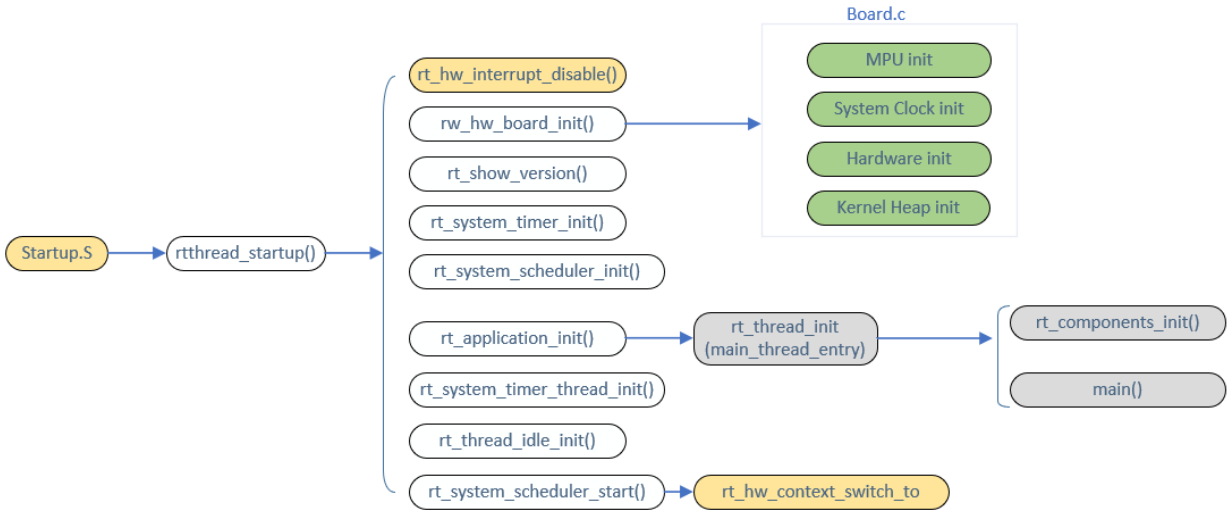

Figure 2. RT-Thread startup sequence

The start entry of RT-Thread is `rtthread_startup()`. After chip startup file completes the hardware initialzation (such as clock configuration, interrupt vector table, initializing heap and stack), jump to the start entry of RT-Thread. The startup sequence for RT-Thread is as follows :

1. Disable global interrupt, initializing the system hardware.
2. Print system information and initialize system or modules such as system tick, scheduler.
3. Initialize the main thread.
4. Initialize software timer thread, idle thread.
5. Start scheduler, enable global interrupt, and switch context to main thread.
6. In the main thread, initialize the components, including drivers, network, vfs, and other component services in programmed order and then enter the main function.

4 Directory structure

In the RT-Thread source code, Figure 3 shows the porting-related files located in the colored path, yellow for libCPU-related files and green for board-related files.

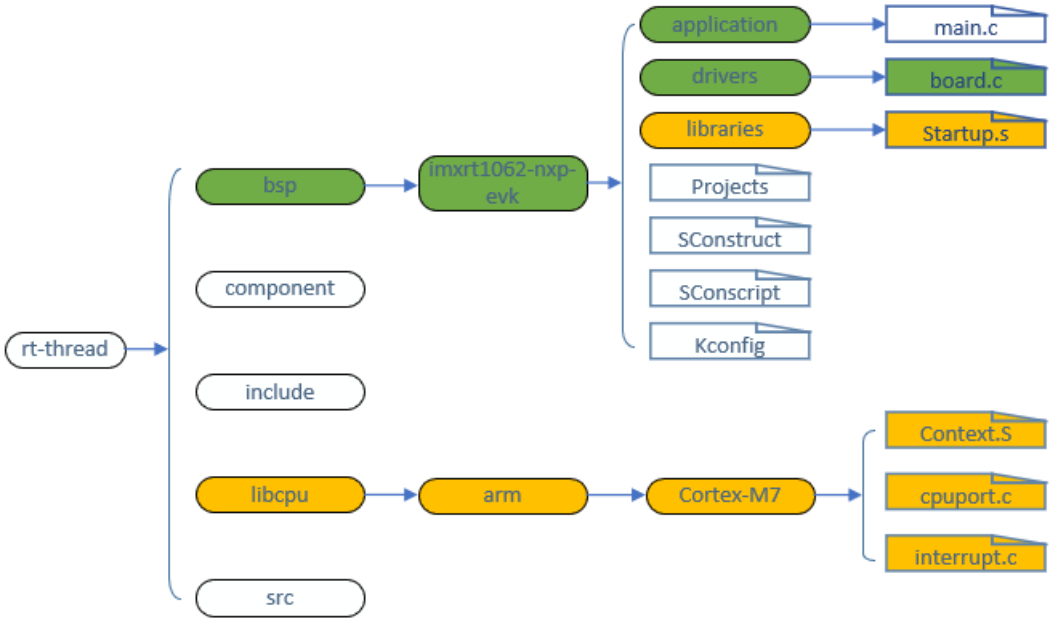

Figure 3. Directory structure

- bsp
  - hardware related files
- component
  - rtthread components such as: finsh, libc, cplusplus, net …
- include
  - rtthread header files
- libcpu
  - cpu related files
- src
  - rtthread kernel source codes

5 Start porting

5.1 Source code

Download source codes from rt-thread

5.2 Libcpu Porting

RT-Thread's libCPU abstraction layer provides a set of unified CPU architecture abstraction interface. This part of the interface contains the global interrupt control function, Thread context switching function, the configurations of clock tick and interrupt function, Cache, and so on.

- Context switch: `context_xx.s`

A context switch represents a CPU switch from one thread to another, or between threads and interrupts, and so on. During a context switch, the CPU typically stops processing the currently running code and saves the exact location where the current program is running so that it can resume later. Table 1 describes key functions to be implemented.

Table 1. Functions to be implemented

`rt_base_t rt_hw_interrupt_disable(void);`	Disable global interrupt
`void rt_hw_interrupt_enable(rt_base_t level);`	Enable global interrupt
`void rt_hw_context_switch_to(rt_uint32 to);`	Switch context, called while starting thread or used in signal
`void rt_hw_context_switch(rt_uint32 from, rt_uint32 to);`	Switch **from** thread to **to** thread
`void rt_hw_context_switch_interrupt(rt_uint32 from, rt_uint32 to);`	Called in interrupt, switch **from** thread to **to** thread

- Thread initialization: `cpuport.c`

  In RT-Thread, the Thread has an independent stack. When the Thread is switched, the context of the current Thread will be stored in the stack. When the Thread wants to resume running, the context information will be read from the stack for recovery.

  This file implements the initialization of the thread stack `rt_hw_stack_init ()` and the hard Fault exception handler. Table 2 describes key functions to be implemented.

Table 2. Functions to be implemented

`rt_hw_stack_init()`	Initializing stack of a thread
`rt_hw_hard_fault_exception()`	Exception handler for hardfault

5.3 Libraries porting

5.3.1 Startup file

Startup.s provided by SDK of chip handles the following tasks:

- Initialize clock and the configuration of interrupt vector.
- Initialize global/static variables.
- Initialize the stack.
- Initialize library functions.
- Jump to next stage startup.

  Under Keil MDK or IAR, without any modifications, the program will jump to `rtthread_startup()`. But under gcc compiler, you need to change **bl main** to **bl entry**.

```
Startup.S
//Before change：
  bl  SystemInit
  bl  main
-------------------------------
  //after change：
    bl  SystemInit
    bl  entry
-------------------------------
In main.c
```

```
int entry(void)
{
    rtthread_startup();
    return 0;
}
```

5.4 Drivers porting

5.4.1 RTT device framework

RT-Thread provides a simple I/O device model framework, as shown in Figure 4, between the hardware and the application. It falls into three layers, from top to bottom, I/O device interface layer, device driver framework layer (HAL), and BSP driver layer.

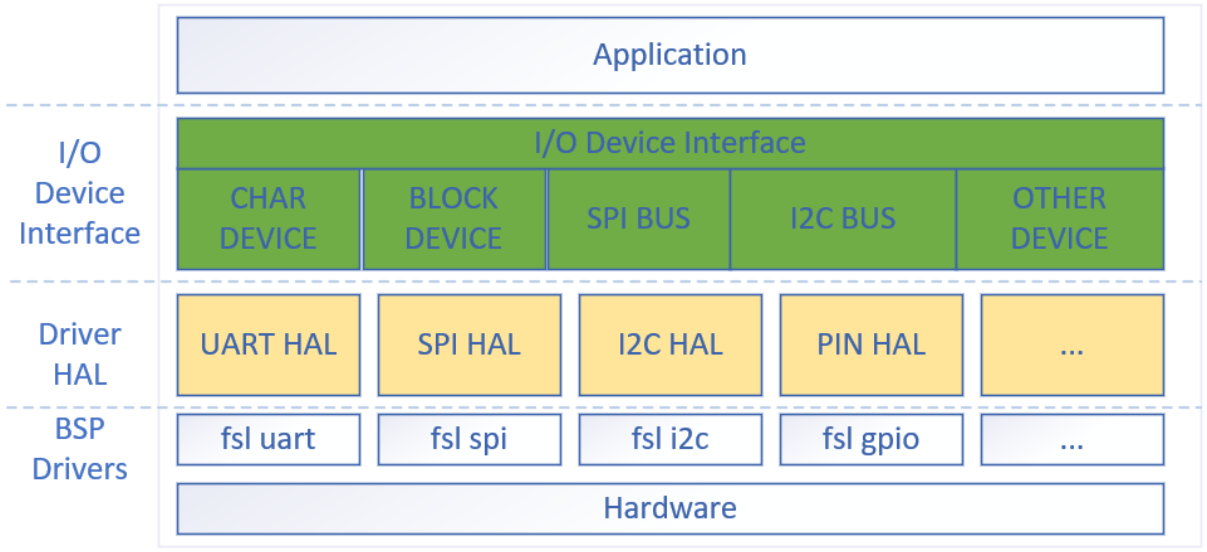

Figure 4. I/O device model framework

The device driver layer is a set of programs that drive hardware devices to work and provide the functions to access hardware devices. It is responsible for creating and registering I/O devices. For devices with simple operation logic (init, read, write, close), the device can be registered directly into the I/O device manager without going through the device driver framework (HAL) layer. Figure 5 shows the sequence diagram.

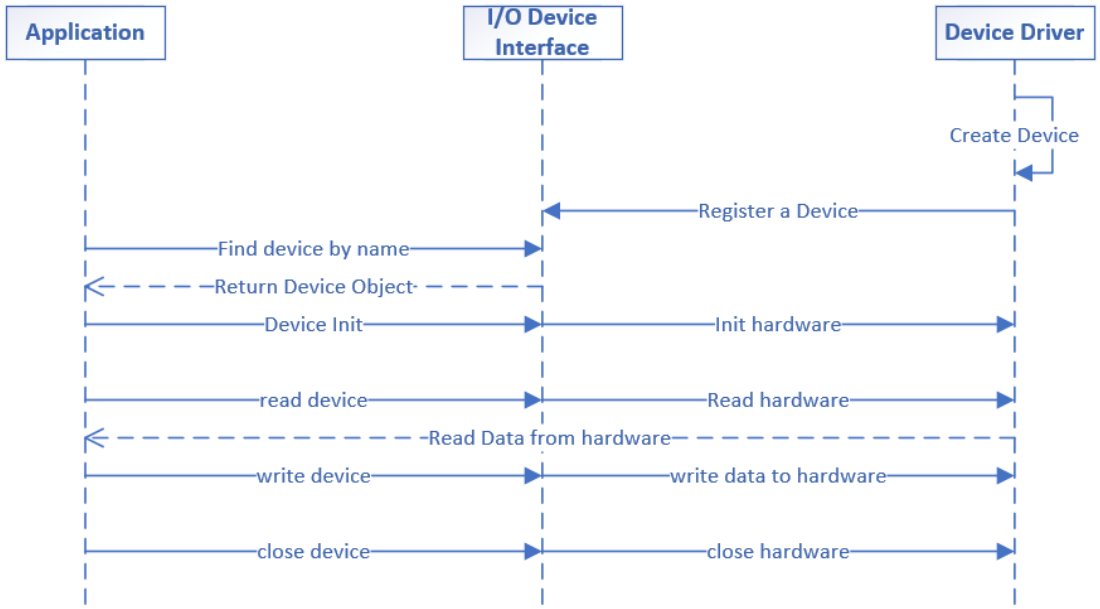

Figure 5. Sequence diagram

5.4.2 RTT device structure

RT-Thread's device model is based on the kernel object model. Devices are considered as a class of objects and are included in the category of object manager. Each device object is derived from the base object, and each concrete device can inherit the properties of its parent class object and derive its new properties. Figure 6 shows the inheritance and derivation relationship of device object.

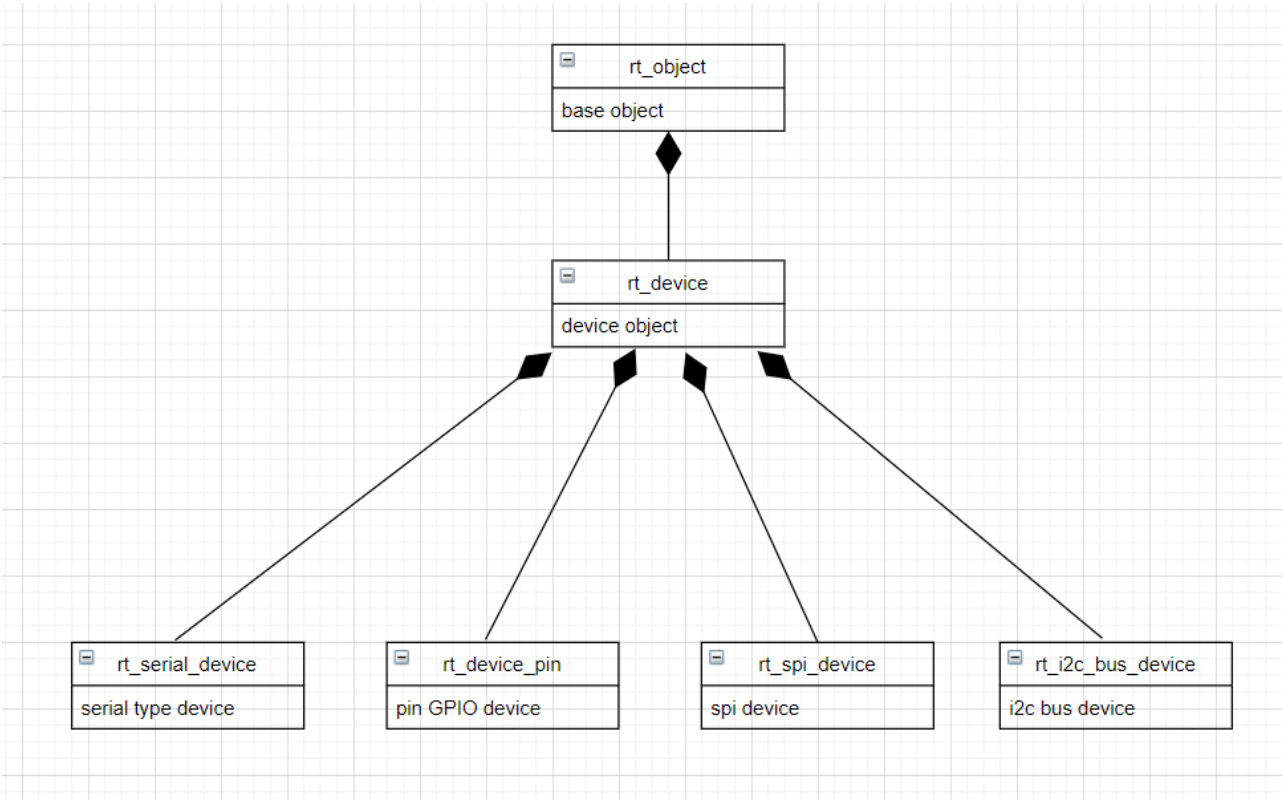

Figure 6. Inheritance and derivation relationship of device object

The device object is specifically defined as follows:

```
struct rt_device
{
    struct rt_object          parent;        /* kernel base object */
    enum rt_device_class_type type;          /* device type */
    rt_uint16_t               flag;          /* device param */
    rt_uint16_t               open_flag;     /* device open flag */
    rt_uint8_t                ref_count;     /* device open count */
    rt_uint8_t                device_id;     /* device ID,0 - 255 */

    /* data transfer call back */
    rt_err_t (*rx_indicate)(rt_device_t dev, rt_size_t size);
    rt_err_t (*tx_complete)(rt_device_t dev, void *buffer);

    const struct rt_device_ops *ops;    /* device operator pointer*/

    /* device data */
    void *user_data;
};
```

RT-Thread supports a variety of I/O device types. The main device types are as follows:

```
RT_Device_Class_Char             /* char device       */
RT_Device_Class_Block            /* block device         */
RT_Device_Class_NetIf            /* network device    */
RT_Device_Class_MTD              /* MTD device       */
RT_Device_Class_RTC              /* RTC device        */
RT_Device_Class_Sound            /* sound device        */
```

```
RT_Device_Class_Graphic          /* graphic device        */
RT_Device_Class_I2CBUS           /* I2C bus device     */
RT_Device_Class_USBDevice        /* USB device  */
RT_Device_Class_USBHost          /* USB host   */
RT_Device_Class_SPIBUS           /* SPI bus     */
RT_Device_Class_SPIDevice        /* SPI device        */
RT_Device_Class_SDIO             /* SDIO device       */
RT_Device_Class_Miscellaneous    /* miscellaneous device        */
```

The application accesses the hardware device through the I/O device operator interface, which must be implemented by a underlying device driver. Figure 7 shows the mapping between the I/O device interface and the operation method.

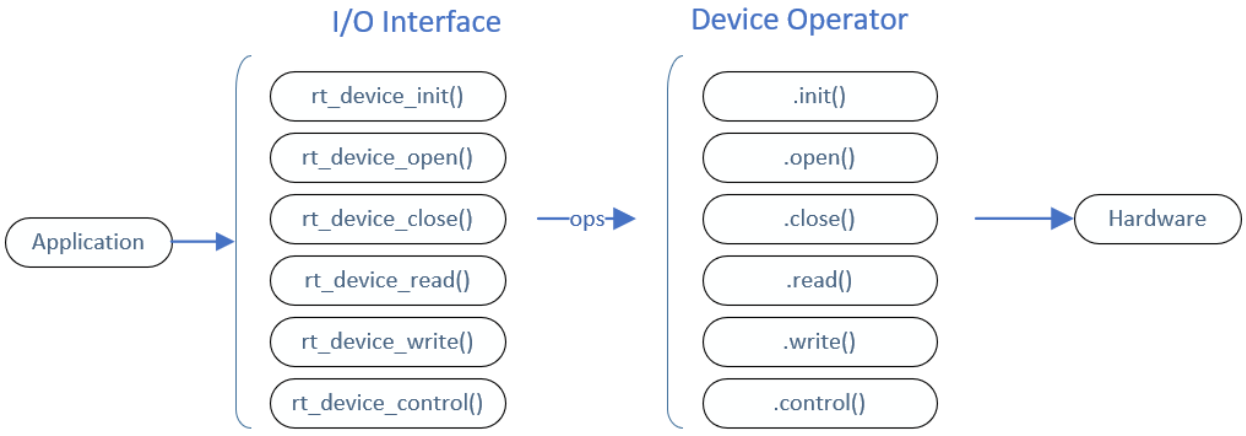

Figure 7. Mapping between I/O device interface and operation method

5.4.3 Example: A simple UART driver

Figure 8 shows an code example of a simple UART driver.

```
138 int rt_simple_uart_init(void)
139 {
140     int i;
141     rt_uint32_t flag;
142     rt_err_t ret = RT_EOK;
143     struct rt_device *device;
144     static struct serial_configure config = RT_SERIAL_CONFIG_DEFAULT;
145
146     flag = RT_DEVICE_FLAG_RDWR | RT_DEVICE_FLAG_INT_RX;
147     device = rt_device_create(RT_Device_Class_Char, 1);
148     if (device == NULL)
149         return RT_ERROR;
150
151     device->init        = s_uart_init;
152     device->open        = s_uart_open;
153     device->close       = s_uart_close;
154     device->read        = s_uart_read;
155     device->write       = s_uart_write;
156     device->control     = s_uart_control;
157     device->user_data = &config;
158
159     /* register a character device */
160     ret = rt_device_register(device, "s_uart", flag);
161
162     return ret;
163 }
164 /* board init routines will be called in board_init() function */
165 INIT_BOARD_EXPORT(rt_simple_uart_init);
```

Figure 8. Example code of a simple UART driver

In the `rt_simple_uart_init()` function, the `s_uart` device is initialized and registered.

```
32  rt_err_t  s_uart_init(rt_device_t dev)
33  {
34      struct imxrt_uart *uart;
35      lpuart_config_t config;
36
37      RT_ASSERT(dev != RT_NULL);
38      struct serial_configure *cfg = (struct serial_configure *)dev->user_data;
39      RT_ASSERT(cfg != RT_NULL);
40
41      LPUART_GetDefaultConfig(&config);
42      config.baudRate_Bps = cfg->baud_rate;
43
44      switch (cfg->data_bits)
45      {
55      switch (cfg->stop_bits)
56      {
65      switch (cfg->parity)
66      {
78      config.enableTx = true;
79      config.enableRx = true;
80
81      LPUART_Init(LPUART1, &config, GetUartSrcFreq());
82
83      return RT_EOK;
84  }
```

Figure 9.

In `s_uart_init()` function, LPUART hardware is initialized. This function is called by `rt_device_init` when user code initializes the board with the `rt_components_board_init()` function.

Figure 10 shows the data read from UART.

```
rt_size_t s_uart_read(rt_device_t dev, rt_off_t pos, void *buffer, rt_size_t length)
{
    int ch;
    int size;
    rt_uint8_t *data = (rt_uint8_t *)buffer;
    RT_ASSERT(dev != RT_NULL);
    size = length;

    while (length)
    {
        if (LPUART_GetStatusFlags(LPUART1) & kLPUART_RxDataRegFullFlag)
        {
            ch = LPUART_ReadByte(LPUART1);
        }
        if (ch == -1) break;

        *data = ch;
        data ++; length --;

        if (ch == '\n') break;
    }

    return size - length;
}
```

Figure 10. Data read from UART

In the `s_uart_read` function, the `s_uart` device reads data from hardware and this function is called in `rt_device_read`.

UART write is similar too.

```
rt_size_t s_uart_write(rt_device_t dev, rt_off_t pos, const void *buffer, rt_size_t length)
{
    int size;
    RT_ASSERT(dev != RT_NULL);
    rt_uint8_t *data = (rt_uint8_t *)buffer;
    size = length;
    while (length)
    {
        if (*data == '\n' && (dev->open_flag & RT_DEVICE_FLAG_STREAM))
        {
            LPUART_WriteByte(LPUART1, '\r');
            while (!(LPUART_GetStatusFlags(LPUART1) & kLPUART_TxDataRegEmptyFlag));
        }
        LPUART_WriteByte(LPUART1, *data);
        while (!(LPUART_GetStatusFlags(LPUART1) & kLPUART_TxDataRegEmptyFlag)){};

        ++ data;
        -- length;
    }

    return size - length;
}
```

Figure 11. Data write from UART

In the `s_uart_write` function, the `s_uart` writes data to hardware and this function is called in `rt_device_write`.

5.5 Board porting

Board level hardware resource initialization is realized by the `RT_HW_BOARD_INIT` function, which completes the initialization of system devices, such as:

- MPU configuration
- Pin function configuration
- System clock configuration
- kernel heap initialization

- Component board initialization

5.6 Project construction

`Env` is a development tool launched by RT-Thread and can be downloaded on RT-Thread Env. It provides compilation and build environment, graphical system configuration and package management functions for projects based on RT-Thread operating system. Its built-in `menuconfig` provides easy-to-use configuration tools and can configure the kernel, components and software packages.

Key features of RT-Thread construction tool include:

- `Menuconfig`: A **graphical** configuration interface, good interactivity. On exit, it generates `rtconfig.h` automatically.
- A variety of highly reliable, modular software packages that are loosely coupled, good maintained. The software package can be downloaded online by `env` tool.
- `Scons`: The default build tool, easy to use. `Scons` can both generate projects for IAR, MDK and invoke the GCC tool chain to build.

SCONS is an open source build system written in Python similar to **GNU Make**. It takes a different approach: instead of processing a Makefile, it uses *SConstruct* and *SConscript* files to guide the build process. These files are also Python scripts that can be written using standard Python syntax. Therefore, you can call the Python standard library in *SConstruct* and *SConscript* files for all kinds of complex processing, not limited to the rules set by the Makefile. For more documents about Scons, see Reference.

SCONS uses *SConscript* and *SConstruct* files to organize the source code structure. Typically, there is only one *SConstruct* file for a project, but there can be multiple *SConscript* files. In general, there is a *SConscript* file in each subdirectory where the source code is stored. SConscript files are the backbone of the organization's source code.

To make it easier for RT-Thread to support multiple compilers and to adjust compilation parameters, RT-Thread creates a separate file named rtconfig.py for each BSP. Therefore, each RT-thread BSP directory contains the following three files: rtconfig.py, SConstruct, and SConscript, which control the compilation of the BSP. There is only one SConstruct file in a BSP but multiple SConscript files. As shown in Directory structure, the project SConstruct SConsript Kconfig files are located in `imxrt1062-nxp-evk`.

RT-Thread has *Sconscript* files in most source folders. These script files are linked with the Sconscript of BSP directory to add the source code for the macros defined in *rtconfig.h* to the compiler. We will take **imxrt1062-nxp-evk BSP** as an example to explain how to build a project with SCONS in the below.

5.6.1 SConstruct

As mentioned above, BSP has only one *SConstruct* file which controls the compilation process. Figure 12 shows an example of SConstruct.

```
import os
import sys
import rtconfig

if os.getenv('RTT_ROOT'):
    RTT_ROOT = os.getenv('RTT_ROOT')
else:
    RTT_ROOT = os.path.normpath(os.getcwd() + '/../../..')

sys.path = sys.path + [os.path.join(RTT_ROOT, 'tools')]
try:
    from building import *
except:

TARGET = 'rtthread.' + rtconfig.TARGET_EXT
DefaultEnvironment(tools=[])
if rtconfig.PLATFORM == 'armcc':
else:

env.PrependENVPath('PATH', rtconfig.EXEC_PATH)

if rtconfig.PLATFORM == 'iar':

Export('RTT_ROOT')
Export('rtconfig')

# prepare building environment
objs = PrepareBuilding(env, RTT_ROOT, has_libcpu=False)

objs = objs + SConscript('../libraries/drivers/SConscript')
objs = objs + SConscript('../libraries/drivers/wlan/SConscript')
objs = objs + SConscript('../libraries/MIMXRT1062/SConscript')
objs = objs + SConscript('../libraries/sensors/SConscript')
objs = objs + SConscript('../components/SConscript')
objs = objs + SConscript('../ml_demos/SConscript')
# make a building
DoBuilding(TARGET, objs)
```

Figure 12. Example of SConstruct

5.6.2 SConscript

Sconscript files connect all the source files, almost every source directory has one *Sconscript* file. There are two typical usages for SConscript files.

- The *SConcript* file in the **imxrt1062-nxp-evk BSP** directory manages all other *SConscript* files under the BSP, as shown in Figure 13.

```
import os
from building import *

cwd = GetCurrentDir()
objs = []
list = os.listdir(cwd)

for d in list:
    path = os.path.join(cwd, d)
    if os.path.isfile(os.path.join(path, 'SConscript')):
        objs = objs + SConscript(os.path.join(d, 'SConscript'))

Return('objs')
```

Figure 13. SConscript files

As shown in Figure 13, it involves all the *SConstruct* files of its subdirectories.

- The *SConcript* file in the **Application** directory manages the source code under the **Application** directory.

```
import rtconfig
from building import *

cwd = GetCurrentDir()
src = Glob('main.c')
src += Glob('app_simple_uart.c')
src += Glob('simple_uart_driver.c')
CPPPATH = [cwd]

# add for startup script
LOCAL_CCFLAGS=''
if rtconfig.CROSS_TOOL == 'gcc':
    LOCAL_CCFLAGS = ' -std=c99 __START=entry'
elif rtconfig.CROSS_TOOL == 'keil':
    LOCAL_CCFLAGS = ' --c99 --gnu'

group = DefineGroup('Applications', src, depend = [''], CPPPATH = CPPPATH)

Return('group')
```

Figure 14. Sconscript file in Application directory

As shown in Figure 14, the script creates a group named **Application**, including **main.c**, **app_simple_uart.c** and **simple_uart_driver.c**.

For complex and large systems, it is obvious that more than just a few files in a directory are required. It is likely to be composed of several folders level by level.

In SCons, you can write **SConscript** scripts to compile files in these relatively separate directories and the **Export and Import** functions in SCons to share data between *SConstruct* and *SConscript* files (that is an object data in Python). For more information on how to use Scons, see SCons document.

5.6.3 Rtconfig.py

Rtconfig.py is a standard compiler configuration file for RT-Thread that controls most of the compilation options. It is a script file written in Python that performs the following:

- Specify a compiler (choose the one you are using from the supported compilers).
- Specify compiler parameters, such as compile options, link options, and so on.

When compiling a project using the scons command, we compile the project according to the compiler configuration options of rtconfig.py. The following code is part of the code for *rtconfig.py* in the **imxrt1062-nxp-evk BSP** directory.

```
import os

# toolchains options
ARCH='arm'
CPU='cortex-m7'
CROSS_TOOL='gcc'

if os.getenv('RTT_CC'):
    CROSS_TOOL = os.getenv('RTT_CC')
if os.getenv('RTT_ROOT'):
    RTT_ROOT = os.getenv('RTT_ROOT')

# cross_tool provides the cross compiler
# EXEC_PATH is the compiler execute path, for example, CodeSourcery, Keil MDK, IAR
if  CROSS_TOOL == 'gcc':
    PLATFORM    = 'gcc'
    EXEC_PATH   = r'C:\Users\XXYYZZ'
elif CROSS_TOOL == 'keil':
    PLATFORM    = 'armcc'
    EXEC_PATH   = r'C:/Keil_v5'
    #EXEC_PATH   = r'D:/Keil_v5'
elif CROSS_TOOL == 'iar':
    PLATFORM    = 'iar'
    EXEC_PATH   = r'C:/Program Files (x86)/IAR Systems/Embedded Workbench 8.1'

if os.getenv('RTT_EXEC_PATH'):
    EXEC_PATH = os.getenv('RTT_EXEC_PATH')

#BUILD = 'debug'
BUILD = 'release'
```

Figure 15. Code for rtconfig.py

5.6.4 Kconfig

`Kconfig` is used to configure the kernel. The `menuconfig` command generates a configuration interface for users to configure the kernel by reading various `Kconfig` files of the project. The output of menuconfig is the *rtconfig.h*: all configurations related macro definitions will be automatically saved to the *rtconfig.h* file in the BSP directory. Each BSP has a *rtconfig.h* file, which is the configuration information of the board.

After entering the **imxrt1062-nxp-evk BSP** directory with the `Env` tool, you can see the configuration menu of the main page with the `menuconfig` command, as shown in Figure 16.

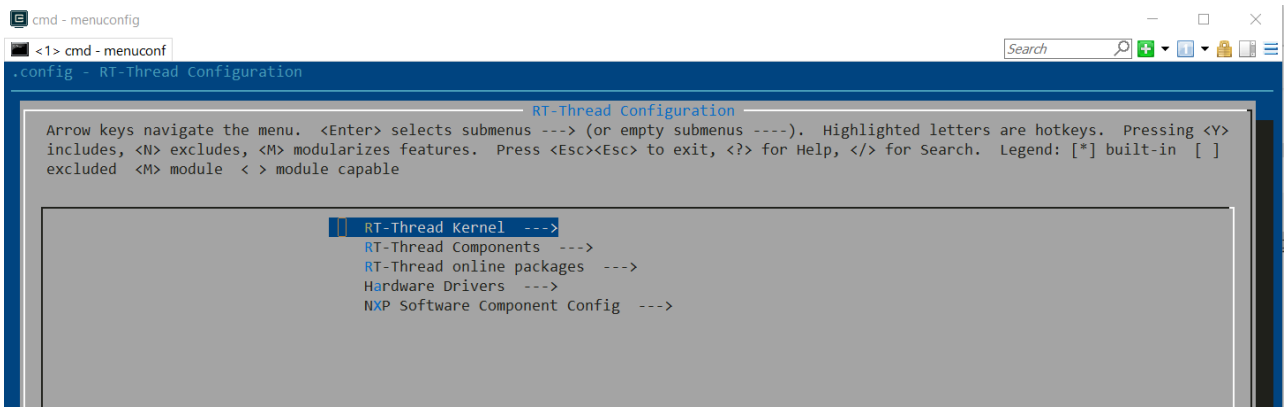

Figure 16. Configuration menu

Save the configurations, exit the configuration interface and open the *rtconfig.h* file under the **imxrt1062-nxp-evk** directory. All the configuration information is already available.

NOTE

DO NOT modify it manually.

Make sure to use the `scons --target=mdk5` command to generate a new KEIL project every time when the `menuconfig` configurations is complete.

6 How to implement application

In the RT-Thread source code, the application locates in the **Application** directory. For example, to create a demo code to use the simple **uart** driver, name it as **app_simple_uart.c** and put it into the **Application** directory. Figure 17 shows the code.

```c
#include <rtthread.h>
#include "rthw.h"
#include <rtdevice.h>

void app_simple_uart()
{
    char buffer[128];
    char test_str[] = "\r\n[Hello World!]\r\n";
    rt_device_t device = rt_device_find("s_uart");
    if (device == RT_NULL) {
        return;
    }
    if (device->open(device, 0) != RT_EOK) {
        return;
    }
    if (device->init(device) != RT_EOK) {
        return;
    }

    device->write(device, 0, test_str, sizeof(test_str));

    device->close(device);
}
//register app_simple_uart into msh command
MSH_CMD_EXPORT(app_simple_uart, app_simple_uart);
```

Figure 17. Demo code

As shown in Figure 17, there are two methods to call application API in RT-Thread:

- Call API in the main thread
- Call API in the `msh` shell command

7 Build and run

After implementing the application codes, add the file into the project. As mentioned in Project construction, RT-Thread uses **scons** to generate the project file, so we need to add file into SConscript in the Application folder.

```
import rtconfig
from building import *

cwd = GetCurrentDir()
src = Glob('main.c')
src += Glob('app_simple_uart.c')
src += Glob('simple_uart_driver.c')
CPPPATH = [cwd]

# add for startup script
LOCAL_CCFLAGS=''
if rtconfig.CROSS_TOOL == 'gcc':
    LOCAL_CCFLAGS = ' -std=c99 __START=entry'
elif rtconfig.CROSS_TOOL == 'keil':
    LOCAL_CCFLAGS = ' --c99 --gnu'

group = DefineGroup('Applications', src, depend = [''], CPPPATH = CPPPATH)

Return('group')

```

Figure 18. Add file in Application folder

Use `env` tool to generate *keil* project by the `scons --target=mdk5` command.

In the project, `app_simple_uart.c` and `simple_uart_driver.c` are included in **Applications** group.

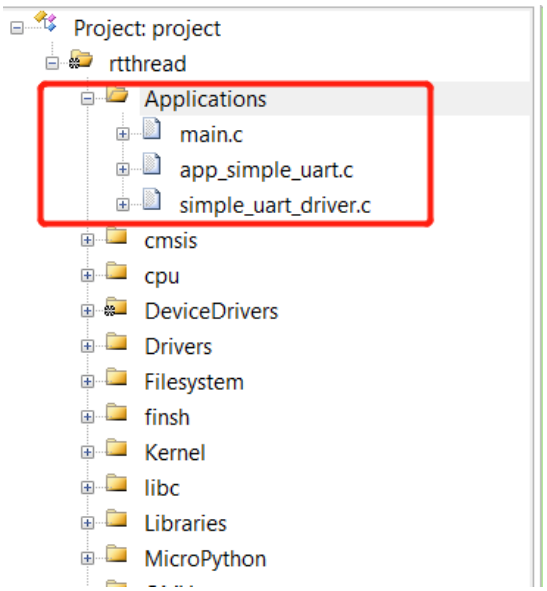

Figure 19. Applications group

After building the project, download the program by jLink and run. As shown in Figure 20, the program runs into the `msh` command line. Type the **app_simple_uart** command to run the example. The log shows the result: the test string came out through UART hardware.

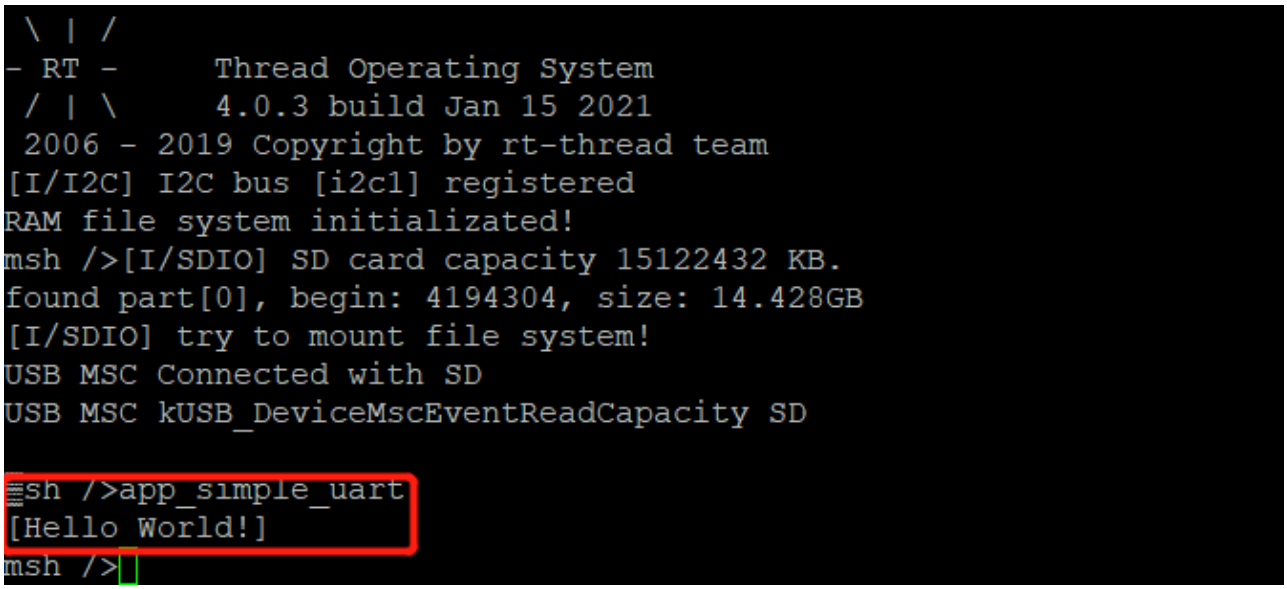

Figure 20. Log

8 Reference

- RT-Thread document
- SCons document

9 Revision history

Revision number	Date	Substantive changes
0	04/2021	Initial release

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

For more information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 04/2021

Document identifier: AN13232

arm