
kim-os Documentation

Release 1.0

Ing. Aurelio Colosimo

May 20, 2020

TABLE OF CONTENTS:

1	What is KIM-OS	1
2	Source code organization	3
3	Quick start reference	5
4	Cortex-M cores: from power-on to running application	7
5	The <i>arch</i> directory	13
6	The <i>lib</i> library	15
7	Tasks definition and scheduling	17
8	I/O system: devices and drivers	19
9	Command Line Interface (CLI)	21
10	Writing your own application	23
11	License	25
12	Glossary	27
	Index	29

WHAT IS KIM-OS

KIM-OS (briefly, KIM) stands for “**Keep-It-Minimal Operating System**”. 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](#) 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 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).

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 usable and intended as an example about how to write your tasks.

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.

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_isr_v_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_tskts = .;
    *(tskts)
    __stop_tskts = .;
    __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 automatically 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.

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 hierarchy 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 context (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 peripherals by writing in its registers instead of writing a driver 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.

THE *LIB* LIBRARY

[TODO: Complete this Chapter]

TASKS DEFINITION AND SCHEDULING

[TODO: Complete this Chapter]

I/O SYSTEM: DEVICES AND DRIVERS

[TODO: Complete this Chapter]

COMMAND LINE INTERFACE (CLI)

[TODO: Complete this Chapter]

WRITING YOUR OWN APPLICATION

[TODO: Complete this Chapter]

LICENSE

Copyright © 2016 Aurelio Colosimo

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

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

INDEX

K

KIM, [27](#)

M

MCU, [27](#)

S

SoC, [27](#)