kim-os Documentation

Release 1.0

Ing. Aurelio Colosimo

May 22, 2020

TABLE OF CONTENTS:

CHAPTER

ONE

WHAT IS KIM-OS

KIM-OS (briefly, KIM) stands for "**K**eep-**I**t-**M**inimal **O**perating **S**ystem". It is a state-machine based operating system targeted to Micro Controller Units. Its architecture is intended to be fully cross-platform, though the MCUs currently supported are all ARM Cortex-M core based.

The basic characteristic of KIM is that it can be compiled with simple, opensource, tools, like *gcc*-based toolchain, `make` and basic `bash` commands. A laptop with your favourite Linux distribution is all you need to compile it.

The idea of starting with this project, and most of its contents, came from my professional activities. I am a freelance firmware engineer, with a background in Linux Embedded. I started to work on MCUs a few years ago, and my first understanding about how to approach was provided by the BATHOS[1] project, which was my first reference for experiments and tests, and is the model I adopted for KIM itself.

When having to release specific firmware for my clients, I began to collect some code snippets and conceptual ideas in a library that was growing over the time, and eventually ended up in KIM project. So, I can say KIM is the sum of all of my experience in microcontrollers, put together in a (hopefully) orderly form.

All of the source code is provided on MIT-style[2] license, so that you can do whatever you want with it, as long as you declare that you are using it (see LICENSE for more information).

[1] **B**orn-**A**gain **T**wo **H**our **O**perating **S**ystem, see https://github.com/a-rubini/bathos

[2] See https://opensource.org/licenses/MIT

CHAPTER

TWO

SOURCE CODE ORGANIZATION

Directly going into the details of the code, this the result of `ls` command run at KIM-OS root folder:

```
$ ls -l
total 60
drwxrwxr-x 4 colosimo colosimo 4096 mag 18 15:32 app
drwxrwxr-x 4 colosimo colosimo 4096 mag 18 15:32 arch
drwxrwxr-x 2 colosimo colosimo 4096 mag 18 15:32 cli
drwxrwxr-x 2 colosimo colosimo 4096 mag 18 15:32 config
-rw-rw-r-- 1 colosimo colosimo  768 mag 18 15:32 COPYING
-rw-rw-r-- 1 colosimo colosimo  176 mag 18 15:32 CREDITS
drwxrwxr-x 2 colosimo colosimo 4096 mag 18 15:32 doc
drwxrwxr-x 2 colosimo colosimo 4096 mag 19 15:32 drivers
drwxrwxr-x 2 colosimo colosimo 4096 mag 18 15:32 include
drwxrwxr-x 2 colosimo colosimo 4096 mag 18 15:32 kernel
drwxrwxr-x 2 colosimo colosimo 4096 mag 18 15:32 lib
-rw-rw-r-- 1 colosimo colosimo 1225 mag 18 15:32 LICENSE
-rwxrwxr-x 1 colosimo colosimo  146 mag 18 15:32 makeall.sh
-rw-rw-r-- 1 colosimo colosimo 3632 mag 18 15:32 Makefile
-rw-rw-r-- 1 colosimo colosimo  326 mag 18 15:32 README
drwxrwxr-x 2 colosimo colosimo 4096 mag 18 15:32 tasks
```

Here is a brief explanation for all of the most relevant files and subdirectories you can find:

- *arch*: contains all the MCU-specific code. At present, it contains two subfolders, *unix* and *arm*: the first is intended to make the firmware compile on Unix systems, simulating the presence of low-level hardware; the second one contains the code for all supported ARM chips;
- *app*: this is the place where your high-level application(s) will reside; any line of code inside this folder should be arch-independent, so that the same application can be compiled for different SoCs;
- *cli*: it is the **C**ommand **L**ine **I**nterface facility; both the programmer and the end user can have a (quite simple) shell-like user experience (e.g. on UART), to debug and interact with the firmware itself; a set of pre-defined commands is provided; in addition, it is easily extensible;
- *config*: contains all available configurations; a configuration file is a *.mk* file included by main Makefile, containing some variables which define the desired compile options;
- *drivers*: the code for peripherals and chips external to the SoC; the code here must be arch-independent, so that the same chip (e.g. a I2C temperature sensor) will be easily usable from whatever arch providing the needed bus (e.g. I2C);

- *kernel*: the core of KIM-OS; contains all the (hardware independent) functions needed to handle tasks and I/O;
- *include*: header files to be included when using the hardware independent code;
- *lib*: some useful generic functions; `k_printf`, a printf-like function, is one of the most important, providing a way of writing formatted strings;
- *tasks*: some generic arch-independent tasks, directly usables and intended as an example about how to write your tasks.

CHAPTER

THREE

QUICK START REFERENCE

The prerequisite to compile kim-os for ARM cores is to have basic tools for development in C language on ARM:

- ARM gcc toolchain, e.g. (in my Ubuntu 18.10 system) gcc-arm-none-eabi package; check and fix `CROSS_COMPILE` variable in `arch/arm/arm.mk` if needed
- `make`
- `git`

If you want to compile the basic kim-os system from scratch, these are the steps:

- go to a suitable local path and download the up-to-date git version:

```
$ cd /your/local/path
$ git clone git@github.com:colosimo/kim-os.git
Cloning into 'kim-os'...
remote: Enumerating objects: 286, done.
remote: Counting objects: 100% (286/286), done.
remote: Compressing objects: 100% (137/137), done.
remote: Total 1537 (delta 157), reused 256 (delta 147), pack-reused 1251
Receiving objects: 100% (1537/1537), 203.59 KiB | 0 bytes/s, done.
Resolving deltas: 100% (851/851), done.
Checking connectivity... done.
```

- go into downloaded directory and compile your config (e.g. `discovery_f407vg` is for STM32 F407 Discovery Board):

```
$ cd kim-os
$ make CONFIG=discovery_f407vg
```

The files `cli.elf`, `cli.hex` and `cli.bin` will be created; their name before prefix is given by the `APP` variable defined in the choosen config (*cli* in our example regarding Discovery Board).

Note: The configuration chosen by `make CONFIG=` is saved in local *.config* file; for this reason, all subsequent compile operations on the same configuration do not need `CONFIG=` to be specified. Before changing from one configuration to another, it is recommended to perform a `make clean` operation.

CHAPTER

FOUR

CORTEX-M CORES: FROM POWER-ON TO RUNNING APPLICATION

I think that one of the most instructive subjects to be mastered by any computer programmer, especially if working in embedded scope, is the boot of a processor. However, most of software APIs and RTOSes (especially those released by hardware vendors) seem to have the goal of hiding that all; the reason is quite simple: chip vendors just need to sell their hardware, so they want to show *how easy* is to make them boot.

On the other side, KIM allows you to fully understand what happens in the low levels and be able to completely handle the boot process. I hope this will be useful, at least from an educational point of view.

Since Cortex-M is currently the only family of chips supported by KIM, I will refer to ARM Cortex-M core boot process. It is based on ISR vector: at each interrupt is assigned an address, which is automatically called when the interrupt happens. In KIM, the ISR vector is defined as a C array of pointers, defined in the files:

- `arch/arm/cpu-cortex-m0/cpu.c` for Cortex-M0
- `arch/arm/cpu-cortex-m3/cpu.c` for Cortex-M3
- `arch/arm/cpu-cortex-m4/cpu.c` for Cortex-M4

Let's have a look at Cortex-M0, which is the basic model:

```
static const void *attr_isrv_sys _isrv_sys[] = {
        /* Cortex-M0 system interrupts */
        STACK_TOP,      /* Stack top */
        isr_reset,      /* Reset */
        isr_none,       /* NMI */
        isr_none,       /* Hard Fault */
        0,              /* Reserved */
        0,              /* Reserved */
        0,              /* Reserved */
        0,              /* Reserved */
        0,              /* Reserved */
        0,              /* Reserved */
        0,              /* Reserved */
        isr_none,       /* SVC */
        0,              /* Reserved */
        0,              /* Reserved */
        isr_none,       /* PendSV */
        isr_systick,    /* SysTick */
};
```

The boot is defined by the first two elements:

- the first one must contain the initial address of Stack Pointer; KIM sets it to the top of available RAM, so that all the RAM is available for the stack; this behaviour can be changed, since `STACK_TOP` is defined in the `reg.h` (register definition) file(s) (e.g. `arch/arm/cpu-cortex-m4/soc-stm32f407xx/include/reg.h`)
- the second element in the above ISR vector is the `isr_reset` function, that is pointer to the function executed at power on or reset.

Actually, ARM expects the ISR vector to reside in the first part of the SoC flash; the definition of where each compiled object will be put is done by the so-called linker script. In KIM, the linker script has extension *.lds*. For instance, the file `arch/arm/cpu-cortex-m4/soc-stm32f407xx/kim.lds` contains the "map" of how we expect the compiled objects be put together in the final binary file.

Let's have a look at the linker script:

```
MEMORY
{
#ifdef SOC_VARIANT_stm32f407vg
        flash : o = 0x08000000, l = 1024k
        sram  : o = 0x20000000, l =  128k
#else
#error Unhandled SOC_VARIANT: $(SOC_VARIANT)
#endif
}
```

In the first part (here above), the MEMORY section, the address and size of RAM and Flash are declared; it is conditioned within C pre-processor `#ifdef` directories, because the KIM Makefile runs it in order to obtain a lds compliant with the specific SoC version (after make, you will find `kim.ldscpp` which is the output of C preprocessor).

```
SECTIONS
{
        .text 0x08000000: {
                *(isrv_sys); /* Interrupt Service Routine Vector - System */
                *(isrv_irq); /* Interrupt Service Routine Vector -␣
↪Peripherals IRQs */
                . = 0x0000240;
                *(.text)
        } > flash
```

The second part declares the "sections": the first one is text, which contains the executable instructions. All ISRs (`isrv_sys` and `isrv_irq`) are located at the beginning of the flash, then all remaining compiled code `*(.text)` follows.

```
. = ALIGN(16);
.etext = .;

.rodata : {
                *(.rodata)
} > flash
```

```
. = ALIGN(16);
_erom = .;

.data : {
        __start_data_flash = LOADADDR(.data);
        __start_data_sram = .;
        *(.data);
        . = ALIGN(4);
        __start_tsks = .;
        *(tsks)
        __stop_tsks = .;
        __start_drvs = .;
        *(drvs);
        __stop_drvs = .;
        __start_devs = .;
        *(devs);
        __stop_devs = .;
} > sram AT > flash
__end_data_sram = .;
```

In the next sections, the data (r/w and readonly), are declared. According to gcc definition, data contains all the initialized data. In KIM, some special sections are defined (see `include/linker.h` for more information). The goal of these sections is to provide a *declarative* approach for some relevant structs. In this way, if you want to add a task to the system, you will just have to declare, wherever you want, a `struct task_t` with `attr_tasks` attributes, and it will automagically be part of main task array.

```
        .bss : {
                . = ALIGN(16);
                __start_bss = .;
                *(.bss);
                . = ALIGN(16);
                __end_bss = .;
        } > sram
}
```

The final part just reminds the linker script to include the bss section, and to put it at the end of the RAM; bss is the uninitialized data, and is set to 0 at startup.

Coming back to our boot process, let's see what happens in `isr_reset` routine. Here follows the C code copied from `arch/arm/cpu-cortex-m-common.c` (shared by any Cortex-M SoC):

```
void isr_reset(void)
{
        unsigned char *src, *dest;

        /* Load data to ram */
        src = &__start_data_flash;
        dest = &__start_data_sram;
        while (dest != &__end_data_sram)
                                        *dest++ = *src++;

        /* Set bss section to 0 */
```

```
        dest = &__start_bss;
        while (dest != &__end_bss)
                                        *dest++ = 0;

        /* Skip to mach or board specific init */
        init();
}
```

`isr_reset` performs three main tasks:

- load data section into RAM;
- set to zero the bss section;
- call the `init` function, which is declared externally, and is specific for each SoC.

Going on with stm32f407xx configuration, `isr_reset` will call the init function defined in `arch/arm/cpu-cortex-m4/soc-stm32f407xx/init.c`:

```
void attr_used init(void)
{
        u32 cpu_freq, ahb_freq, apb_freq;

        /* Init board */
        board_init(&cpu_freq, &ahb_freq, &apb_freq);

        /* Init system ticks */
        wr32(R_SYST_RVR, cpu_freq / SYSTICKS_FREQ);
        wr32(R_SYST_CVR, 0);
        wr32(R_SYST_CSR, BIT0 | BIT1 | BIT2);

        log("ahb freq is %d\n", (uint)ahb_freq);

        /* Skip to main */
        k_main();
}
```

Here, three actions are executed:

- initialize the cpu according to the board needs (`board_init` function); it is specific to each board; please have a look at `arch/arm/cpu-cortex-m4/soc-stm32f407xx/board/discovery_f407vg.c` source code for STM32 Discovery F407 board. Typically, the first settings performed by `board_init` consist in choosing the right configuration for the system clock:

```
/* Enable HSE (8MHz external oscillator) */
or32(R_RCC_CR, BIT16);
while (!(rd32(R_RCC_CR) & BIT17));

/* PLLM=8 PLLN=336, PLLP=00 (2), PLLQ=7; f_PLL=168MHz, f_USB=48MHz */
and32(R_RCC_PLLCFGR, ~0x0f037fff);
or32(R_RCC_PLLCFGR, BIT22 | (7 << 24) | (336 << 6) | 8);
or32(R_RCC_CR, BIT24);
while (!(rd32(R_RCC_CR) & BIT25));
```

```
/* Flash latency */
or32(R_FLASH_ACR, 0b111);

/* Use PLL as system clock, with AHB prescaler set to 4 */
wr32(R_RCC_CFGR, (0x9 << 4) | 0x2);
while (((rd32(R_RCC_CFGR) >> 2) & 0x3) != 0x2);

*cpu_freq = *apb_freq = *ahb_freq = 42000000;
```

- initialize the System Ticks, using the ARM SysTick timer; System Ticks are widely explained on the Internet;
- call the `k_main` function, implemented in `kernel/kim.c`: it is the place where tasks are started and the domain of source code becomes independent on the chip; tasks will be deeply investigated in *Tasks definition and scheduling*.

Note: An exhaustive description regarding the ARM system is out of the scope of the present document: more information about the boot process, the Core registers, the interrupt routines, etc, can be found in the Arm Information Center website. A basic knowledge of gcc linker script is also required to better understand KIM details.

CHAPTER

FIVE

THE *ARCH* DIRECTORY

The *arch* directory is where all hardware-specific code should be: MCU register definitions, drivers for SoC peripherals, code to boot a specific board with required alternate GPIO settings, etc…

A quick look at its tree will clarify the logic behind it:

```
$ tree -d arch/
arch/
├── arm
│   ├── cpu-cortex-m0
│   │   ├── include
│   │   ├── soc-lpc111x
│   │   │   ├── board
│   │   │   └── include
│   │   └── soc-stm32f030
│   │       ├── board
│   │       └── include
│   ├── cpu-cortex-m3
│   │   ├── include
│   │   └── soc-stm32f103
│   │       ├── board
│   │       └── include
│   └── cpu-cortex-m4
│       ├── include
│       ├── soc-stm32f407xx
│       │   ├── board
│       │   └── include
│       ├── soc-stm32f469xx
│       │   ├── board
│       │   └── include
│       └── soc-stm32l4rxxx
│           ├── board
│           └── include
└── unix
```

The *arm* directory is where most of the support efforts have been made so far; the subdirectories hieararchy reflects the hardware hierarchy itself:

- at top, we can find *arm*, and in this folder some common code for all Cortex-M processor is present;
- next level is the Cortex-M version (M0/M3/M4); the init function for these cores is here;

- next follows the SoC level: for each specific chip model, it basically hosts the internal peripherals drivers; in the *include* directory is the `reg.h` file, where all registers for that SoC are listed;
- last level is the board one, which hosts the specific low-level initialization needed by a board, together with the devices declarations.

Note: Despite other contexts (e.g. the separation between Kernel and User Space in Linux), in KIM there are no hard constraints in having all low-level code in arch: registers are accessible from wherever, and it is sometimes quicker to directly configure a peripheral by writing in its registers instead of writing the driver abstraction and use it. This flexibility is left to developer's needs and feelings; the main suggestion is not to abuse of direct register calls, especially if the code is needed to maintained and reused over the time, and/or will run on different hardware platforms.

CHAPTER

SIX

THE *LIB* LIBRARY

In the folder lib are located some common and hw independent facilities for the firmware development:

- `basic.c`: some basic functions, replacement of what would be normally found in C standard libraries;
- `cbuf.c`: an implementation of a circular buffer;
- `crc.c`: at present, it only includes CRC16-CCIT algorithm;
- `kprint.c`: the replacement of printf-like functions, including k_sprintf for string formatting in a buffer.

The header files (*.h*) for lib and kernel source code are mixed in include/ subfolder; so, for instance, `include/basic.h` contains the declarations for the functions defined in `lib/basic.c`

The file `include/basic.h` also contains more useful definition, e.g. those to manipulate the registers:

```
/* 32 bits registers */
static inline void wr32(volatile u32 *reg, u32 val) {*reg = val;}
static inline u32 rd32(volatile u32 *reg) {return *reg;}
static inline void or32(volatile u32 *reg, u32 val) {*reg |= val;}
static inline void and32(volatile u32 *reg, u32 val) {*reg &= val;}
static inline void clr32(volatile u32 *r, int nbit) {and32(r, ~(1 << nbit));}
static inline void set32(volatile u32 *reg, int nbit) {or32(reg, 1 << nbit);}

/* 16 bits registers */
static inline void wr16(volatile u16 *reg, u16 val) {*reg = val;}
static inline u16 rd16(volatile u16 *reg) {return *reg;}
static inline void or16(volatile u16 *reg, u16 val) {*reg |= val;}
static inline void and16(volatile u16 *reg, u16 val) {*reg &= val;}
static inline void clr16(volatile u16 *reg, int nbit) {and16(reg, ~(1 <<␣
↪nbit));}
static inline void set16(volatile u16 *reg, int nbit) {or16(reg, 1 << nbit);}

/* 8 bits registers */
static inline void wr8(volatile u8 *reg, u8 val) {*reg = val;}
static inline u8 rd8(volatile u8 *reg) {return *reg;}
static inline void or8(volatile u8 *reg, u8 val) {*reg |= val;}
static inline void and8(volatile u8 *reg, u8 val) {*reg &= val;}
static inline void clr8(volatile u8 *reg, int nbit) {and8(reg, ~(1 << nbit));}
static inline void set8(volatile u8 *reg, int nbit) {or8(reg, 1 << nbit);}
```

CHAPTER

SEVEN

TASKS DEFINITION AND SCHEDULING

A *task* inside KIM system is basically defined as a set of callbacks to be executed at start, stop and at a periodic interval (called *step* inside KIM sources. Basically, this allows to implement a Finit State Machine (FSM). This is a different approach compared to commonly used threads, with some advantages and a drawback:

- all machine interrupts are directly handled by the lower layer drivers (in arch/)
- when looking at the application code there always is one and only function running, leading to:
  - no concurrency on variables to be handled (code more deterministic, easier debug and test);
  - no context switching (better performance on a single-core CPU, no need to continuously save thread status from registers to RAM).
- the main drawback of FSM approach is the lack of a proper sleep system, since the task function must always immediately return; if a time wait is needed, this must be manually handled, by comparing current system time with the starting time + desired timeout.

The `struct task_t` is defined in `include/kim.h`, together with the functions to start and stop tasks:

```
struct task_t {
        void (*start)(struct task_t *t);
        void (*step)(struct task_t *t);
        void (*stop)(struct task_t *t);
        void *priv;
        u32 last_run;
        u32 intvl_ms;
        u32 tstart;
        u32 max_duration;
        u32 hits;
        const char *name;
        int running: 1;
        int no_autorun: 1;
        int async_start: 1;
};
```

As stated before, the most important fields are the first three function pointers, which must be filled with the desired functions for each task. Since more tasks of the same kind can cohexist (i.e. more tasks can share the same callbacks), `priv` is a generic pointer where the programmer can hold some private information dedicated to each *instance* of the task.

`last_run`, `tstart`, `running`, `async_start` and `hits` are used runtime to correctly schedule the task execution; `max_duration` can optionally be declared with a positive value in case the task has a fixed execution total time; `no_autorun` can be set to `1` if the task must not automatically start at reset (which is the default behaviour); finally, each task has a `name`, which can be used to look for the task and start it.

An example of task is provided in the file `tasks/task-clock.c`. It counts on the first UART found, counting up to 60s before issuing a newline. In the meantime, each character received on the same UART is echoed, in order to show both sides of UART communication.

To understand how tasks are started, a practical example is given by `cli/cli-tasks.c`, where the CLI command set to handle tasks is implemented (see *Command Line Interface (CLI)* for more information).

For instance, this is a portion of code showing how a task is run, assuming the task name is known:

```
static task_t *task_get_by_arg(const char *argv, int fdout)
{
        struct task_t *t = tasks(0);

        if (isdigit(argv[0]))
                t = task_get(atoi(argv));
        else
                t = task_find(argv);

        if (!t)
                k_fprintf(fdout, "task not found: %s\n", argv);

        return t;
}

static int start_cmd_cb(int argc, char *argv[], int fdout)
{
        int i;
        for (i = 1; i < argc; i++) {

                struct task_t *t = task_get_by_arg(argv[i], fdout);

                if (!t)
                        continue;

                if (t->running)
                        k_fprintf(fdout, "task already running: %s\n",␣
↪argv[i]);
                else
                        task_start(t);
        }
        return 0;
}
```

Tasks can be started *directly* (function `task_start`) or *asynchronously* (function `task_start_async`); on direct start, the task `start` callback is immediately called; on asynchronous start, the main loop calls the `start`.

If you want to have a look at how tasks are handled, just watch the `k_main` and `task_stepall` functions, both defined in `kernel/kim.c`:

```
void task_stepall(void)
{
        struct task_t *t = tasks(0);
        for (;t != &__stop_tsks; t++) {
                if (!t->running) {
                        if (t->async_start) {
                                task_start(t);
                                t->async_start = 0;
                        }
                        continue;
                }

                if (t->max_duration && k_elapsed(t->tstart) > t->max_duration)
                        task_done(t);

                if (k_elapsed(t->last_run) < MS_TO_TICKS(t->intvl_ms))
                        continue;
                t->last_run = k_ticks();
                t->step(t);
                t->hits++;
        }
}

void attr_weak k_main(void)
{
        struct task_t *t = tasks(0);
        struct k_dev_t *d = devs(0);
        int fd;

        for (; d != &__stop_devs; d++) {
                fd = k_fd(dev_major(d->id), dev_minor(d->id));
                if (fd < 0 || !d->drv) {
                        err("Could not open %s (%04x)\n", d->name, d->id);
                        continue;
                }
                if (d->drv->init)
                        d->drv->init(fd);
        }

        for (;t != &__stop_tsks; t++) {
                if (!t->no_autorun)
                        task_start(t);
        }

        while(1) {
                sleep();
                task_stepall();
        }
}
```

In the first part of `k_main` function, the init callback for each device is called; see *I/O system: devices and drivers* for more information.

The subsequent `while(1)` in `k_main` is the core of what is continuously running in the system; the whole main is composed of a few lines, according to the minimalistic principle KIM is adopting.

CHAPTER

EIGHT

I/O SYSTEM: DEVICES AND DRIVERS

The I/O model adopted by KIM is similar to the basic one in Linux, according to the principle "everything is a file", with a major difference: there is no need of *open* and *close* the device: everything is always available, and is up to the programmer avoid any conflict; instead, the typical *read*, *write* and *ioctl* functions are present with the same meaning and syntax of common Unix systems.

The lack of open and close is thought to simplify the whole system, giving more flexibility to the programmer.

Going directly in the source code, you can find the definition of device and driver related structs in the file `include/kim-io.h`:

```
/* dev id <-> (major, minor) conversions */
#define dev_id(maj, min) ((uint16_t)((uint16_t)(maj) << 8 | (min)))
#define dev_major(did) ((uint8_t)(did >> 8))
#define dev_minor(did) ((uint8_t)(did & 0xff))

#define declare_dev(maj, min, _priv, _name) \
                const k_dev_t attr_devs dev_##_name = { \
                .id = dev_id(maj, min), \
                .priv = _priv, \
                .name = str(_name), \
};
```

```
struct __attribute__((packed)) k_drv_t {
        int (*init)(int fd);
        int (*read)(int fd, void *buf, size_t count);
        int (*write)(int fd, const void *buf, size_t count);
        int (*avail)(int fd);
        int (*ioctl)(int fd, int cmd, void *buf, size_t count);
        const char *name;
        uint8_t maj;
        uint8_t unused[3];
};
typedef struct k_drv_t k_drv_t;

struct __attribute__((packed)) k_dev_t {
        uint16_t id;
        const struct k_drv_t *drv;
        void *priv;
        const char *name;
```

```
        u16 class_id;
};
typedef struct k_dev_t k_dev_t;
```

Each device has an `id`, a 16 bit number actually composed of two parts: the *major* and the *minor*, as well as it happens in Linux kernel space drivers (I suggest a search on the Internet if you're not familiar with it).

The major identifies the *kind* of the device, whereas the minor identifies each device of the same kind; in other words, the major is a property of the driver, while the minor is what identifies each device associated with the driver. A device uses the driver having the matching major number.

One more property of devices is the `name` field: it is a mnemonic, human-friendly string which uniquely identifies each device.

The typical procedure to use a device is:

- obtain its *file descriptor* (`fd`) by calling k_fd_byname (for a search by name) or k_fd for a search with major and minor;
- call any of k_read, k_write, k_ioctl or k_avail as they are defined in `include/kim-io.h`

To have a general idea about how to write your driver, I suggest a look at `arch/arm/cpu-cortex-m4/soc-stm32f407xx/uart.c` source file.

Devices and drivers must be statically *declared*, by using the proper attributes. Typically, the driver is declared in the same file as its callback functions are defined:

```
const k_drv_t attr_drvs uart_drv = {
        .maj = MAJ_SOC_UART,
        .name = "stm32-uart",
        .read = uart_read,
        .write = uart_write,
        .avail = uart_avail,
};
```

Devices are instead tipically declared in the board initialization file:

```
const k_dev_t attr_devs uart2_dev = {
        .id = dev_id(MAJ_SOC_UART, MINOR_UART2),
        .name = "uart2",
};
```

Anyway, this is not mandatory, because the linker script mechanism will group any device and driver in the proper section.

CHAPTER

NINE

COMMAND LINE INTERFACE (CLI)

The Command Line Interface provides the capability to interact with KIM-OS on any device proving read and write callback functions.

Typically, this is achieved on a UART connection, but it is fully portable on any custom device providing read and write (e.g. a TCP socket for a telnet-like connection): more CLI can co-exhist in the same firmware.

The code implementing the core of this feature is all contained in `tasks/task-cli.c`. The embedded commands, provided with the basic realease of KIM, are in *cli* subfolder, and grouped by type:

- `cli-dbg.c`: commands helping in debug (register read/write);
- `cli-dev.c`: commands to read and write over devices;
- `cli-tasks.c`: commands to start and stop a task;
- `cli-version.c`: the version command, retreving information about firmware version;

One special command, `help`, is also available (and implemented in `tasks/task-cli.c` itself).

Going deeper into CLI implementation and usage, here is how the struct cli is defined (in `include/kim.h`):

```
struct __attribute__((packed)) cli_cmd_t {
        int narg;
        int (*cmd)(int argc, char *argv[], int fdout);
        const char *name;
        const char *descr;
};
```

`name` is the command name itself, `descr` is the description shown by the `help` command, `narg` is the minimum number of arguments required by the command, and `cmd` is the callback executing the command.

For an example about how to add some command to the system, please just browse into *cli* subfolder. For instance, the `rw` command, which performs a direct write into a 32-bit register, is implemented in a these few lines:

```
static int rw_cmd_cb(int argc, char *argv[], int fdout)
{
                wr32((u32*)atoi_hex(argv[1]), atoi_hex(argv[2]));
                return 0;
}
```

```
const struct cli_cmd_t attr_cli cli_rw = {
                .narg = 2,
                .cmd = rw_cmd_cb,
                .name = "rw",
                .descr = "Register Write: set a CPU register (in hex)\r\n"
                         "\tUsage: rw <addr> <val>",
};
```

Note: When executing the callback, the CLI core calls `cmd` with the `argc` and `argv` filled in like standard C *main function* convention: `argv[0]` is the command name itself, while `argc` is 1 + the number of command line parameters.

CHAPTER

TEN

WRITING YOUR OWN APPLICATION

This chapter provides a practical description about how to implement your own application, (*app* in brief). Basically, these are the steps:

- prepare a suitable config, e.g.:

```
ARCH := arm
CPU := cortex-m0
SOC := stm32f030
SOC_VARIANT := stm32f030c8
BOARD := wonderful-board
APP := wonderful-app
```

- if your SoC is not yet supported, you need to implement low-level drivers, by adding a proper subdirectory in *arch* (see *The arch directory*), at least for the peripherals you're about to use; *soc-stm32l4rxxx* or *soc-stm32f407xx* can be a taken as a model for your implementation;

Note: Quite often, the SoC peripherals of chips produced by the same vendor are very similar, if not exactly the same; at present, KIM does has no concept of *generic vendor drivers*, but it's something to improve quite soon, in order to avoid, or at least reduce, code duplication.

- once you have all your low-level part implemented, create the proper *app* subdirectory, *app/wonderful-app* in our example;
- inside this directory, you need to add:
  - the `.mk` Makefile include, having the same name of the app itself;
  - one or more tasks with at least one task declared without the `no_autorun` flag set to 1;
- as a basic example, the clock app uses the default generic tasks/task-clock.c in order to provide the implementation of a simple task.

CHAPTER

ELEVEN

LICENSE

CHAPTER

TWELVE

GLOSSARY

KIM "Keep It Minimal", the acronym for KIM Operating System, designed with a minimalistic approach to MCU programming.

SoC System on Chip

MCU Micro Controller Unit

CLI Command Line Interface

CRC Cyclic Redundancy Code

INDEX