
EwoK Documentation

Release 1.0

ANSSI

May 17, 2019

CONTENTS

1	What is EwoK ?	3
2	EwoK architecture	5
3	EwoK API	7
4	Internals	43
5	FAQ	59



Contents

- *EwoK microkernel*
 - *What is EwoK ?*
 - *EwoK architecture*
 - *EwoK API*
 - *Internals*
 - *FAQ*

WHAT IS EWOK ?

EwoK is a microkernel targeting micro-controllers and embedded systems aiming at building secure and trusted devices.

Drivers are hold in userspace. Unlike most of other microkernels, the goal is to support complex drivers (ISO7816, USB, CRYPT, SDIO) while achieving high performances.

1.1 Security properties

EwoK supports the following properties:

- Strict memory partitioning
- Strict partitioning of physical resources (devices, etc.)
- Fixed permissions management, set at compile time and easily verifiable
- Kernel Random Number Generation support (based on True RNG HW on STM32)
- Stack smashing protection in both kernel and userspace tasks
- Userspace Heap smashing defenses
- Proved W^X memory mappings
- Strict temporal separation between declarative phase and execution phase

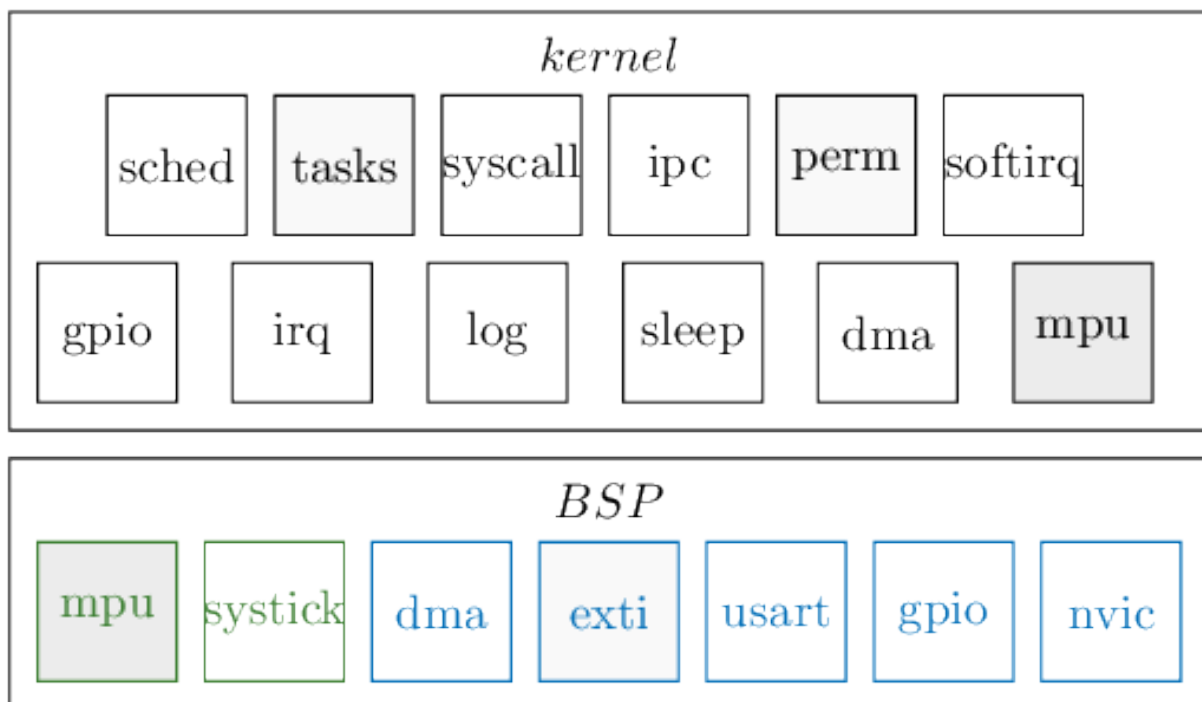
1.2 Performances

Unlike other microkernels, EwoK allows userland drivers to use DMA, GPIOs and EXTIs with the help of some specific syscalls. Such interfaces are directly implemented in the kernel, in order to achieve better performance. Indeed, some hardware need a very responsive software and such responsiveness can not easily be achieved in a typically minimalistic microkernel (like the ones from the L4 family).

EWOK ARCHITECTURE

2.1 Kernel architecture

The kernel is divided into two main components: the **libbsp** and the **kernel** parts.

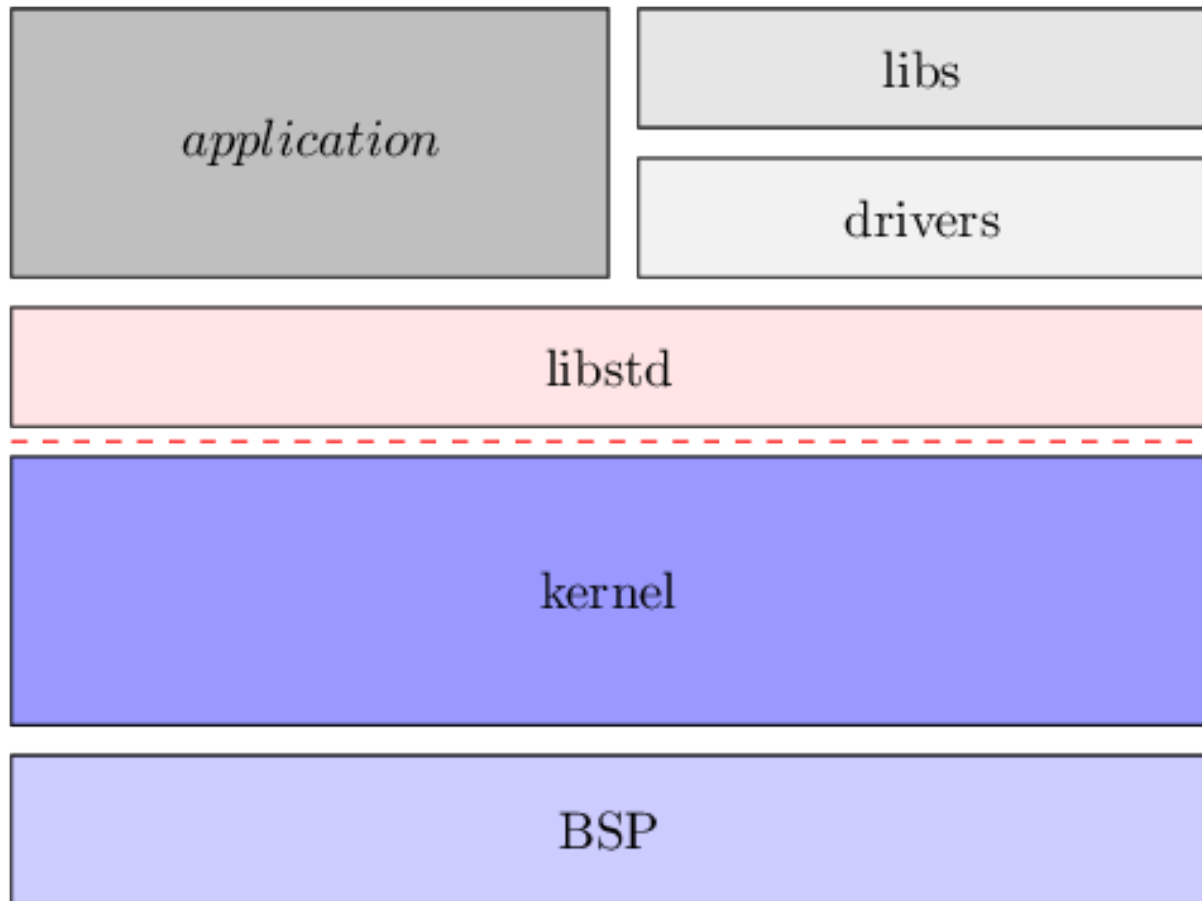


The *libbsp* is the hardware abstraction layer, hosting all the low level and arch-specific drivers (MPU, GPIOs, timers, DMAs, etc.). The *libbsp* is itself separated in two blocks:

1. *SoC-specific drivers*, such as DMA or GPIO support for the STM32F407 board
2. *Core-specific drivers*, such as MPU support for the Cortex-M4 ARMv7m micro-architecture

The *kernel* part contains all specific high level content (scheduling, task management, syscalls, etc.) and uses the *libbsp* as a hardware abstraction for any low-level interaction.

2.2 Drivers



The `lib_std` (*libstd*) is a C standard library that can be used by the user tasks (and hence the userspace drivers). Like the *libc* for *UNIX*-like systems, it implements some useful functions.

The **drivers** are written as userspace libraries. They depend on the *libstd*, and may sometimes depend on each others. Here is the list of the existing drivers.

Libraries bring various userspace features with arch-independent implementations.

EWOK API

EwoK is tuned for high performance embedded systems. The whole microkernel architecture and the provided API are specifically designed for this purpose. Note that for these specific performance constraints, EwoK is not a full-IPC driven microkernel, like the L4 family.

The EwoK API is described here:

3.1 EwoK syscalls

Contents

- *EwoK syscalls*
 - *General principles*
 - * *Triggering a syscall from userland*
 - * *Returned values*
 - * *Synchronous and asynchronous syscalls*
 - * *Syscalls and permissions*
 - *Syscall overview*

3.1.1 General principles

EwoK is designed to host userspace drivers and protocol stacks. Syscalls are driver-oriented and mostly expose device management primitives. Some more *usual* syscalls, like IPC, are also proposed.

Triggering a syscall from userland

In EwoK, syscall parameters are passed by a structure that resides on the stack. The task writes the address of this structure in the `r0` register and executes the `SVC` instruction, which triggers the *SVC interrupt*. The *SVC interrupt* automatically saves registers on the stack, before switching to the MSP stack. The `r0` register has a double function here. It is used to transmit the address of the structure containing the syscalls parameters, but it also stores the value returned by the syscall.

An example of a syscall implementation:

```
e_syscall_ret sys_cfg_CFG_GPIO_GET(uint32_t cfgtype, uint8_t gpioref,
                                   uint8_t * value)
{
    struct gen_syscall_args args =
        { SYS_CFG, cfgtype, gpioref, (uint32_t) value, 0 };
    return do_syscall(&args);
}

e_syscall_ret do_syscall(__attribute__((unused)) struct gen_syscall_args *args)
{
    e_syscall_ret ret;
    asm volatile (
        "svc #0\n"
        "str r0, %[ret]\n"
        : [ret] "=m"(ret) :: "r0");
    return ret;
}
```

Returned values

Syscalls return values may be the following:

SYS_E_DONE	Syscall has succesfully being executed
SYS_E_INVALID	Invalid input data
SYS_E_DENIED	Permission is denied
SYS_E_BUSY	Target is busy, not enough resources, resource is already used

Danger: Never use a syscall without checking its return value, this may lead to unchecked errors in your code

Synchronous and asynchronous syscalls

EwoK kernel is not reentrant. As a consequence, syscalls can be synchronously or asynchronously executed depending on their expected duration.

Most of syscalls are synchronously executed by the kernel. To avoid hindering the whole system, slow syscalls are asynchronously executed: their execution is postponed and is ought to be accomplished by the *softirq* kernel thread.

Note: Actually, *sys_log* is the sole asynchronous syscall

It is worth mentioning that as *Interrupt Service Routines (ISR)* should be quickly executed, EwoK forbids asynchronous syscalls while in this context.

Note also that some syscalls should require some specific permissions, which are set at build time.

Syscalls and permissions

A part of the syscalls require dedicated permissions. See *EwoK permissions* section for more information about EwoK permissions and their impact on the syscall API.

3.1.2 Syscall overview

sys_init, initializing devices

Devices declaration, initialization and configuration is done with the help of the `sys_init()` syscall family.

Contents

- *sys_init, initializing devices*
 - *sys_init(INIT_GETTASKID)*
 - *sys_init(INIT_DEVACCESS)*
 - * *Mapping devices in memory*
 - * *Using GPIOs*
 - * *IRQ handler*
 - *sys_init(INIT_DMA)*
 - *sys_init(INIT_DMA_SHM)*
 - *sys_init(INIT_DONE)*

sys_init(INIT_GETTASKID)

If a task *A* wants to communicate with another task *B*, task *A* needs to retrieve task's *B* identifier.

Note:

- Each running task is identified by a unique identifier: its *task id*. A task have also a name, given by the implementor, to ease its identification.
- If IPC *domains* are supported by the kernel, only tasks in the same *domain* can identify each other.

Getting a *task id* is done with `sys_init(INIT_GETTASKID)` syscall:

```
uint8_t      peer_id;
e_syscall_ret ret;

ret = sys_init(INIT_GETTASKID, "task_b", &peer_id);
if (ret != SYS_E_DONE) {
    ...
}
```

In the example above, if the call is succesful, the `peer_id` parameter is updated with the *task id*.

sys_init(INIT_DEVACCESS)

If a task wants to use a device, it must request it to the kernel using the `sys_init(INIT_DEVACCESS)` syscall.

Warning: DMA streams are not initialized with `sys_init(INIT_DEVACCESS)` but with `sys_init(INIT_DMA)`

To make that request, a `device_t` structure, whose prototype is defined in `kernel/src/C/exported/devices.h`, must be filled. That structure describes the requested device. Its content is:

```
typedef struct {
    char          name[16];
    physaddr_t    address;
    uint32_t      size;
    uint8_t       irq_num;
    uint8_t       gpio_num;
    dev_map_mode_t map_mode;
    dev_irq_info_t irqs[MAX_IRQS];
    dev_gpio_info_t gpios[MAX_GPIOS];
} device_t;
```

The fields of the `device_t` structure are explained here:

- `name` contains a name, useful for debugging purposes
- `address` is the base address of the device in memory (0 if the device is not mapped in memory)
- `size` is the size of the mapping in memory (0 if the device is not mapped in memory)
- `irq_num` is the number of configured IRQs in the `irqs[]` array
- `gpio_num` is the number of configured GPIOs in the `gpios[]` array
- `map_mode` tell if the device must be automatically mapped in task's address space.
- 0 up to 4 *IRQ lines* are defined in `irqs[]` array
- 0 up to 16 *GPIOs* are defined in `gpios[]` array

Below is an example, excerpt from the blinky demo:

```
device_t    leds;
int         desc_leds;

memset (&leds, 0, sizeof (leds));

strncpy (leds.name, "LEDs", sizeof (leds.name));
leds.gpio_num = 4; /* Number of configured GPIO */

leds.gpios[0].kref.port = GPIO_PD;
leds.gpios[0].kref.pin = 12;
leds.gpios[0].mask      = GPIO_MASK_SET_MODE | GPIO_MASK_SET_PUPD |
                          GPIO_MASK_SET_TYPE | GPIO_MASK_SET_SPEED;
leds.gpios[0].mode      = GPIO_PIN_OUTPUT_MODE;
leds.gpios[0].pupd      = GPIO_PULLDOWN;
leds.gpios[0].type      = GPIO_PIN_OTYPER_PP;
leds.gpios[0].speed     = GPIO_PIN_HIGH_SPEED;

leds.gpios[1].kref.port = GPIO_PD;
leds.gpios[1].kref.pin = 13;
leds.gpios[1].mask      = GPIO_MASK_SET_MODE | GPIO_MASK_SET_PUPD |
                          GPIO_MASK_SET_TYPE | GPIO_MASK_SET_SPEED;
...
```

(continues on next page)

(continued from previous page)

```
ret = sys_init(INIT_DEVACCESS, &leds, &desc_leds);
```

In this example:

- `leds` parameter is a `device_t` structure that describes the requested device. Here, the leds on the *stm32f407* board.
- `desc_leds` is a *device id* returned by the syscall. It's not very useful here as the *device id* is used only in some few syscalls (`sys_cfg(CFG_DEV_MAP)` and `sys_cfg(CFG_DEV_UNMAP)`)

Mapping devices in memory

Due to MPU constraints on Cortex-M, a task can not map simultaneously more than 4 devices in memory.

If a task needs to manage more than 4 devices, it should use the syscalls `sys_cfg(CFG_DEV_MAP)` and `sys_cfg(CFG_DEV_UNMAP)` to voluntary map and unmap the desire devices. Those syscalls can be used only if:

- `map_mode` field of the `device_t` structure is set to `DEV_MAP_VOLUNTARY`
- the task is granted with a specific permission to do this (set in the *menuconfig* kernel menu).

Using GPIOs

Each GPIO port/pin pair is identified by a `kref` value. That value must be filled in when using the `sys_cfg(CFG_GPIO_GET)` and `sys_cfg(CFG_GPIO_GET)` syscalls (see *sys_cfg, configuring devices*).

IRQ handler

For each IRQ, an *Interrupt Service Routine* (ISR) should be declared. Here is an example excerpt from the demo :

```
memset (&button, 0, sizeof (button));
strncpy (button.name, "BUTTON", sizeof (button.name));

button.gpio_num = 1;
button.gpios[0].kref.port = button_dev_infos.gpios[BUTTON].port;
button.gpios[0].kref.pin  = button_dev_infos.gpios[BUTTON].pin;
button.gpios[0].mask      = GPIO_MASK_SET_MODE | GPIO_MASK_SET_PUPD |
                           GPIO_MASK_SET_TYPE | GPIO_MASK_SET_SPEED |
                           GPIO_MASK_SET_EXTI;
button.gpios[0].mode      = GPIO_PIN_INPUT_MODE;
button.gpios[0].pupd     = GPIO_PULLDOWN;
button.gpios[0].type      = GPIO_PIN_OTYPER_PP;
button.gpios[0].speed     = GPIO_PIN_LOW_SPEED;
button.gpios[0].exti_trigger = GPIO_EXTI_TRIGGER_RISE;
button.gpios[0].exti_lock  = GPIO_EXTI_UNLOCKED;
button.gpios[0].exti_handler = (user_handler_t) exti_button_handler;

/* Now that the button device structure is filled, use sys_init to
 * initialize it */
ret = sys_init(INIT_DEVACCESS, &button, &desc_button);
```

sys_init(INIT_DMA)

If a task wants to use a DMA stream, it must request it to the kernel using the `sys_init(INIT_DMA)` syscall. To make that request, a `dma_t` structure, whose prototype is defined in `kernel/src/C/exported/dma.h`, must be filled. That structure describes the requested DMA stream. Its content is:

```
typedef struct {
    uint8_t dma;           /* DMA controller identifier (1 for DMA1, 2 for DMA2, etc.
    ↪) */
    uint8_t stream;
    uint8_t channel;
    uint16_t size;         /* Transferring size in bytes */
    physaddr_t in_addr;    /* Input base address */
    dma_prio_t in_prio;    /* Priority */
    user_dma_handler_t in_handler; /* ISR with one argument (irqnum), see types.h */
    physaddr_t out_addr;   /* Output base address */
    dma_prio_t out_prio;   /* Priority */
    user_dma_handler_t out_handler; /* ISR with one argument (irqnum), see types.h */
    dma_flowctrl_t flow_control; /* Flow controller */
    dma_dir_t dir;         /* Transfert direction */
    dma_mode_t mode;       /* DMA mode */
    dma_datasize_t datasize; /* Data unit size (byte, half-word or word) */
    bool mem_inc;          /* Increment for memory */
    bool dev_inc;          /* Increment for device */
    dma_burst_t mem_burst; /* Memory burst size */
    dma_burst_t dev_burst; /* Device burst size */
} dma_t;
```

Example:

```
dma.dma = DMA2;
dma.stream = DMA2_STREAM_SDIO_FD;
dma.channel = DMA2_CHANNEL_SDIO;
dma.size = 0; /* Set later with DMA_RECONF */
dma.in_addr = (physaddr_t) 0; /* Set later with DMA_RECONF */
dma.in_prio = DMA_PRI_HIGH;
dma.in_handler = (user_dma_handler_t) sdio_dmacallback;
dma.out_addr = (volatile physaddr_t) sdio_get_data_addr();
dma.out_handler = (user_dma_handler_t) sdio_dmacallback;
dma.flow_control = DMA_FLOWCTRL_DEV;
dma.dir = MEMORY_TO_PERIPHERAL;
dma.mode = DMA_FIFO_MODE;
dma.datasize = DMA_DS_WORD;
dma.mem_inc = 1;
dma.dev_inc = 0;
dma.mem_burst = DMA_BURST_INC4;
dma.dev_burst = DMA_BURST_INC4;

ret = sys_init(INIT_DMA, &dma, &dma_descriptor);
```

In this example, the `dma_descriptor` is an identifier returned by the syscall and used by the `sys_cfg(CFG_DMA_RECONF)` and `sys_cfg(CFG_DMA_RELOAD)` syscalls.

Note: For the sake of security, the EwoK DMA implementation denies *memory-to-memory* transfers.

sys_init(INIT_DMA_SHM)

When multiple tasks take part in a complex data flow with multiple DMA copies from one device to another (e.g. from a USB High Speed device to the SDIO interface), it may be efficient to support pipelined DMA transfers with low latency between tasks.

As tasks have no rights to request a DMA transfer from another task's buffer toward a device they own, this syscall allows to explicitly declare this right, based on the Ewok permission model.

Using such a mechanism, the task can initiate a DMA transfer from a foreign memory buffer without any direct access to it, but only toward a given peripheral (e.g. a CRYPT device or an SDIO device).

Sharing a DMA buffer with another task is done with the following API:

```
e_syscall_ret sys_init(INIT_DMA_SHM, dma_shm_t *dma_shm);
```

Declaring a DMA SHM does not create a mapping of the other task's buffer in the current task memory map. Only the DMA controller is able to access the other task's buffer, as a source or destination of the transaction. The current task is not able to read or write directly into the buffer. As the MEMORY_TO_MEMORY DMA transaction is also forbidden, the task is not able to use the DMA to get back its content from the DMA controller by requesting a copy into its own memory map.

sys_init(INIT_DONE)

As previously described, this syscall locks the initialization phase and starts the nominal phase of the task. From now on, the task can execute all syscalls but the `sys_init()` one under its own permission condition.

Finalizing the initialization phase is done with the following API:

```
e_syscall_ret sys_init(INIT_DONE);
```

sys_cfg, configuring devices

The resources (GPIOs, DMA, etc.) reconfiguration request is done by the `sys_cfg()` syscall family.

Contents

- *sys_cfg, configuring devices*
 - *sys_cfg(CFG_GPIO_SET)*
 - *sys_cfg(CFG_GPIO_GET)*
 - *sys_cfg(CFG_GPIO_UNLOCK_EXTI)*
 - *sys_cfg(CFG_DMA_RECONF)*
 - *sys_cfg(CFG_DMA_RELOAD)*
 - *sys_cfg(CFG_DMA_DISABLE)*
 - *sys_cfg(CFG_DEV_MAP)*
 - *sys_cfg(CFG_DEV_UNMAP)*
 - *sys_cfg(CFG_DEV_RELEASE)*

sys_cfg(CFG_GPIO_SET)

GPIOs are not directly mapped in the task's memory. As a consequence, setting the GPIO output value is done using a syscall. The GPIO must be registered as output for the syscall to succeed. Only the GPIO kref is needed by this syscall, see the `sys_init(INIT_DEVACCESS)` explanations about kref for further details.

Setting an output GPIO is done with the following API:

```
e_syscall_ret sys_cfg(CFG_GPIO_SET, uint8_t gpioref, uint8_t value);
```

The value set is the third argument.

Important: The GPIO to set must have been previously declared as output in the initialization phase.

sys_cfg(CFG_GPIO_GET)

Getting a GPIO value for a GPIO configured in input mode is done using a syscall. Only the GPIO kref is needed by this syscall, see the `sys_init(INIT_DEVACCESS)` explanations about kref for further details.

Getting an input value of a GPIO is done with the following API:

```
e_syscall_ret sys_cfg(CFG_GPIO_GET, uint8_t gpioref, uint8_t *val);
```

The value read is put in the third argument.

Important: The GPIO queried must have been previously declared as input in the initialization phase.

sys_cfg(CFG_GPIO_UNLOCK_EXTI)

Note: Synchronous syscall, executable in ISR mode

There are times when external interrupts may:

- Arise only one time and need to be muted voluntarily for a given amount of time
- Be unstable and generate uncontrolled bursts, when the external IP is not clean and has hardware bugs

For these two cases, the EwoK kernel supports a specific GPIO configuration which allows, when an EXTI interrupt is configured, to choose whether:

- The EXTI line is masked at handler time, by the kernel. The user ISR will be executed but there will be no more EXTI interrupts pending on the interrupt line
- The EXTI line is not masked, and the EXTI is only acknowledged. The EXTI source can continue to emit other interrupts and the userspace ISR handler will be executed for each of them

The choice is done using the `exti_lock` field of the `gpio` structure, using either:

- `GPIO_EXTI_UNLOCKED` value: the EXTI line is not masked and will continue to arise when the external HW IP emits events

- `GPIO_EXTI_LOCKED` value: the EXTI line is masked once the interrupt has been scheduled for being serviced. The userspace task needs to unmask it voluntarily using the appropriate syscall. No other EXTI will be received without unmasking.

Unmasking a given EXTI interrupt is done using the `sys_cfg(CFG_GPIO_UNLOCK_EXTI)` syscall. This syscall has the following API:

```
e_syscall_ret sys_cfg(CFG_GPIO_UNLOCK_EXTI, uint8_t gpio_ref);
```

The `gpio_ref` parameter is the kref identifier of the GPIO, like the one used in the other GPIO manipulation syscalls. Unlocking the EXTI line is a synchronous syscall.

`sys_cfg(CFG_DMA_RECONF)`

Note: Synchronous syscall, executable in ISR mode

DMA operations are performed by EwoK microkernel on the behalf of userspace tasks. After completion of a DMA transfert the DMA channel is disable until it is either reloaded or reconfigured. For allowing the user to change the input/output buffers of a DMA channel, it is permitted to reconfigure part of the DMA channel information.

Only some fields of the `dma_t` can be reconfigured :

- ISR handlers address
- Input buffer address (for memory to peripheral mode)
- Output buffer address (for peripheral to memory mode)
- Buffer size
- DMA mode (direct, FIFO or circular)
- DMA priority

Reconfiguring a part of a DMA stream is done with the following API:

```
e_syscall_ret sys_cfg(CFG_DMA_RECONF, dma_t* dma, dma_reconf_mask_t reconfmask);
```

The mask parameter allows the user to specify which field(s) need(s) to be reconfigured.

As these fields are a part of the `dma_t` structure (see Ewok kernel API technical reference documentation), the syscall requires this entire structure.

Hint: The easiest way to use this syscall is to keep the `dma_t` structure used during the initialization phase and to update it during the nominal phase

Important: The DMA that needs to be reconfigured must have been previously declared in the initialization phase.

`sys_cfg(CFG_DMA_RELOAD)`

Note: Synchronous syscall, executable in ISR mode

When a DMA tranfert is finished, the corresponding DMA channel is disable until it is either reloaded or reconfigured. A reload can be performed when the DMA controller is requested to redo exactly the same action, without any modification of the DMA channel properties. Reloading a DMA channel is faster than reconfiguring it. The kernel only needs to identify the DMA controller and stream, and does not need a whole DMA structure. The task can then use only the `id` field of the `dma_t` structure.

Reloading a DMA stream is done with the following API:

```
e_syscall_ret sys_cfg(CFG_DMA_RELOAD, uint32_t dma_id);
```

Important: The DMA that needs to be reloaded must have been previously declared in the initialization phase.

`sys_cfg(CFG_DMA_DISABLE)`

Note: Synchronous syscall, executable in ISR mode

It is possible to disable a DMA stream. In this case, the DMA channel is stopped and can be re-enabled by calling one of `sys_cfg(CFG_DMA_RELOAD)` or `sys_cfg(CFG_DMA_RECONF)` syscalls.

This is useful for DMA streams in circular mode, as they never stop unless the software asks them to.

Disabling a DMA stream is done with the following API:

```
e_syscall_ret sys_cfg(CFG_DMA_DISABLE, uint32_t dma_id);
```

Important: The DMA that needs to be disabled must have been previously declared in the initialization phase.

`sys_cfg(CFG_DEV_MAP)`

Note: Synchronous syscall, executable only in main thread mode

Ewok Microkernel allows a task to map only a restricted number of devices at a time. Voluntary mapped devices permit to map, configure and unmap in a task more than the maximum number of concurrently mapped devices. It also allows us to avoid mapping devices whose concurrent mapping is dangerous (e.g. concatenated mappings).

It is possible to declare a device as voluntary mapped (field `map_mode` of the `device_t` structure. This field can be set to the following values:

- `DEV_MAP_AUTO`
- `DEV_MAP_VOLUNTARY`

When using `DEV_MAP_AUTO`, the device is automatically mapped in the task address space when finishing the initialization phase, and is kept mapped until the end of the task life-cycle.

When using `DEV_MAP_VOLUNTARY`, the device is not mapped by the kernel and the task has to map the device itself (later in the life-cycle). In that case, the device is mapped using this very syscall.

Mapping a device is done using the device id, hosted in the `id` field of the `device_t` structure, which is set by the kernel at registration time.

Mapping a device is done with the following API:

```
e_syscall_ret sys_cfg(CFG_DEV_MAP, uint8_t dev_id);
```

Important: Declaring a voluntary mapped device requires a specific permission: `PERM_RES_MEM_DMAP`

Note: Mapping a device requires a call to the scheduler, in order to reconfigure the MPU, this action is costly

`sys_cfg(CFG_DEV_UNMAP)`

Note: Synchronous syscall, executable only in main thread mode

When using `DEV_MAP_VOLUNTARY`, a previously voluntary mapped device can be unmapped by the task. Unmapping a device frees the corresponding MPU slot, this is useful e.g. when the task requires more than the maximum number of concurrently devices.

Important: While the device is configured, device's ISR still maps the device, even if it is unmapped from the main thread

Important: Unmapping a device does not mean disabling it, the hardware device still works and emits IRQs that are handled by the task's registered ISR. It is the task's responsibility to properly disable the device before unamping it if necessary

Note: Unmapping a device requires a call to the scheduler, in order to reconfigure the MPU, this action is costly

Unmapping a device is done using the device id, stored in the `id` field of the `device_t` structure, which is set by the kernel at registration time.

Unmapping a device is done with the following API:

```
e_syscall_ret sys_cfg(CFG_DEV_UNMAP, uint8_t dev_id);
```

`sys_cfg(CFG_DEV_RELEASE)`

Note: Synchronous syscall, executable only in main thread mode

A task may want to revoke its accesses to a given device. This can be done by requesting the kernel to release the device using its device descriptor. The device is then fully deactivated (including associated RCC clock and interrupts) and fully removed from the task's context.

Warning: This action cannot be undone. The device is released until reboot

A released device shall never be allocated by another task. This can only happen if the device is released by a given task before another task has finished its initialization phase.

Danger: You should **not** interleave nominal and initializing phases between tasks to avoid potential unwanted device reallocation. Take care to synchronize init sequences correctly. The kernel **does not** clear the device registers at release time

Releasing a device is done with the following API:

```
e_syscall_ret sys_cfg(CFG_DEV_RELEASE, uint8_t dev_id);
```

sys_log

Contents

- *sys_log*
 - *sys_log()*

EwoK provides a kernel-controlled logging facility with which any userspace task can communicate. This kernel-controlled logging interface is used for debugging and/or informational purpose and is accessible through the configured kernel U(S)ART.

Important: When the kernel is configured in KERNEL_NOSERIAL mode, the kernel doesn't print out any log. The U(S)ART line isn't even activated. Although, the `sys_log()` behavior stays unchanged. This is particularly useful in a paranoid mode when generating 'production' firmwares: we are ensured that no information leak through serial debug can be exploited by an attacker.

sys_log()

Note: Asynchronous syscall, not executable in ISR mode

This syscall permits to transmit a logging message on the kernel-handled serial interface. The message must be short enough (less than 128 bytes). Any longer message is truncated.

Message printing is not synchronous and is handled by a kernel thread. As a consequence, message printing is not a reactive syscall and may take some time before being executed if multiple IRQ and/or ISR are to be executed before.

The `sys_log` syscall has the following API:

```
e_syscall_ret sys_log(logsize_t size, const char *msg);
```

Important: The kernel logging facility is a debugging helper feature. It should not be used as a trusted console. For this, please use a dedicated task holding a userspace USART support with specific security properties instead

sys_ipc, Inter-Process Communication

Contents

- *sys_ipc, Inter-Process Communication*
 - *Synopsis*
 - *Prerequisites*
 - *sys_ipc(SEND_SYNC)*
 - *sys_ipc(SEND_ASYNC)*
 - *sys_ipc(RECV_SYNC)*
 - *sys_ipc(RECV_ASYNC)*

Synopsis

Inter-Process Communication is done using the `sys_ipc()` syscall family. IPC can be either *synchronous* or *asynchronous*:

- *synchronous* IPC requests are blocking until the message has been sent and received
- *asynchronous* IPC are non blocking. They may return an error if the other side of the channel is not ready.

EwoK detects IPC mutual lock (two task sending IPC to each other), returning `SYS_E_BUSY` error but it does not detect cyclic deadlocks between multiple tasks (more than 2). Be careful when designing your IPC automaton!

Note that IPC are half-duplex. For example, if task *A* can send messages to task *B*, the reciprocity is not always true and task *B* may have no permission to send any message to *A*.

Prerequisites

If a task *A* want to communicate with another task *B*, task *A* need to retrieve *B*'s *task id*. Getting a task identifier is done with `sys_init(INIT_GETTASKID)` syscall:

```
uint8_t      id;
e_syscall_ret ret;

ret = sys_init(INIT_GETTASKID, "task_b", &id);
if (ret != SYS_E_DONE) {
    ...
}
```

For more details, see *sys_init*, *initializing devices*.

Important: Notice that any attempt to receive or to send a message with an IPC during the task *init mode* fails with `SYS_E_DENIED`.

sys_ipc(SEND_SYNC)

A task can synchronously send data to another task. The task is blocked until the other task emit either a `sys_ipc(RECV_SYNC)` or a `sys_ipc(RECV_ASYNC)` syscall:

```
uint8_t      id;
logsize_t    size;
char         *msg = "hello";
e_syscall_ret ret;
...
ret = sys_ipc(IPC_SEND_SYNC, id, sizeof(msg), msg);
if (ret != SYS_E_DONE) {
    ... /* Error handling */
}
```

The `sys_ipc(IPC_SEND_SYNC, ...)` can return:

- `SYS_E_DONE`: The message has been succesfully emitted
- `SYS_E_DENIED`: The current task is not allowed to communicate with the other task
- `SYS_E_INVALID`: One of the syscall argument is invalid (invalid task id, pointer value, etc.)
- `SYS_E_BUSY`: This happens only in the very rare condition of a synchronous send is emitted after an asynchronous send and while the receiver has not emitted any receive syscall.

sys_ipc(SEND_ASYNC)

Asynchronous send is used to send a message without waiting for it to be received. The message is kept in a kernel's buffer until the receiver read it:

```
uint8_t      id;
logsize_t    size;
char         *msg = "hello";
e_syscall_ret ret;
...
ret = sys_ipc(IPC_SEND_ASYNC, id, sizeof(msg), msg);
if (ret != SYS_E_DONE) {
    ... /* Error handling */
}
```

The `sys_ipc(IPC_SEND_ASYNC, ...)` can return:

- `SYS_E_DONE`: The message has been succesfully emitted
- `SYS_E_DENIED`: the current task is not allowed to communicate with the other task
- `SYS_E_INVALID`: one of the syscall argument is invalid (invalid task id, pointer value, etc.)
- `SYS_E_BUSY`: (only in rare occasion) only when the target has already a message from the current task that has not been read yet. This also happens if the target task has sent an IPC to the current task which is not yet received (communication channel already used)

Mixing synchronous and asynchronous IPC is possible but, of course, need some very careful thinking.

sys_ipc(RECV_SYNC)

A task can synchronously wait for a message:

- from another specific task, by setting accordingly the task *id*
- from any task, by setting the task *id* to ANY_APP

The task is blocked until a readable message is feed:

```
uint8_t      id;
logsize_t    size;
char         buf[128];
e_syscall_ret ret;

id  = ANY_APP;      /* Waiting a msg from any task */
size = sizeof(buf); /* Receiving buffer max size */

ret = sys_ipc(IPC_RECV_SYNC, &id, &size, buf);
if (ret != SYS_E_DONE) {
    ... /* Error handling */
}
```

When a message is received, the kernel modify the following parameters (based on the example above):

- *id*: to know which task has sent the message
- *size*: to set message's size
- *buf*: the message is copied into the receiving buffer

The `sys_ipc(IPC_RECV_SYNC, ...)` can return:

- `SYS_E_DONE`: The message has been succesfully received
- `SYS_E_DENIED`: the current task is not allowed to communicate with the other task set as target
- `SYS_E_INVALID`: one of the syscall argument is invalid (invalid task id, pointer value, etc.) or the buffer size is too small to get back the message.
- `SYS_E_BUSY`: (only in rare occasion) only when the target is already in receiving mode, waiting for the current task to send a message.

sys_ipc(RECV_ASYNC)

Asynchronous receive is used to read any pending message. The task is not blocked and directly returns:

```
ret = sys_ipc(IPC_RECV_ASYNC, &id, &size, buf);
```

This syscall returns the same values that is synchronous counterpart plus `SYS_E_BUSY` if there is no message to read.

sys_get_systick

Return elapsed time since the boot.

Contents

- *sys_get_systick*
 - *sys_get_systick()*

sys_get_systick()

Returns the current timestamp in a `uint64_t` value. The precision might be:

```
typedef enum {
    PREC_MILLI, /* milliseconds */
    PREC_MICRO, /* microseconds */
    PREC_CYCLE  /* CPU cycles */
} e_tick_type;
```

Example:

```
uint64_t dma_start_time;

ret = sys_get_systick(&dma_start_time, PREC_MILLI);
if (ret != SYS_E_DONE) {
    ...
}
```

sys_yield

Contents

- *sys_yield*
 - *sys_yield()*

In some situations, yielding is an efficient way to optimize the scheduling. Historically, `yield()` is a collaborative syscall requesting the end of the current slot, asking for the scheduling of a new task. In embedded systems, such a call may help the scheduler in optimizing the tasks execution by voluntarily reducing a task's slot when no more execution is required.

sys_yield()

Note: Synchronous syscall, **not** executable in ISR mode as an ISR as no reason to yield

In EwoK, all tasks main threads can yield. When this happens, the task's thread will not be scheduled again until an external event requires its execution. Such external events can be:

- An IRQ registered by the task. When the ISR is executed, the task's main thread becomes runnable again
- An IPC targeting the task is sent by another task

The yield syscall has the following API:

```
e_syscall_ret sys_yield()
```

Warning: Using `sys_yield()` should be a requirement when using MLQ_RR scheduling scheme with asynchronous communication mechanisms, to avoid starvation

sys_sleep

Contents

- *sys_sleep*
 - *sys_sleep()*

It is possible to request a temporary period during which the task is not schedulable anymore. This period is fixed and the task is awoken at the end of it. This is a typical sleep behavior.

sys_sleep()

Note: Synchronous syscall, but **not** executable in ISR mode, as there is no reason for an ISR to sleep

EwoK supports two sleep modes:

- deep, non-preemptive sleep: the task is requesting a sleep period during which its main thread is not awoken, even by its ISR or external IPC
- preemptive sleep: the task is requesting a sleep period that can be shortened if an external event (ISR, IPC) arises

The sleep syscall has the following API:

```
typedef enum {
    SLEEP_MODE_INTERRUPTIBLE,
    SLEEP_MODE_DEEP
} sleep_mode_t;

e_syscall_ret sys_sleep(uint32_t duration, sleep_mode);
```

The sleep duration is specified in milliseconds. There is no specific permission required to sleep.

Warning: The sleep time is always a scheduler period multiple and is rounded to the next scheduler execution

sys_reset

Contents

- *sys_reset*

– `sys_reset()`

There are some situations where an event may require a board reset. These events may be:

- external: receiving an IRQ or an EXTI at a certain time of the execution phase
- internal: reading a strange value or receiving an IPC/hang request from another task (case of a security monitor for e.g.)

In these cases, the application may require the board to reboot. This reboot implies a full memory RAM reset of various application RAM slots and a complete cleaning of the ephemeral values (e.g. locally duplicated cryptographic information in SoC HW IP).

Doing such request is ensured by calling `sys_reset()` syscall.

`sys_reset()`

Note: Synchronous syscall, executable in ISR mode

In EwoK, only tasks with `TSK_RST` permission can ask the kernel for board reset. Reset is synchronous. Any current DMA transfer may be incomplete (and e.g. generate mass-storage consistency errors when dealing with SCSI, and so on).

The reset syscall has the following API:

```
e_syscall_ret sys_reset()
```

Warning: `sys_reset()` is highly impacting the system behavior and should be used only by specific (at most one in a secure system) task(s). This permission should not be given to a task with a big attack surface. Usually, only a trusted security monitor task is allowed to perform a board reset

`sys_lock`

Pure userspace semaphore, as proposed in `libstd.h`, do not permit easy handling of variable shared between ISR and main threads. ISR treatments do not allow to sleep or wait for the main thread to release semaphores. The solution to this problem is to instruct Ewok not to schedule the ISR routine while the shared variable is in use in the main thread. Of course such a situation ought to be short.

Contents

- `sys_lock`
 - `sys_lock()`

`sys_lock()`

`sys_lock`: postpone the ISR while a lock is set by the main thread. This efficiently creates a critical section in the main thread with respect to the ISR thread.

Note: Synchronous syscall, executable in main thread mode only

In EwoK, all tasks main threads can lock one of their variables without requesting any specific permission.

The lock syscall has the following API:

```
e_syscall_ret sys_lock(LOCK_ENTER);
e_syscall_ret sys_lock(LOCK_EXIT);
```

Warning: Locking the task should be done for a very short amount of time, as associated ISR are postponed, which may generate big slowdown on the associated devices performance.

sys_get_random

Contents

- *sys_get_random*
 - *sys_get_random()*

The random number generator (RNG) is hold by the EwoK kernel as it is used to initialize the user and kernel tasks canary seed values. This entropy source may be implemented in the kernel as:

- a true random number generator (TRNG) when the corresponding IP exists and its driver is implemented in EwoK. This is the case of the STM32F4 SoCs for which a TRNG IP exists and is supported
- a pseudo-random number generator (PRNG), implemented as a full software algorithm. This entropy source can't be considered with the same security properties as the TRNG one
- a mix of the two sources (i.e. a TRNG as one of the entropy sources of a PRNG)

As the RNG support is hosted in the EwoK kernel, a specific syscall exists to get back a random content from the kernel. This content can be used as a source of entropy for various algorithms, and the userland can also implement its own PRNG processing based on this entropy.

sys_get_random()

Note: Synchronous syscall, executable in ISR mode

Get back some random content from the kernel is easy with EwoK and can be done using a single, synchronous, syscall.

If the random content is okay, the syscall returns `SYS_E_DONE`. If the random number generator fails to generate the asked random content, the syscall returns `SYS_E_BUSY`. If the task doesn't have the `RES_TSK_RNG` permission, the syscall returns `SYS_E_DENIED` and the buffer is not modified.

The `sys_get_random()` syscall has the following API:

```
e_syscall_ret sys_get_random(char *buffer, uint16_t buflen)
```

Warning: Using `sys_get_random` requires the specific `RES_TSK_RNG` permission for security reasons (e.g. avoid entropy source exhaustion by an untrusted task). This permission is not needed for initializing the task's canaries as this is automatically performed when creating the tasks.

3.2 Managing devices from userland

Contents

- *Managing devices from userland*
 - *Task life-cycle*
 - * *Initialization state*
 - * *Nominal state*
 - *Declaring and initializing resources*
 - * *Declaring and initializing a device*
 - *Declaring a GPIO pin*
 - *GPIOs and external interrupts (EXTI)*
 - *Declaring an IRQ*
 - *Acknowledging interrupts with posthooks*
 - * *Declaring and initializing a DMA stream*
 - * *Manipulating a DMA*
 - *Reconfiguring a DMA stream*
 - *Reloading a DMA stream*
 - * *Declaring and initializing a DMA SHM*

3.2.1 Task life-cycle

EwoK userspace tasks should follow a specific life-cycle, based on two sequential states:

- The *initialization state*, during which devices are declared and initialized
- The *Nominal state*

Initialization state

All resources declarations are performed during the initialization state. During this state, the task can:

- declare and initialize a device
- request DMA channels
- ask for other tasks' identifiers
- request some DMA shared memory (that will be shared with another task)

- log messages into the kernel log console

These actions depend on permissions, as defined in *EwoK permission model*.

During this state, the task cannot use any device, nor interact with any other task. Trying to use a device at this state or to interact with other tasks will elicit a memory fault or a `SYS_E_DENIED`. The only possible syscalls are `sys_log()`, used by `printf()`, and the `sys_init()` syscalls family.

Danger: Do not try to access any registered device memory during the initialization phase, this will result into a memory fault

Ending the initialization phase is done with the following:

```
sys_init(INIT_DONE);
```

After that step, the task is in *nominal state*. It has no way to request some new hardware or software resources.

Nominal state

In this state, the task can use the previously declared resources. All memory mapped devices are mapped in the task memory space, which can therefore access that memory area.

Warning: If a device is configured as a *voluntary mapped* device, its registers are not automatically mapped in the task's memory space. The task needs to voluntarily map it to be able to access it.

The task is no more authorized to execute any `sys_init()` call to the kernel.

Other syscalls can be used:

- `sys_log()` to transmit a message on the kernel logging facility
- `sys_ipc()` syscalls family, to communicate through kernel IPC with other tasks
- `sys_cfg()` syscalls family, to (re)configure previously declared devices and DMA
- `sys_get_systick()` to get time stamping information
- `sys_yield()` to voluntarily release the CPU core and sleep until an external event arises (IRQ or IPC targeting the task)
- `sys_sleep()` to voluntarily release the CPU core and sleep for a given number of milliseconds
- `sys_reset()` to voluntarily reset the SoC
- `sys_lock()` to voluntarily lock a critical section and postpone the task's ISR for some time

3.2.2 Declaring and initializing resources

Declaring and initializing a device

Before using a device, a task must declare and initialize it. Declaring and initializing a DMA stream is a particular case (see below).

The device structure is the following:

```
typedef struct {
    char          name[16];          /**< device name */
    physaddr_t    address;           /**< device base address */
    uint16_t      size;              /**< device size (in bytes) */
    uint8_t       irq_num;           /**< number of device IRQs */
    uint8_t       gpio_num;          /**< number of device associated GPIOs */
    dev_irq_info_t irqs[MAX_IRQS];   /**< table of IRQ management infos */
    dev_gpio_info_t gpios[MAX_GPIOIS]; /**< table of GPIO configurations */
} device_t;
```

The `device_t` structure is composed by:

- The `name` field contains a name, used to ease debugging
- The `address` and the `size` contains the MMIO address space, as defined in the datasheet
- The `irqs` and `gpios` define a list of IRQs and GPIO pins (see below)

The device is then declared and initialized by using the `sys_init(INIT_DEVACCESS)` syscall (see [sys_init](#), [initializing devices](#)). It is submitted to a set of permissions (see [EwoK permissions](#)).

The device is activated, including the RCC line(s), when the task ends its initialization phase by calling `sys_init(INIT_DONE)`.

Note: A device can be declared and initialized by only one task.

Warning: Ada kernel is very strict with the syscall arguments types conformance. When passing structures, it is highly recommended to `memset` them to 0 before setting their content, otherwise the kernel will probably return `SYS_E_INVALID`.

Declaring a GPIO pin

GPIOs connect the SoC to the outside world (peripherals, buttons, leds, etc.) Even if GPIO ports are devices per se (they are memory mapped, with their own registers), EwoK never allows to directly map them in the user space. A GPIO port controls several *pins* in a single register. A device usually needs to control, at most, only some few pins. Thus, GPIO ports are shared resources and the access to the pins are managed and mediated by the kernel.

The `dev_gpio_info_t` structure is the following:

```
typedef struct {
    gpio_mask_t      mask;
    gpio_ref_t       kref;
    gpio_mode_t      mode;
    gpio_pupd_t      pupd;
    gpio_type_t      type;
    gpio_speed_t     speed;
    uint32_t         afr;
    uint32_t         lck;
    gpio_exti_trigger_t exti_trigger;
    gpio_exti_lock_t  exti_lock;
    user_handler_t    exti_handler;
} dev_gpio_info_t;
```

The `mode`, `pupd`, `type`, `speed` and `afr` are usual information about a GPIO pin. The configuration `mask` allows to configure only some of these fields (e.g. if there is no alternate function to configure).

The `kref` field the GPIO port/pin couple.

Here is an example of some GPIO pins declaration:

```
usart_dev.gpios[0].mask =
    GPIO_MASK_SET_MODE | GPIO_MASK_SET_TYPE | GPIO_MASK_SET_SPEED |
    GPIO_MASK_SET_PUPD | GPIO_MASK_SET_AFR;

usart_dev.gpios[0].kref.port = GPIO_PA;
usart_dev.gpios[0].kref.pin = 6;

usart_dev.gpios[0].type = GPIO_PIN_OTYPER_PP;
usart_dev.gpios[0].pupd = GPIO_NOPULL;
usart_dev.gpios[0].mode = GPIO_PIN_ALTERNATE_MODE;
usart_dev.gpios[0].speed = GPIO_PIN_VERY_HIGH_SPEED;
usart_dev.gpios[0].afr = GPIO_AF_USART1;

usart_dev.gpios[1].mask =
    GPIO_MASK_SET_MODE | GPIO_MASK_SET_TYPE | GPIO_MASK_SET_SPEED |
    GPIO_MASK_SET_PUPD | GPIO_MASK_SET_AFR;

usart_dev.gpios[1].kref.port = GPIO_PA;
usart_dev.gpios[1].kref.pin = 7;

usart_dev.gpios[1].afr = GPIO_AF_USART1;
usart_dev.gpios[1].type = GPIO_PIN_OTYPER_PP;
usart_dev.gpios[1].pupd = GPIO_NOPULL;
usart_dev.gpios[1].mode = GPIO_PIN_ALTERNATE_MODE;
usart_dev.gpios[1].speed = GPIO_PIN_VERY_HIGH_SPEED;
```

GPIOs and external interrupts (EXTI)

GPIOs can be associated to external interrupts (EXTI). This is required to asynchronously detect some external events based on GPIOs such as a button pressed, an event on the touchscreen, etc.

These fields of the `dev_gpio_info_t` structure permit to configure such EXTIs:

- `exti_trigger` specifies the kind of EXTI trigger
- `exti_lock` specifies whether the EXTI line has to be masked each time an EXTI interrupt arises (see `sys_cfg(SYS_CFG_UNLOCK_EXTI)` in *sys_cfg, configuring devices*)
- `exti_handler` has the address of the ISR handler to execute

The IRQ line associated to the EXTI must not be declared: it is already fully managed by the microkernel.

exti_trigger	Description
GPIO_EXTI_TRIGGER_NO_TRIGGER	No trigger (the default)
GPIO_EXTI_TRIGGER_RISING	Trigger only on rising edge (value rising from 0 to 1)
GPIO_EXTI_TRIGGER_FALLING	Trigger only on falling edge (value rising from 1 to 0)
GPIO_EXTI_TRIGGER_BOTH_EDGES	Trigger on both edges

exti_lock	Description
GPIO_EXTI_UNLOCKED	The EXTI interrupt arises normally
GPIO_EXTI_LOCKED	The EXTI line is muted at the first interrupt. No more interrupt on this line arises until the task voluntary unlock the line

Declaring an IRQ

Declaring some IRQ is made through the use of the `dev_irq_info_t` structure:

```
typedef struct {
    user_handler_t      handler;
    uint8_t             irq;
    dev_irq_isr_scheduling_t mode;
    dev_irq_ph_t         posthook;
} dev_irq_info_t;
```

The parameters:

- `handler` stores the address of the user defined ISR handler
- `irq` is the IRQ number, given by the kernel
- `mode` is a special field (described below)
- `posthook_status` and `posthook_data` are described below

For each IRQ, the task must declare an IRQ handler. An IRQ handler takes three parameters:

```
void my_irq_handler (uint8_t irq, uint32_t posthook_status, uint32_t posthook_data);
```

The IRQ handler is executed in *ISR mode*. It has access to the task content except for the stack. It has its own stack, which is erased each time the handler terminates. By default the termination of an ISR handler awakes its related task's main thread if it's sleeping or idle. This behavior can be modified by modifying the `mode` field of the `dev_irq_info_t` structure:

mode	Description
IRQ_ISR_STANDARD	Make main thread runnable
IRQ_ISR_FORCE_MAINTHREAD	Make main thread runnable and force its execution
IRQ_ISR_WITHOUT_INTERRUPT	Do not modify main thread's state

The `IRQ_ISR_FORCE_MAINTHREAD` may be required by devices needing some highly responsive software. Because of the not so negligible impact on the scheduling policy, using this value requires specific permissions.

Note that user ISRs are not executed synchronously:

- ISR treatment is postponed
- Acknowledgement of the hardware device's interrupt is not executed by the user ISR. It is done by the *posthooks*, described hereafter

Acknowledging interrupts with *posthooks*

Posthook mechanism allows to synchronously acknowledge external interrupts, when they are handled by the kernel, before their management is postponed to be managed by a user ISR handler.

Device interrupt acknowledgements may vary from one device to another. They are usually a sequence of reads, writes or masks of some device registers. EwoK provides a small API to make the kernel managing all these in generic and a safe way. Posthook API is mostly used to acknowledge hardware device interrupts.

Posthook action	Description
IRQ_PH_NIL	No action
IRQ_PH_READ	Reading a value from a device's register
IRQ_PH_WRITE	Writing a value into a device's register
IRQ_PH_AND	<ol style="list-style-type: none"> 1. Reads a value from a register (usually a status register) 2. Mask that value to in order to write only active bits 3. Might invert the bits 4. Write the calculated value in a destination register (usually dedicated to acknowledge the interrupt)
IRQ_PH_MASK	<ol style="list-style-type: none"> 1. Reads a value from a register 2. Reads a mask from a register 3. Mask that value to in order to write only active bits 4. Might invert the bits 5. Write the obtained value in a destination register

A device's register is specified as an offset, calculated from the base of the device's memory space.

Hint: The posthook implementation keeps memory of the *read* in order to avoid multiple reads of the same register, which could lead to unexpected behaviors (e.g. ToCToU vulnerability)

As we already see above, an IRQ handler takes three parameters:

```
void my_irq_handler (uint8_t irq, uint32_t posthook_status, uint32_t posthook_data);
```

The `posthook_status` and `posthook_data` parameters may contain values read during the *posthook* action, and ought to be transmitted to the user handler. Most of the time, `posthook_status` stores the value read from a status register while the `posthook_data` stores a value read from another device's register. If the device declares a posthook with (at least) two register read, it can also ask for getting back these registers values as they were at the posthook execution time, by specifying the very same register offset in the `posthook_status` and `data` fields.

Below is an example for the USART driver:

```
usart_dev.irqs[0].posthook.status = 0x0000; /* status register */
usart_dev.irqs[0].posthook.data   = 0x0004; /* data register */

usart_dev.irqs[0].posthook.action[0].instr = IRQ_PH_READ;
usart_dev.irqs[0].posthook.action[0].read.offset = 0x0000; /* reading status register */
↪ */

usart_dev.irqs[0].posthook.action[1].instr = IRQ_PH_READ;
usart_dev.irqs[0].posthook.action[1].read.offset = 0x0004; /* reading data register */

usart_dev.irqs[0].posthook.action[2].instr = IRQ_PH_WRITE;
usart_dev.irqs[0].posthook.action[2].write.offset = 0x0000; /* write to status */
↪ register... */
usart_dev.irqs[0].posthook.action[2].write.value  = 0x00; /* ...the value 0x0 */
usart_dev.irqs[0].posthook.action[2].write.mask   = 0x3 << 6; /* using the given */
↪ write mask
                                                    (clear TC & Tx */
↪ status in SR register) */
```

Caution:

- When declaring posthooks, you can only use offsets based on current device base address
- The offsets must be a part of the device address map
- The posthook sanitation is done at device declaration time, posthooks cannot be modified

Declaring and initializing a DMA stream

A DMA controller is shared among several devices. Thus, its access by the tasks is mediated by the kernel.

EwoK allows only *memory-to-peripheral* and *peripheral-to-memory* DMA usage. *Memory-to-memory* is not safe enough and is forbidden in EwoK (since the DMA controller bypasses the MPU controller, which is obviously very dangerous).

A task can request multiple DMA streams. Note that it is possible to reconfigure the previously configured stream after the initialization phase.

The `dma_t` structure is the following:

```
typedef struct {
    physaddr_t in_addr;          /* DMA input base address */
    physaddr_t out_addr;        /* DMA output base address */
    dma_prio_t in_prio;         /* DMA priority for memory to peripheral */
    dma_prio_t out_prio;        /* DMA priority for peripheral to peripheral */
    uint16_t size;              /* DMA buffer size to copy (in bytes) */
    uint8_t dma;                /* DMA controller identifier */
    uint8_t channel;            /* DMA channel to configure */
    uint8_t stream;             /* DMA stream to configure */
    dma_flowctrl_t flow_control; /* DMA Flow controller */
    dma_dir_t dir;              /* Current DMA direction */
    dma_mode_t mode;            /* Current DMA mode */
    bool mem_inc;               /* DMA incremental mode for memory */
    bool dev_inc;               /* DMA incremental mode for device */
    dma_datasize_t datasize;    /* data unit size */
    dma_burst_t mem_burst;      /* type of DMA burst mode */
    dma_burst_t dev_burst;      /* type of DMA burst mode */
    user_dma_handler_t in_handler; /* DMA ISR for memory to peripheral */
    user_dma_handler_t out_handler; /* DMA ISR for peripheral to memory */
} dma_t;
```

A task declaring a `dma_t` structure does not have to fill all the fields. The `in_handler`, `out_handler`, `in_addr`, `out_addr` and `size` can be set later, in *nominal mode*. The reason is that a single stream can be used for sending or receiving data.

Here is a typical declaration used in the SDIO stack:

```
dma.channel = DMA2_CHANNEL_SDIO;
dma.dir = MEMORY_TO_PERIPHERAL; /* write by default */
dma.in_addr = (physaddr_t) 0; /* to set later via DMA_RECONF */
dma.out_addr = (volatile physaddr_t)sdio_get_data_addr();
dma.in_prio = DMA_PRI_HIGH;
dma.dma = DMA2;
dma.size = 0; /* to set later via DMA_RECONF */

dma.stream = DMA2_STREAM_SDIO_FD;
```

(continues on next page)

(continued from previous page)

```

dma.mode = DMA_FIFO_MODE;
dma.mem_inc = 1;
dma.dev_inc = 0;
dma.datasize = DMA_DS_WORD;
dma.mem_burst = DMA_BURST_INC4;
dma.dev_burst = DMA_BURST_INC4;
dma.flow_control = DMA_FLOWCTRL_DEV;
dma.in_handler = (user_dma_handler_t) sdio_dmacallback;
dma.out_handler = (user_dma_handler_t) sdio_dmacallback;

ret = sys_init(INIT_DMA, &dma, &dmadesc);

```

When calling `sys_init(INIT_DMA, &dma, &dmadesc)`, the `dmadesc` identifier is updated with a unique identifier that can be used later by some syscalls.

Manipulating a DMA

When calling `sys_init(INIT_DONE)`, the DMA controller has its clock enabled if it is not already, but the DMA stream is **not** activated. To activate the DMA transfer, the task needs to call `sys_cfg(CFG_DMA_RECONF)`. This syscall will configure all the fields involved in the transfer and launch it if every required field is properly set. This behavior allows the task to activate the DMA at will, e.g. when the input buffer is ready, or after receiving a dedicated IPC.

Reconfiguring a DMA stream

Most of the time, reconfiguring a DMA stream requires to reconfigure `in_addr`, `out_addr` and `size` fields, to set the input/output addresses involved in the DMA transfer and the size of the transfer.

Here is an example of a DMA reconfiguration:

```

dma.out_addr = (physaddr_t)buffer;
dma.size = buf_len;
ret = sys_cfg(CFG_DMA_RECONF, (void*)&dma, DMA_RECONF_BUFOUT | DMA_RECONF_BUFSIZE);

```

The fields that can be reconfigured are the following:

- ISR handlers `in_handler` and `out_handler`
- Input and output addresses `in_addr` and `out_addr`
- Transfer size `size`
- DMA mode (Circular, FIFO, Direct), `mode`
- DMA priority (between other DMA controller tasks), `in_prio` and `out_prio`
- DMA direction, `dir`

Note: The DMA circular mode does not require any action from the task as the DMA is then fully autonomous (until the user task requires a DMA reset to stop the DMA action).

DMA direction is allowed to be reconfigured in the case of DMA streams that are used for both device read and write access (e.g. SDIO device on the STM32F4xx boards).

When passing in parameter the `dma_t` structure to the `sys_cfg (CFG_DMA_RECONF)` syscall, a mask is used to specify which fields are updated.

Reloading a DMA stream

In DMA circular mode, the controller never stops transferring data. It is possible to stop this active stream by using the `sys_cfg (CFG_DMA_DISABLE)` syscall.

Then, the task may reactivate this very same stream by using the `sys_cfg (CFG_DMA_RELOAD)` syscall.

Declaring and initializing a DMA SHM

Sometimes, a dataplane may be implemented using multiple tasks communicating with each others. When the internal device dataplane is manipulating DMA streams, the tasks may wish to optimize the data buffer transfer by using only DMA transfers between them instead of using manual buffer copy through IPC.

For this case, EwoK allows tasks to voluntarily share a memory buffer. One of the task, the caller, owns that memory buffer, mapped in its address space.

The other task, the receiver, will then be able to request DMA transaction *from* or *toward* this memory buffer, from a given hardware device (e.g. CRYPT, HASH, or any device that reads data stream through DMA requests as input). Note that this memory buffer is not mapped in the receiver's memory space and the receiver can therefore never read from or write to it.

Sharing a memory buffer by this mean is subject to specific permissions.

Note: DMA SHM declaration is often associated with IPCs to let the *caller* inform the *receiver* of the buffer address and size

Here is a typical usage of DMA SHM buffer:

```
const uint32_t bufsize = 4096;
buf[bufsize] = { 0 };

dma_shm_t dmashm_rd;

dmashm_rd.target = id_receiver;
dmashm_rd.source = task_id;
dmashm_rd.address = (physaddr_t)flash_buf;
dmashm_rd.size = bufsize;
/* Receiver can only create DMA request *from* this buffer (read only) */
dmashm_rd.mode = DMA_SHM_ACCESS_RD;

printf("Declaring DMA_SHM for read flow\n");
ret = sys_init(INIT_DMA_SHM, &dmashm_rd);
printf("sys_init returns %s !\n", strerror(ret));

sys_init(INIT_DONE);
```

3.3 EwoK permissions

Contents

- *EwoK permissions*
 - *General principle*
 - *Configuring the permissions*
 - * *Menuconfig*
 - *Devices*
 - *Time*
 - *Tasking*
 - *Memory management*
 - * *IPCs*

3.3.1 General principle

Permissions are statically set at configuration time, before building the firmware, and cannot be updated during the device life-cycle. Each application permission is stored in a *.rodata* part of the kernel, reducing the risk of any tampering with.

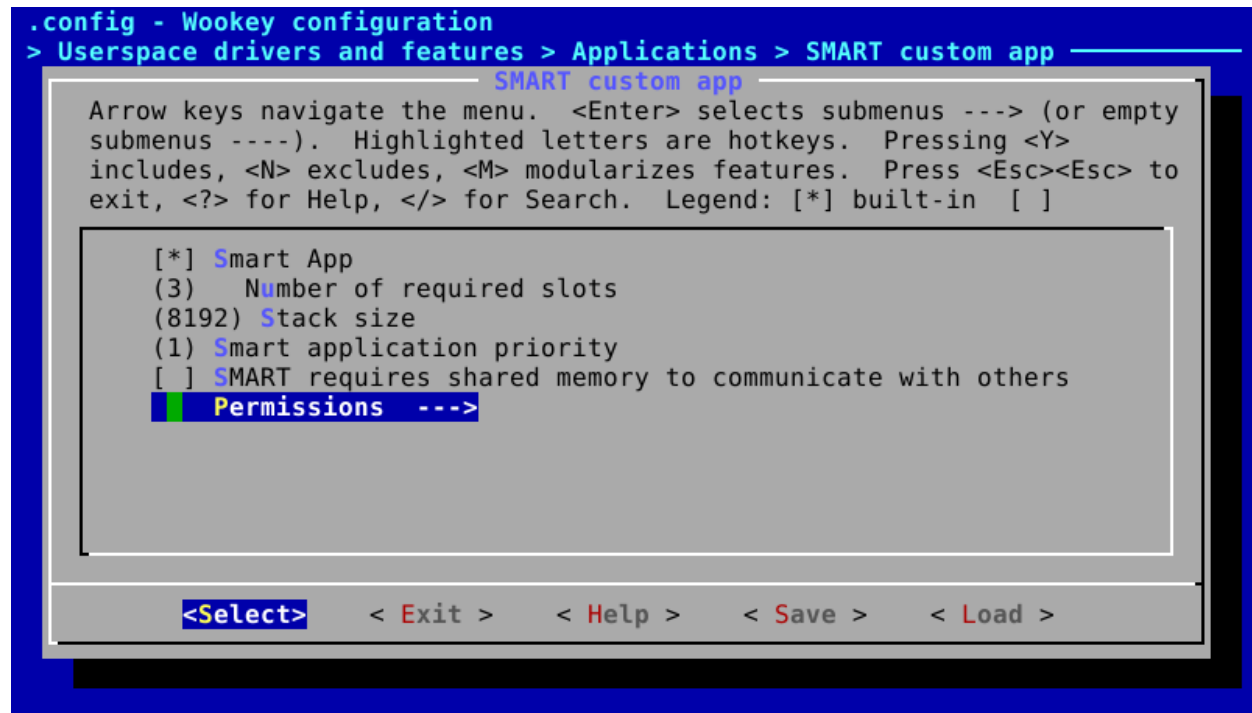
3.3.2 Configuring the permissions

Permissions are configured by using two complementary means:

- The whole permissions, except IPCs, are set using `menuconfig`
- IPCs are configured by editing `apps/ipc.config` and `apps/dmashm.config` files

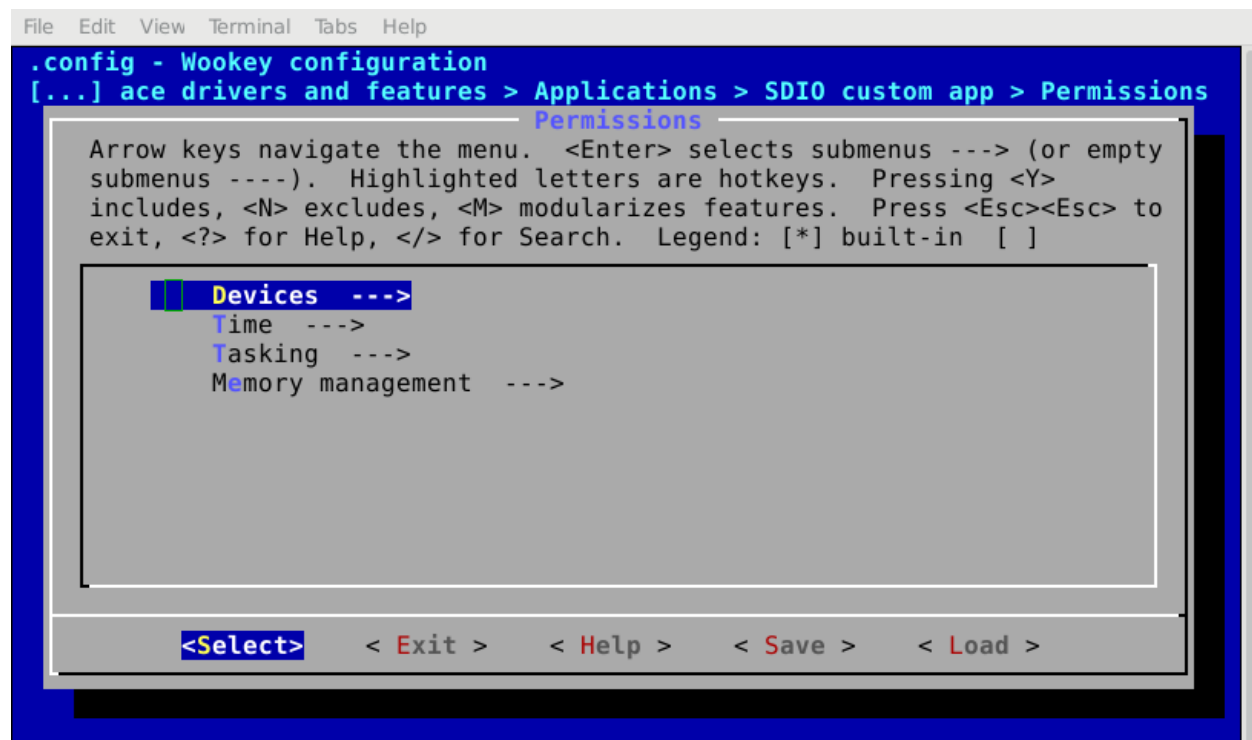
Menuconfig

Each application has its own permissions set by menuconfig:



Permissions are separated into 4 families:

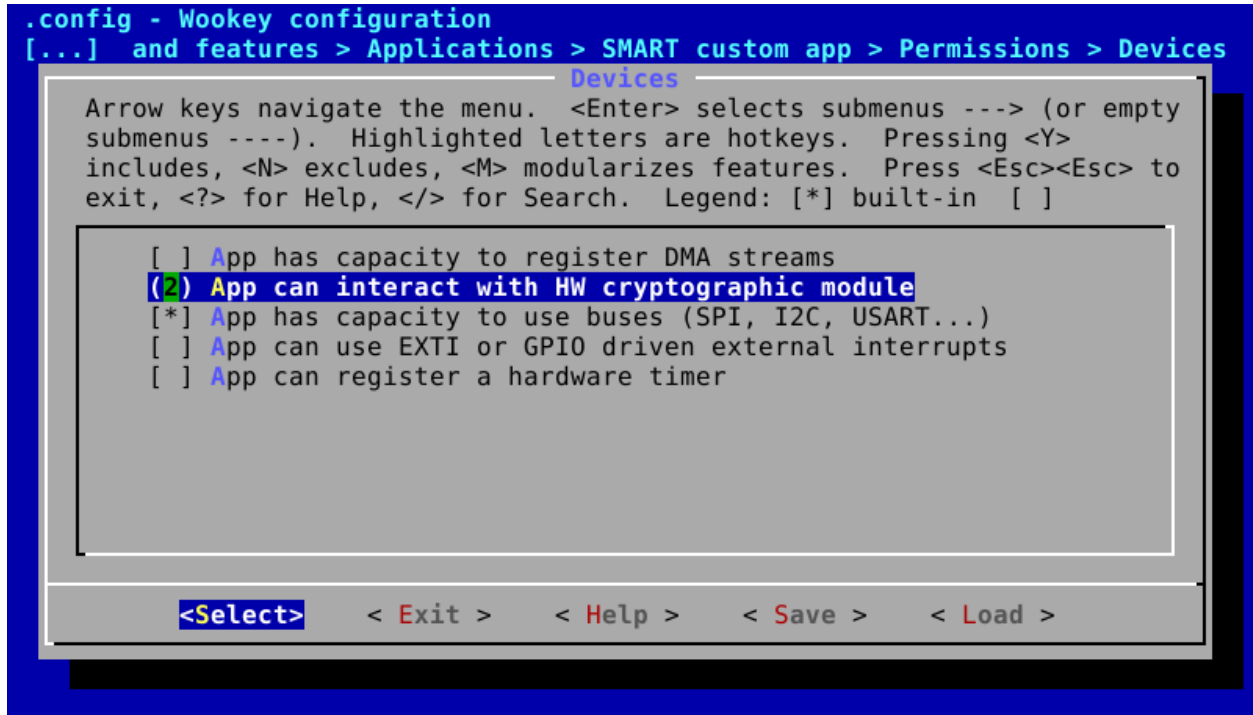
- Devices
- Time
- Tasking
- Memory management



Devices

Devices permissions controls the capability to:

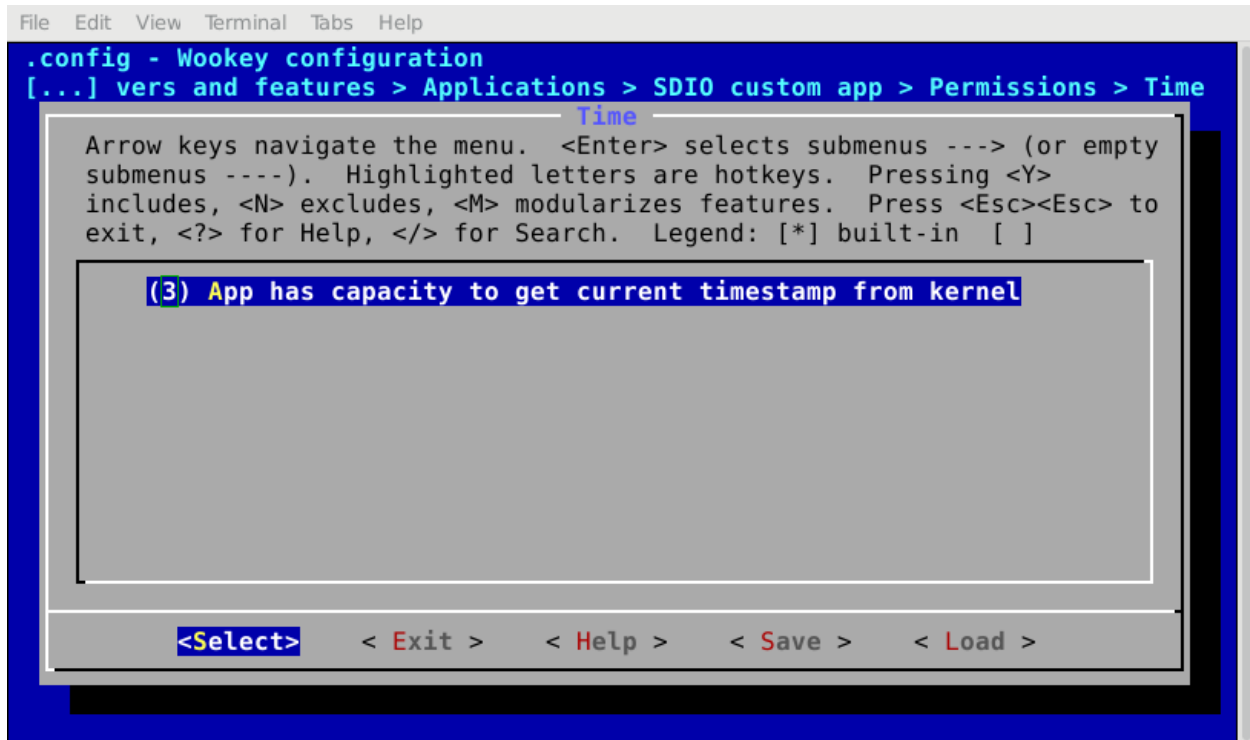
- Use DMA streams
- Use the hardware cryptographic module
- Use buses (SPI, I2C, etc.)
- Use EXTIs
- Use a hardware timer



Warning: Devices permissions impact the `sys_init(INIT_DEVACCESS)` syscall

Time

Time controls the capability to use the kernel internal clock:



This option allows to specify the granted granularity of the timestamp returned by the kernel:

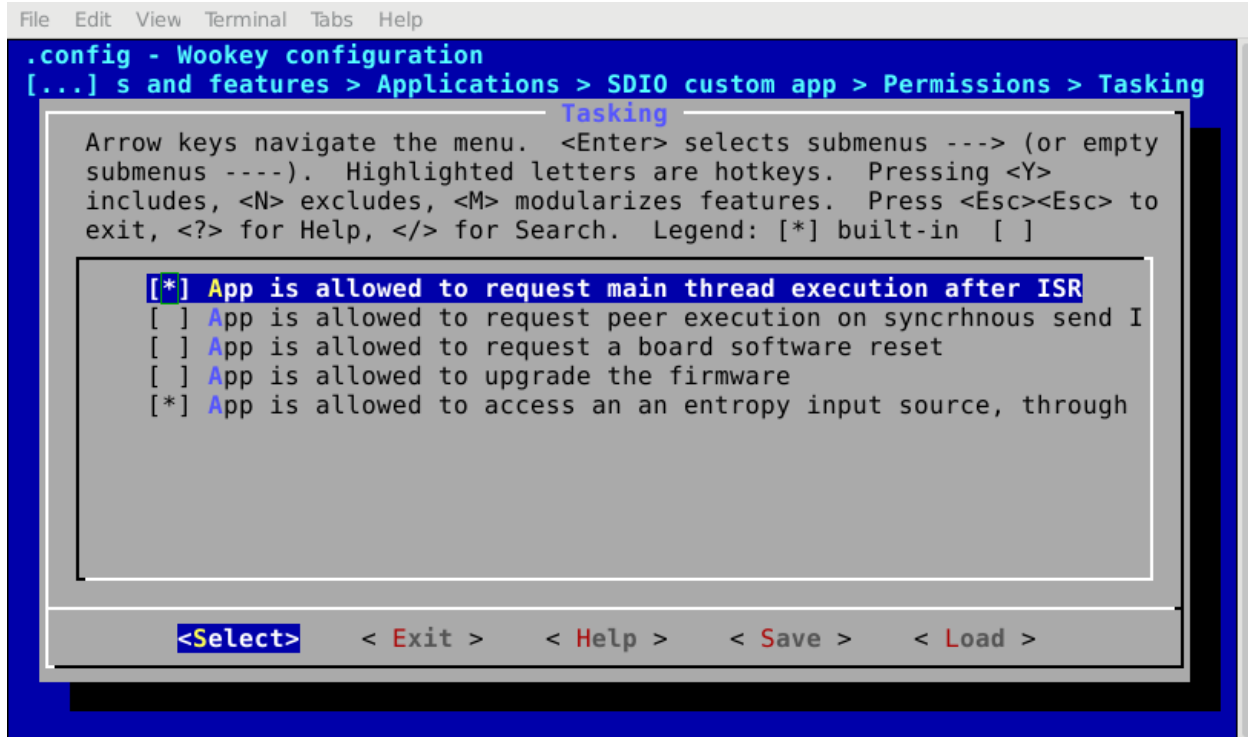
- No time measurement is possible
- Tick granularity
- Microsecond granularity
- CPU cycle granularity

Warning: Time permissions impact the `sys_get_systick()` syscall

Tasking

Tasking controls the capability to:

- When an ISR exit, its main thread is scheduled, bypassing the default scheduling policy. This is needed by devices requiring a high responsiveness (e.g. smart cards over IS7816-3 buses)
- When the task sends an IPC, if the target task is idle or runnable, it is immediately scheduled, bypassing the scheduling policy
- Reset the board
- Upgrade the firmware
- Access to randomness generated by the kernel (and relying on the hardware RNG)

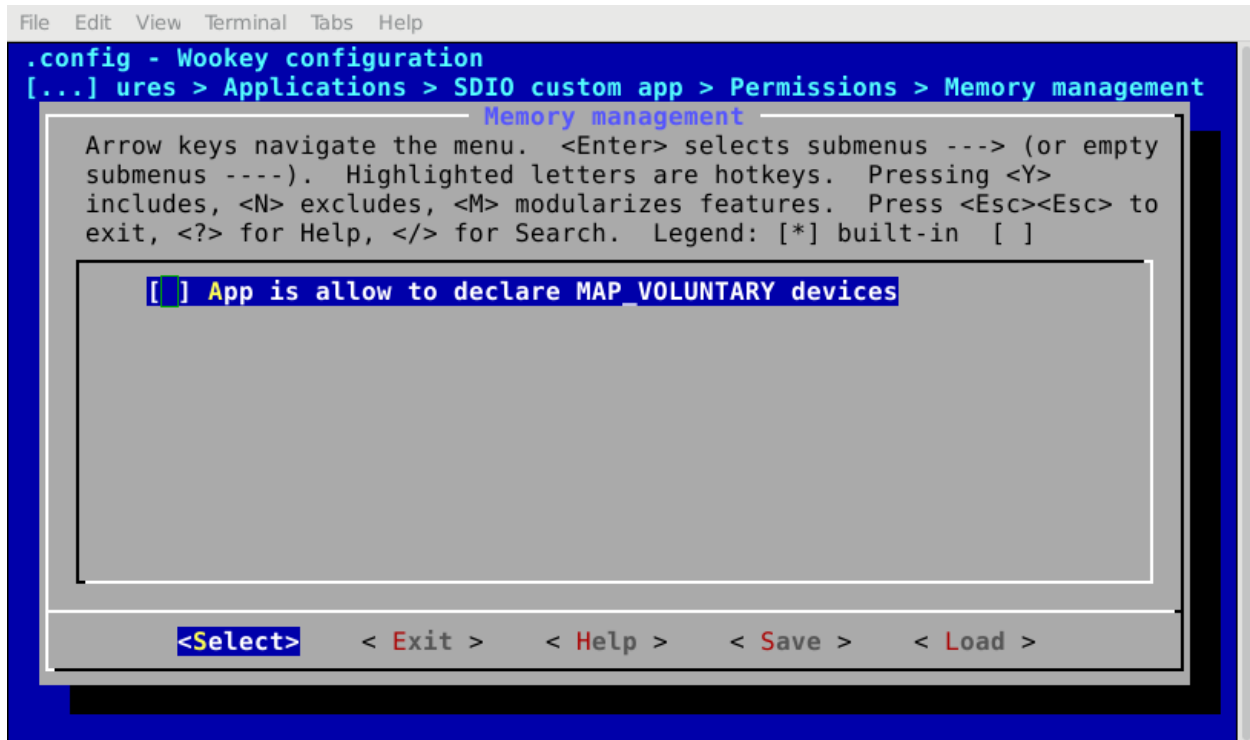


Warning: Devices permissions impact the `sys_init (INIT_DEVACCESS)` and `sys_reset ()` syscalls

Memory management

Memory management controls the capability to voluntary map or unmap a device in the task's address space.

This does not permit to declare a new device, but only to temporary (un)map it from the task's address space, if the driver supports this feature.



Warning: Devices permissions impact the `sys_init (INIT_DEVACCESS)` syscall

IPCs

Communication permissions are based on two arrays, found in plain-text files:

- The array in `apps/ipc.config` is used to set the permissions for using the IPC mechanism
- The array in `apps/dmashm.config` is used to set the permissions for using the DMA shared memory mechanism

IPC array is in `apps/ipc.config`. The sender is on the left column. Setting 1 in a box means that the task on the left is able to send a message using IPCs to the one above:

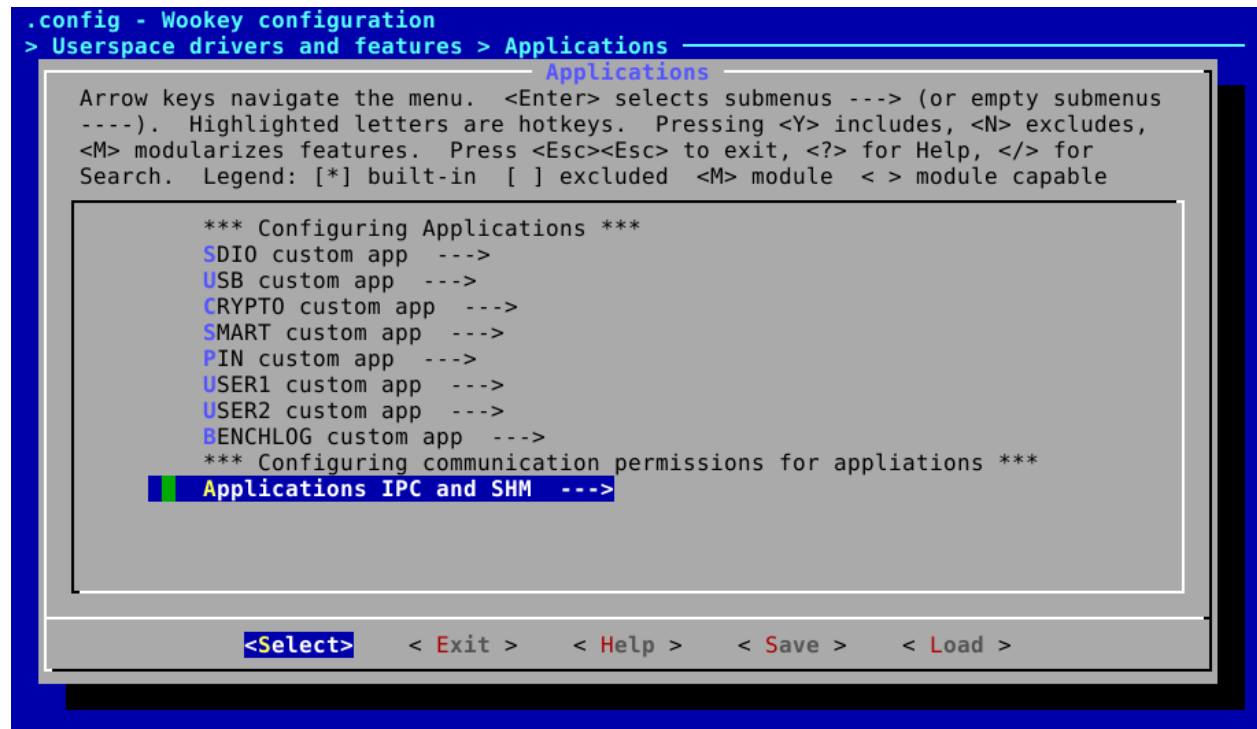
```
comment "----- SDIO  USB  CRYPTO  SMART  PIN"
comment "SDIO    [#]  [1]  [ ]  [ ]  [ ]"
comment "USB      [ ]  [#]  [ ]  [ ]  [ ]"
comment "CRYPTO     [ ]  [ ]  [#]  [ ]  [ ]"
comment "SMART     [ ]  [ ]  [ ]  [#]  [ ]"
comment "PIN       [ ]  [ ]  [ ]  [ ]  [#]"
```

Warning: A task is not allowed to send IPC to itself

DMA shared memory array is in `apps/dmashm.config`. The “caller” is on the left column. A mark in a box means that the task on the left (the “caller”) is able to share a buffer with another task (the “granted”). The task selected on the right columns are granted to use a buffer in “caller” address space for DMA transfers:

```
comment "----- SDIO  USB  CRYPTO  SMART  PIN"
comment "SDIO    [#]  [ ]  [ ]  [ ]  [ ]"
comment "USB      [ ]  [#]  [ ]  [ ]  [ ]"
comment "CRYPTO    [ ]  [ ]  [#]  [ ]  [ ]"
comment "SMART    [ ]  [ ]  [ ]  [#]  [ ]"
comment "PIN      [ ]  [ ]  [ ]  [ ]  [#]"
```

Note that menuconfig displays those arrays, but without the possibility to modify them: you will have to edit the associated files manually.



```
.config - Wookey configuration
> Userspace drivers and features > Applications > Applications IPC and SHM
    Applications IPC and SHM
    Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus
    ----).  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M>
    modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.
    Legend: [*] built-in  [ ] excluded  <M> module  <> module capable

    -*- Allow communicating using kernel IPC
    *** ----- SDIO  USB CRYPTO SMART PIN ***
    *** SDIO    [#]  [ ]  [1]  [ ]  [ ] ***
    *** USB     [ ]  [#]  [1]  [ ]  [ ] ***
    *** CRYPTO  [1]  [1]  [#]  [1]  [ ] ***
    *** SMART   [ ]  [ ]  [1]  [#]  [2] ***
    *** PIN     [ ]  [ ]  [ ]  [2]  [#] ***
    [*] Allow Sharing DMA buffer between tasks
    *** ----- SDIO  USB CRYPTO SMART PIN ***
    *** SDIO    [#]  [ ]  [1]  [ ]  [ ] ***
    *** USB     [ ]  [#]  [1]  [ ]  [ ] ***
    *** CRYPTO  [1]  [1]  [#]  [ ]  [ ] ***
    *** SMART   [ ]  [ ]  [ ]  [#]  [ ] ***
    *** PIN     [ ]  [ ]  [ ]  [ ]  [#] ***

    <Select>  < Exit >  < Help >  < Save >  < Load >
```

Warning: A task is not allowed to declare DMA SHM to itself

INTERNALS

4.1 Ada/SPARK for a secure kernel

The EwoK microkernel is an Ada/SPARK kernel with very few lines of C.

4.1.1 Why implementing Ewok in Ada?

Most kernels and microkernels are written in C. The major drawback of the C language is its proneness to coding errors. Out-of-bound array accesses, integer overflows and dangling pointers are difficult to avoid due to the weakly enforced typing. Such bugs can become nonetheless devastating when exploited in a privileged context. A way to prevent such vulnerabilities is to use a safe language or to use formal methods to prove the lack of runtime error.

Using a safe language for implementing low-level kernel code is an approach that goes back to the early 1970's. However, there are very few alternatives and we made the choice of [Ada](#), designed for building high-confidence and safety-critical applications in embedded systems.

Note: The Ada/SPARK kernel is based on about 10 Klines of Ada and about 500 lines of C and assembly.

4.1.2 From C to Ada

Interoperability between C and Ada is facilitated by GNAT providing a [full interface to C](#).

Importing C symbols in Ada

Importing a C symbol in an Ada program is done using the following directive:

```
function my_ada_function ( myarg : unsigned_8) return unsigned_32
with
  convention      => c,
  import          => true,
  external_name   => "my_c_function",
  global          => null;
```

Using this directive, the symbol resolved by `my_c_function` in the C object file can be used using `my_ada_function` in the Ada implementation.

When importing a C function, it is required to comply with less restrictive types such as `unsigned_32`, `unsigned_8` or bit-length boolean (Ada booleans are bigger types).

To do so, writing a C types specification for Ada is highly recommended. EwoK keeps its C types for Ada in the Ada types.c unit of the libbsp.

As using C symbols makes Ada strict typing and SPARK inefficient, their usage must be reduced to a **small and controlled subset of the Ada code**.

In the EwoK case, using C symbols is reduced to the Ada/C interface unit only. This interface has no algorithmic intelligence but must take care of the overtyped C arguments when using C symbols.

A typical usage would be, for the following C code:

```
uint8_t nvic_get_pending_irq()
{
    ... // return the IRQ number as an uint8_t
}
```

An Ada interface that could look like the following:

```
with ada.unchecked_conversion;
pragma warnings (off);
function to_t_interrupt is new ada.unchecked_conversion
(unsigned_8, t_interrupt);
pragma warnings (on);

-- t_interrupt is an Ada type listing only the effective existing
-- IRQs (IRQ 1 to IRQ 96 for e.g.)
function get_interrupt(irq : out t_interrupt)
is
    local_irq : unsigned_8;
begin
    local_irq := nvic_get_pending_irq();
    if local_irq in t_interrupt'range then
        irq = to_t_interrupt(local_irq);
    else
        -- raise exception or react in any way
    end if;
end
```

Exporting Ada symbols to C

Exporting Ada symbols to C is done using the same philosophy:

```
-- initialize the DWT module
-- This procedure is called by the kernel main() function, and as
-- a consequence exported to C
procedure init
with
    convention => c,
    export => true,
    external_name => "soc_dwt_init";
```

Nevertheless, there are some cases that require extra care and attention: **when specific types are handled differently in Ada and C**. This is the case of strings, which are more complex and **not** null-terminated in Ada, or boolean, which are encoded on 8-bits fields.

To solve such an issue, we define for the Ada code some C-compatible types. Here is an example of a C compatible boolean implementation:


```
type bool is new boolean with size => 1;
for bool use (true => 1, false => 0);
```

4.1.3 Ada sources

EwoK Ada sources are hosted in the following directories:

- kernel/Ada for the kernel, arch-independent Ada code
- kernel/Ada/generated hosts the generated Ada files, like kernel/generated hosts the generated C files
- arch-specific Ada content (BSP) is hosted in the Ada subdirectory of each SoC and core source directory

Ada has a hierarchical scoping principle, based on packages. In the case of EwoK, various packages and subpackages are used.

- kernel packages belong to the *ewok* package
- SoC-related packages belong to the *soc* package
- Core-related packages belong to the core-relative package (e.g. *m4* for Cortex-M4)

EwoK kernel is implemented with a little bit of C. Thus, some Ada/SPARK packages require an external interface with the C code. For a given package *foo* interacting with external C code, a *foo_interface* package must be defined.

In the same way, as some various C types (structures, union, enumerates, etc.) have to be used in the interfaces packages, the following C-specific packages exist, containing only specifications:

- *c* package containing all C types and API that are arch-independent
- *c_soc* package, containing all C types and API that are SoC-specific

4.1.4 Preprocessing in Ada

Ada does support preprocessing and the configuration options sometime use the preprocessing principle to enable or not some specific functions. The preprocessing usage is quite similar to C:

```
#if CONFIG_KERNEL_DOMAIN
function is_same_domain
  (from   : in t_real_task_id;
   to     : in t_real_task_id)
return boolean
with
  Global   => null,
  Post     => (if (from = to) then is_same_domain'Result = false);
#end if;
```

4.1.5 Generated files

Generated files are not created by the microkernel internal tools, but by the SDK. The reason is that the generated files contain information about the applications list, associated permissions and layout. All these information are stored by the SDK configuration mechanism, not by the kernel itself.

The scripts generating these files (and the C equivalent) are hosted in the *tools/* directory of the SDK:

- *tools/gen_ld*: generates the global layout and the application layout header

- tools/gen_symhdr.pl: generates the applications section mapping. Used to map .data and zeroify .bss of each application at boot time
- tools/apps/permissions.pl: generates the application permissions header

4.1.6 Static verification with SPARK

SPARK allows to prove the lack of *Run Time Errors* in some code.

EwoK uses **SPARK** in the modules requiring formal validation and proofs. Example:

```
function ipc_is_granted
  (from      : in t_real_task_id;
   to        : in t_real_task_id)
  return boolean
  with
    Global      => (Input => ewok.perm_auto.com_ipc_perm),
    Post        => (if (from = to) then ipc_is_granted'Result = false),
    Contract_Cases => (ewok.perm_auto.com_ipc_perm(from,to) => ipc_is_granted
  ↪ 'Result,
                                others                                => not ipc_is_
  ↪ granted'Result);
```

This specification uses various SPARK *contracts*:

- Contract_Case describes the contract that must be satisfied by the subprogram
- Global describes the global variables used by a subprogram
- Postcondition indicates conditions that must be satisfied when the program has completed.

SPARK in Ewok

With SPARK, we proved that the kernel never maps a memory region which can be both writable and executable (W^X security principle).

4.2 About EwoK permissions internals

4.2.1 EwoK permissions kernel API

Inside the kernel, the permission module exposes a very simple API, in order to support permission management modularity. This permits, in a future work, to add various permission models using the same API to other parts of the kernel.

The permission API defines a list of resource names:

Permission name	Description
PERM_RES_DEV_DMA	access to DMA devices
PERM_RES_DEV_CRYPTO_USR	user access to hardware cryptographic devices
PERM_RES_DEV_CRYPTO_CFG	full access to hardware cryptographic devices (key injection granted)
PERM_RES_DEV_BUSES	access to buses (USART, I2C, SPI...)
PERM_RES_DEV_EXTI	access to external interrupts
PERM_RES_DEV_TIM	access to timers
PERM_RES_TIM_GETMILLI	access to timestamp with tick precision
PERM_RES_TIM_GETMICRO	access to timestamp with microsecond precision
PERM_RES_TIM_GETCYCLE	access to timestamp with CPU cycle precision
PERM_RES_TSK_FISR	task's ISRs are able to require main thread immediate execution
PERM_RES_TSK_FIPC	task's IPC send is able to require target thread immediate execution
PERM_RES_TSK_RESET	task is able to request immediate SoC reset
PERM_RES_TSK_UPGRADE	task is able to upgrade the firmware (i.e. map the internal flash)
PERM_RES_TSK_RNG	task is able to request random data from the kernel RNG source
PERM_RES_MEM_DYNAMIC_MAP	task is able to (un)map its own devices declared as voluntary mapped

Resource names are define using the preprocessor in C, mapped as an `uint32_t`. In Ada, the permission name has its own type.

The permission API also exports the following prototypes:

```
bool perm_ipc_is_granted(e_task_id from,
                        e_task_id to);

bool perm_resource_is_granted(uint32_t  ressource_name,
                              task_t*   task);

bool perm_same_ipc_domain(e_task_id  src,
                          e_task_id  dst);
```

Through this API, it is possible to control all accesses to resources and tasks. This API abstracts the permission memory model described below.

4.2.2 Memory representation of permissions in EwoK

As shown in *Ewok permissions API*, permissions are based on a tree hierarchy.

Permissions are split into two main categories:

- resources access permissions
- communication access permissions

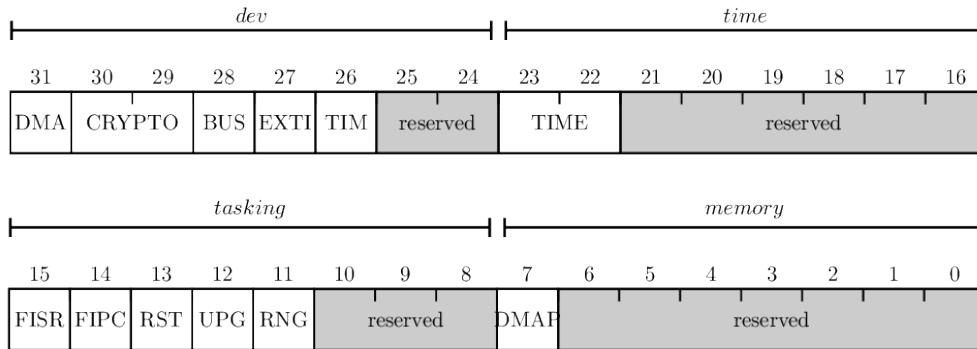
Resource access permissions memory model

Resource access permissions correspond to all permissions associated to an access to a hardware resource (core timeslot, device, current cycle count, etc.).

In order to optimize resources access permissions check at runtime, they are mapped as a *resource permission register*, for each application. This resource permission register works the same way as any hardware register, with bit fields and masks.

The C implementation of such permissions check is not easy as C does not easily manage bitfields, but Ada is clearly more efficient for such checks.

The resource permission register is 32-bit long and has the following mapping:



Checking permissions at run time is done using masks, which allows to optimize permission check time and use boolean constructions.

Booleans are directly mapped as a register bit. Enumerate respects the following structure:

Time permission mapping (2 bits):

- 0b00 : none
- 0b01 : tick permission
- 0b10 : microsecond permission
- 0b11 : cycle permission

Cryptographic IP access mapping (2 bits)

- 0b00 : no access
- 0b01 : data plane access (no key injection)
- 0b10 : configuration access (key injection, RNG access)
- 0b11 : both accesses

The permission register is based on each application permission declaration in the configuration of the Tataouine SDK. The register is created by Tataouine in `include/generated/app_layout.h` (for C code) and in `include/generated/Ada/app_layout.ads` (for Ada code).

The permission register is generated as a static const array of bits denoted `0b110010011100...0001110000` in a dedicated resource permission table in `include/generated/gen_perms.h` by `tools/apps/permissions.pl` script.

The kernel `perm.c/perm.h` (for C) and `perm.adb/perm.ads` file manage the permission register read and return the task permissions based on it.

Communication access permissions memory model

Communications permissions are based on two matrices:

- An IPC matrix, defining which task is able to communicate with which through IPC calls
- A DMA SHM matrix, defining which task is able to share a DMA buffer with which peer

These matrices are generated in `include/generated/gen_perms.h` by `tools/apps/permissions.pl` script.

Here is a typical `gen_perms.h` content:

```

/* ressource register */
typedef uint32_t ressource_reg_t;

static const ressource_reg_t ressource_perm_tab[] = {
    0x10000000, /* benchlog */
    0xc000a000, /* crypto */
    0x90000000, /* pin */
    0x94000000, /* sdio */
    0x50008000, /* smart */
    0x90000000, /* usb */
};

/* ipc communication permissions */
static const bool com_ipc_perm[][6] = {
    {0, 0, 0, 0, 0, 0},
    {0, 0, 0, 1, 1, 1},
    {0, 0, 0, 0, 1, 0},
    {0, 1, 0, 0, 0, 0},
    {0, 1, 1, 0, 0, 0},
    {0, 1, 0, 0, 0, 0}
};

/* dmashm communication permissions */
static const bool com_dmashm_perm[][6] = {
    {0, 0, 0, 0, 0, 0},
    {0, 0, 0, 1, 0, 1},
    {0, 0, 0, 0, 0, 0},
    {0, 1, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0},
    {0, 1, 0, 0, 0, 0}
};

```

The Ada implementation of the permissions is using a strictly typed register instead of a uint32_t bitfield for the resources permissions register. The Ada implementation of EwoK is also using SPARK in order to validate its data flow.

Here is the generated Ada specification

```

package ewok.perm_auto
  with spark_mode => on
is

  -- ressource register definition
  type t_ressource_reg is record
    DEV_DMA      : bit;
    DEV_CRYPT0   : bits_2;
    DEV_BUS      : bit;
    DEV_EXTI     : bit;
    DEV_TIM      : bit;
    DEV_reserved : bits_2;
    TIM_TIME     : bits_2;
    TIM_reserved : bits_6;
    TSK_FISR     : bit;
    TSK_FIPC     : bit;
    TSK_RESET    : bit;
    TSK_UPGRADE  : bit;
    TSK_RANDOM   : bit;

```

(continues on next page)

(continued from previous page)

```

    TSK_reserved      : bits_3;
    MEM_DYNAMIC_MAP   : bit;
    MEM_reserved      : bits_7;
end record
    with Size => 32;

for t_ressource_reg use record
    DEV_DMA           at 0 range 31 .. 31;
    DEV_CRYPT0        at 0 range 29 .. 30;
    DEV_BUS           at 0 range 28 .. 28;
    DEV_EXTI          at 0 range 27 .. 27;
    DEV_TIM           at 0 range 26 .. 26;
    DEV_reserved      at 0 range 24 .. 25;
    TIM_TIME          at 0 range 22 .. 23;
    TIM_reserved      at 0 range 16 .. 21;
    TSK_FISR          at 0 range 15 .. 15;
    TSK_FIPC          at 0 range 14 .. 14;
    TSK_RESET         at 0 range 13 .. 13;
    TSK_UPGRADE       at 0 range 12 .. 12;
    TSK_RANDOM        at 0 range 11 .. 11;
    TSK_reserved      at 0 range 8 .. 10;
    MEM_DYNAMIC_MAP   at 0 range 7 .. 7;
    MEM_reserved      at 0 range 0 .. 6;
end record;

type t_com_matrix is
    array (t_real_task_id'range, t_real_task_id'range) of Boolean;

ressource_perm_register_tab : array (t_real_task_id'range) of t_ressource_reg :=
(
    -- ressource_perm_register for CRYPTO
    ID_APP1 => (
        DEV_DMA           => 1,
        DEV_CRYPT0        => 1,
        DEV_BUS           => 0,
        DEV_EXTI          => 0,
        DEV_TIM           => 0,
        DEV_reserved      => 0,
        TIM_TIME          => 2,
        TIM_reserved      => 0,
        TSK_FISR          => 1,
        TSK_FIPC          => 0,
        TSK_RESET         => 0,
        TSK_UPGRADE       => 0,
        TSK_RANDOM        => 0,
        TSK_reserved      => 0,
        MEM_DYNAMIC_MAP   => 0,
        MEM_reserved      => 0),
    -- ressource_perm_register for PIN
    ID_APP2 => (
        DEV_DMA           => 1,
        DEV_CRYPT0        => 0,
        DEV_BUS           => 1,
        DEV_EXTI          => 0,
        DEV_TIM           => 0,
        DEV_reserved      => 0,
        TIM_TIME          => 1,

```

(continues on next page)

(continued from previous page)

```

TIM_reserved    => 0,
TSK_FISR        => 0,
TSK_FIPC        => 0,
TSK_RESET       => 0,
TSK_UPGRADE     => 0,
TSK_RANDOM      => 1,
TSK_reserved    => 0,
MEM_DYNAMIC_MAP => 0,
MEM_reserved    => 0),
-- ressource_perm_register for SDIO
ID_APP3 => (
DEV_DMA         => 1,
DEV_CRYPT0      => 0,
DEV_BUS         => 1,
DEV_EXTI        => 0,
DEV_TIM         => 1,
DEV_reserved    => 0,
TIM_TIME        => 3,
TIM_reserved    => 0,
TSK_FISR        => 1,
TSK_FIPC        => 0,
TSK_RESET       => 0,
TSK_UPGRADE     => 0,
TSK_RANDOM      => 0,
TSK_reserved    => 0,
MEM_DYNAMIC_MAP => 0,
MEM_reserved    => 0),
-- ressource_perm_register for SMART
ID_APP4 => (
DEV_DMA         => 1,
DEV_CRYPT0      => 2,
DEV_BUS         => 1,
DEV_EXTI        => 1,
DEV_TIM         => 0,
DEV_reserved    => 0,
TIM_TIME        => 3,
TIM_reserved    => 0,
TSK_FISR        => 1,
TSK_FIPC        => 0,
TSK_RESET       => 1,
TSK_UPGRADE     => 0,
TSK_RANDOM      => 1,
TSK_reserved    => 0,
MEM_DYNAMIC_MAP => 0,
MEM_reserved    => 0),
-- ressource_perm_register for USB
ID_APP5 => (
DEV_DMA         => 1,
DEV_CRYPT0      => 0,
DEV_BUS         => 1,
DEV_EXTI        => 0,
DEV_TIM         => 0,
DEV_reserved    => 0,
TIM_TIME        => 3,
TIM_reserved    => 0,
TSK_FISR        => 1,
TSK_FIPC        => 0,

```

(continues on next page)

(continued from previous page)

```

    TSK_RESET      => 0,
    TSK_UPGRADE    => 0,
    TSK_RANDOM     => 0,
    TSK_reserved   => 0,
    MEM_DYNAMIC_MAP => 0,
    MEM_reserved   => 0));

CRYPTO : constant t_real_task_id := ID_APP1;
PIN    : constant t_real_task_id := ID_APP2;
SDIO   : constant t_real_task_id := ID_APP3;
SMART  : constant t_real_task_id := ID_APP4;
USB    : constant t_real_task_id := ID_APP5;

-- ipc communication permissions
com_ipc_perm : constant t_com_matrix :=
  (CRYPTO      => (ID_APP1 => false, ID_APP2 => false, ID_APP3 => true, ID_
↪APP4 => true, ID_APP5 => true),
   PIN        => (ID_APP1 => false, ID_APP2 => false, ID_APP3 => false, ID_APP4 =>
↪true, ID_APP5 => false),
   SDIO       => (ID_APP1 => true, ID_APP2 => false, ID_APP3 => false, ID_APP4 =>
↪false, ID_APP5 => false),
   SMART     => (ID_APP1 => true, ID_APP2 => true, ID_APP3 => false, ID_APP4 =>
↪false, ID_APP5 => false),
   USB       => (ID_APP1 => true, ID_APP2 => false, ID_APP3 => false, ID_APP4 =>
↪false, ID_APP5 => false));

-- dmashm communication permissions
com_dmashm_perm : constant t_com_matrix :=
  (CRYPTO      => (ID_APP1 => false, ID_APP2 => false, ID_APP3 => true, ID_
↪APP4 => false, ID_APP5 => true),
   PIN        => (ID_APP1 => false, ID_APP2 => false, ID_APP3 => false, ID_APP4 =>
↪false, ID_APP5 => false),
   SDIO       => (ID_APP1 => true, ID_APP2 => false, ID_APP3 => false, ID_APP4 =>
↪false, ID_APP5 => false),
   SMART     => (ID_APP1 => false, ID_APP2 => false, ID_APP3 => false, ID_APP4 =>
↪false, ID_APP5 => false),
   USB       => (ID_APP1 => true, ID_APP2 => false, ID_APP3 => false, ID_APP4 =>
↪false, ID_APP5 => false));

end ewok.perm_auto;

```

Contents

- *EwoK IRQ and ISR internals*
 - *ISR mechanism*
 - *ISR postponing*
 - *Posthooks*

4.3 EwoK IRQ and ISR internals

4.3.1 ISR mechanism

Some hardware devices such as the smart card generate interrupts that must be acknowledged within a very tight time frame to avoid timeouts. Other components like the touch screen put pressure on the kernel with interrupts bursts. To deal with these constraints, we designed a simple yet effective system to quickly acknowledge interrupts and to limit as much as possible the overhead of the user mode drivers.

In EwoK, a driver is typically composed of a main thread, which implements all the driver logic, and one or several *Interrupt Service Routine (ISR)* to handle the hardware interrupts. ISRs execution takes place in user mode, with the associated task permissions and memory layout.

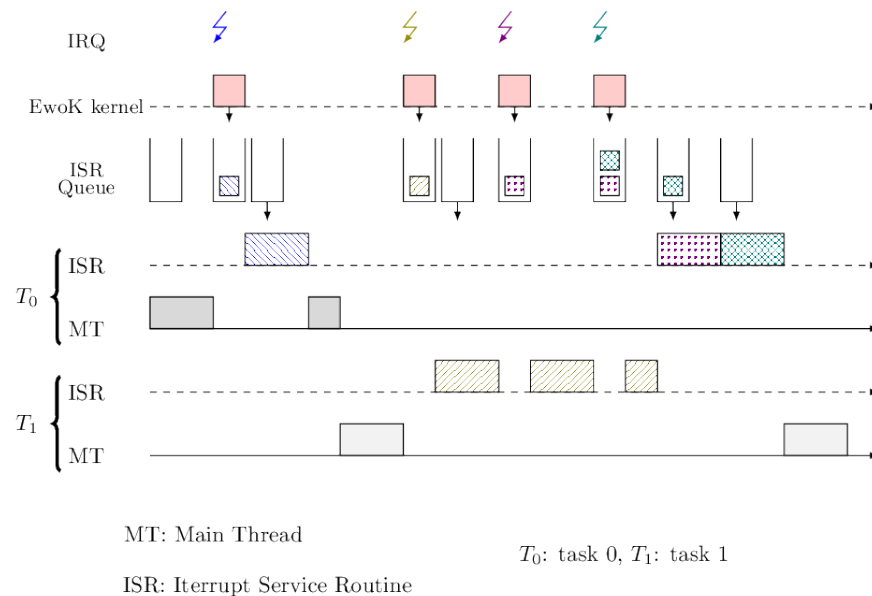
Usually, a user ISR performs only two things: it acknowledges the hardware by reading or writing in some registers and it sets some variables or some shared structures to signal to the main thread that an event happened.

It should be highlighted that a user ISR is scheduled with the highest priority. As a consequence, it must be fast enough to avoid hindering treatment of subsequent hardware interrupts.

4.3.2 ISR postponing

When a hardware interrupt is triggered, the kernel traps it and checks if it must be handled by a user task. If this is the case, the kernel updates a queue structure, managed by the so-called *softirq* module, so that the related user ISR can be scheduled afterward. If an interrupt is triggered while a user ISR is already executing, this interrupt is not lost thanks to the *softirq* queueing mechanism that defers its treatment.

In this design, user ISRs are executed asynchronously. A potential problem is the induced latency in the handling of hardware interrupts.



Previous figure describes a typical scheduling scheme during an IRQ burst. The *posthook* mechanism has been introduced to address this issue.

4.3.3 Posthooks

Posthook instructions define a restricted high level language that allows to read or to set some bits in specific hardware registers when an interrupt occurs. For each kind of interrupt, a driver can use such posthook instructions, that are synchronously interpreted and executed by the kernel, in order to quickly acknowledge hardware interrupts.

4.4 Debugging EwoK Scheduler

Contents

- *Debugging EwoK Scheduler*
 - *About Ewok schedulers*
 - * *Round-Robin scheduler*
 - * *Random scheduler*
 - * *MLQ-RR scheduler*
 - *Activating the scheduler debug mode*
 - *Exploiting the scheduler debug mode*

4.4.1 About Ewok schedulers

There are several scheduling schemes supported in EwoK:

- Basic Round-Robin
- Random scheduler
- MLQ-RR (Multi-Queue Round-Robin)

All scheduling schemes are constrained by the following:

- ISR have a greater priority and are executed before regular main threads.
- Softirqd, which executes asynchronous syscalls and prepares tasks to handle ISR, is executed with a greater priority than other tasks, but lower priority than ISRs.
- If neither an ISR nor a syscall is to be executed, the global thread scheduling scheme is executed (i.e. Round-Robin, Random or MLQ-RR scheme) on all regular threads.

Round-Robin scheduler

The Round-Robin scheduler schedules each runnable task successively. Each task can use the core up to the configured task slot time slice. The task is scheduled if:

- the task reaches the task slot length
- an interrupt arises, requiring an ISR execution
- the task voluntarily yields
- the task executes an asynchronous syscall (IPC or CFG)

When the scheduler is executed, it selects the next task (starting with the next id, based on the current task id), and elects the first task which is runnable.

If no task at all is runnable, the idle task is executed.

Random scheduler

The random scheduler is using the hardware RNG to select a task in the task list. If the task is not runnable, the scheduler gets back another random number and tries again.

The scheduler tries 32 times before executing the idle task.

This scheduler is mostly an example of basic scheduling scheme.

MLQ-RR scheduler

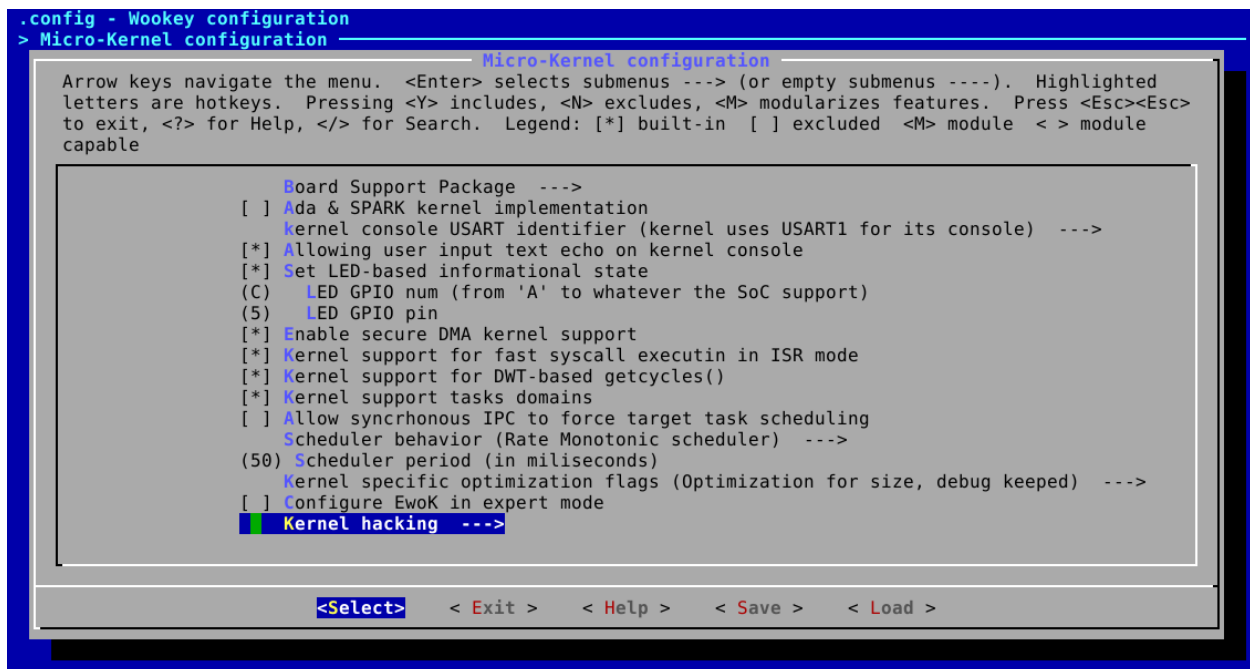
The MLQ-RR scheduler is the only scheduling scheme of EwoK supporting tasks priorities. This scheduler is able to schedule tasks based on their priority using the following rules:

- The scheduler selects the list of runnable tasks having the highest priority
- If more than one task have the same priority, the scheduler uses a round-robin policy on tasks of the same priority

This scheduling is efficient to prioritize tasks with high idle periods and short (but requiring high reactivity) runnable periods. The drawback of such efficiency is that tasks have to voluntarily yield or ask for being idle (for example by locking on IPC receive) to avoid starvation of lower priority tasks.

4.4.2 Activating the scheduler debug mode

The scheduler debug mode can be activated from the kernel menuconfig, in the *'kernel hacking'* menu:



```
.config - Wookey configuration
> Micro-Kernel configuration
Micro-Kernel configuration
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted
letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc>
to exit, <?> for Help, </> for Search. Legend: [*] built-in [ ] excluded <M> module < > module
capable

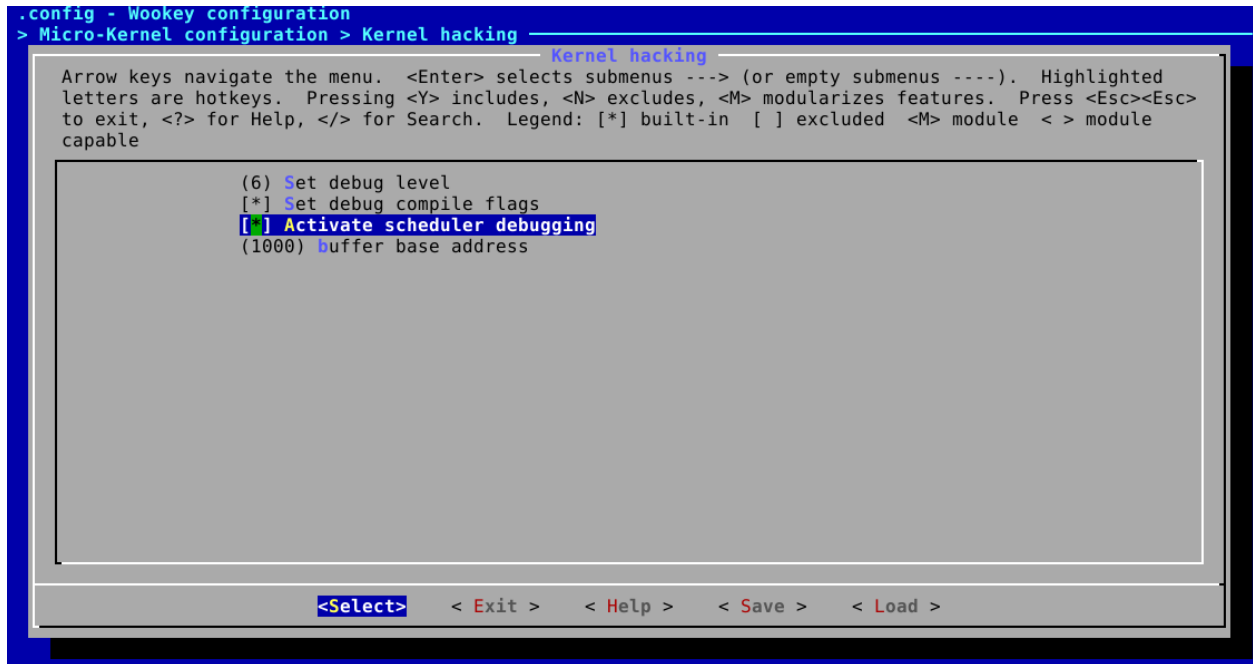
Board Support Package --->
[ ] Ada & SPARK kernel implementation
kernel console USART identifier (kernel uses USART1 for its console) --->
[*] Allowing user input text echo on kernel console
[*] Set LED-based informational state
(C) LED GPIO num (from 'A' to whatever the SoC support)
(5) LED GPIO pin
[*] Enable secure DMA kernel support
[*] Kernel support for fast syscall executin in ISR mode
[*] Kernel support for DWT-based getcycles()
[*] Kernel support tasks domains
[ ] Allow synchronous IPC to force target task scheduling
Scheduler behavior (Rate Monotonic scheduler) --->
(50) Scheduler period (in miliseconds)
Kernel specific optimization flags (Optimization for size, debug kept) --->
[ ] Configure EwoK in expert mode
[ ] Kernel hacking --->

<Select> < Exit > < Help > < Save > < Load >
```

There are two dedicated options in this menu:

- Activate scheduler debugging

- Scheduling buffer size



The option *Activate scheduler debugging* activates the following behaviors:

- **Each task holds three counters:**
 - the number of normal scheduling
 - the number of ISR scheduling
 - the number of forced scheduling after ISR (see `device_t` API)
- the kernel registers the last scheduling information in a ring-buffer

The *Scheduler buffer size* sets the scheduler ring-buffer length. The bigger the buffer is, the bigger the registered temporal window is. Although, the buffer is held in the kernel memory, and hence needs to fit in it. Its default length is 1000 entries.

Hint: Tataouine SDK will help when increasing the buffer size, returning a specific error in the kernel data section if it is too big.

4.4.3 Exploiting the scheduler debug mode

At any time, it is possible to get back each task's counters. You can dump the counters of any task you want. Gdb will help you with variable name completion:

```
(gdb) print tasks_list[2].count
$2 = 0x1bf
(gdb) print tasks_list[2].isr_count
$3 = 0x41
(gdb) print tasks_list[2].force_count
$4 = 0x0
```

It is also possible to get back the scheduling buffer from gdb, to understand how tasks are executed. The scheduling buffer keeps three information:

- the timestamp (in microseconds, since the processor boot time)
- the task id (as referenced in include/generated/app_layout.h)
- the task mode (0x0 is the main thread, 0x1 is an ISR)

Printing the scheduler ring buffer is easy using gdb:

```
arm-none-eabi-gdb
(gdb) target extended-remote localhost:3333
(gdb) monitor reset halt
(gdb) c
... wait for some time
^C
(gdb) symbol-file build/armv7-m/wokey/kernel/kernel.elf
(gdb) set print elements 1000
(gdb) print sched_ring_buffer.buf
```

You can copy the ring buffer content into a text file and clean it to make it easily readable:

```
sed -i -re 's/}, \{\n/g' schedbuf.dat
```

This will delete blocks and generate one scheduling event by line. Such a scheduling trace will then look like this:

```
ts = 0x3ce5c0, id = 0x8, mode = 0x0
ts = 0x3ce648, id = 0x4, mode = 0x1
ts = 0x3ce653, id = 0x4, mode = 0x0
ts = 0x3ce65e, id = 0x8, mode = 0x0
ts = 0x3ce663, id = 0x8, mode = 0x0
ts = 0x3ce6e4, id = 0x4, mode = 0x1
ts = 0x3ce6ef, id = 0x8, mode = 0x0
ts = 0x3ce6f7, id = 0x4, mode = 0x0
ts = 0x3ce703, id = 0x8, mode = 0x0
ts = 0x3ce70c, id = 0x2, mode = 0x0
ts = 0x3da967, id = 0x4, mode = 0x0
ts = 0x3da973, id = 0x8, mode = 0x0
ts = 0x3da97b, id = 0x2, mode = 0x0
```

It is possible to post-process it in various ways, using graphviz, gnuplot or any other tools depending on your need.

5.1 General FAQ

Contents

- *General FAQ*
 - *Why applications main function is named `_main`?*
 - *What is a typical generic task's `main()` function?*
 - *Syscall API is complex: why?*
 - *Why should I define a stack size?*
 - *What is NUMSLOTS and how to know the number of slots an application needs?*

5.1.1 Why applications main function is named `_main`?

EwoK applications entry points have the following prototype:

```
int function(uint32_t task_id):
```

There is an unsigned int argument passed to the main function, giving it the current task identifier.

When using the `main` symbol, the compiler requires one of the following prototypes

```
int main(void);  
int main(int argc, char **argv);
```

As EwoK doesn't generate such a prototype, the `main` symbol cannot be used, explaining why `_main` is used instead. The generated ldscript automatically uses it as the application entry point and the application developer has nothing to do other than to name its main function properly.

5.1.2 What is a typical generic task's `main()` function?

A basic main function should have the following content:

- An initialization phase
- A call to `sys_init(INIT_DONE)` to finish the initialization phase
- A nominal phase

A basic, generic main function looks like the following:

```
int _main(uint32_t task_id)
{
    /* Local variables declaration */
    uint8_t syscall_ret;

    /* Initialization phase */
    printf("starting initialization phase\n");

    /* any sys_init call is made here */

    /* End of initialization sequence */
    sys_init(INIT_DONE);

    /* Nominal sequence */
    printf("starting nominal phase\n");

    /*
     * If any post-init configuration is needed, do it here
     * This is the case if memory-mapped devices need to be configured
     */

    /*
     * Start the main loop or main automaton
     */
    do_main_loop();

    return 0;
}
```

5.1.3 Syscall API is complex: why?

EwoK syscalls is a fully driver-oriented API. Efforts have been made in providing various userspace abstractions to help application developers in using generic devices through a higher level API.

These abstractions are separated in:

- userspace drivers

These drivers supply a higher level, easier API to applications and manage a given device by using the syscall API and configuring the corresponding registers for memory-mapped devices. Drivers API abstract most of the complexity of the hardware devices (such as USARTs, CRYPT, USB, SDIO, etc.)

- userspace libraries

These libraries implement various hardware-independent features, but may depend on a given userspace driver. They supply a functional API for a given service (serial console, AES implementation, etc.), and in case of a dependency with a userspace driver, manage the driver initialization and configuration.

5.1.4 Why should I define a stack size?

This is due to the way EwoK handles the userspace layout. In EwoK userspace mapping, the userspace stack is on the bottom of the user memory map. If the userspace task overflows its own stack, it immediately generates a memory exception error.

This behavior is due to the fact that on MPU-based systems, the page-guard mechanism cannot be used to detect heap/stack smashing, making it harder to detect stack overflow or heap overflow events. Such a layout, pushing the heap on the top addresses and the stack on the bottom helps in detecting such overflows (the heap grows upwards and the stacks grows downwards).

Hint: You can use your compiler to detect the amount of stack needed, as most compilers are able to calculate the effective used stack size based on the compiled code

Danger: Do **not** use recursive code in userspace applications. Embedded systems are not friendly with recursion, as the amount of stack memory is highly reduced

5.1.5 What is NUMSLOTS and how to know the number of slots an application needs?

The NUMSLOTS option of an application specifies the number of memory slots of the flash section dedicated to userspace applications that are required by the application.

In both DFU and FW mode, there are 8 memory slots, as the MPU is able to handle 8 subregions for a given memory region. As a consequence, the total number of slots of the total number of applications of a given mode (DFU or FW) must not exceed 8.

Hint: This is specific to STM32 MPU and may vary on other SoCs MPU

The slot size depends on the selected SoC (as the amount of accessible flash memory may vary) and the mode in which your application is executed (nominal -aka FW- or DFU).

This information can be found in the following file:

kernel/src/arch/soc/<target_soc>/soc-layout.h

The slot size values are the following:

```
#define FW_MAX_USER_SIZE 64*KBYTE
#define DFU_MAX_USER_SIZE 32*KBYTE
```

FW_MAX_USER_SIZE defines the slot size for FW mode and DFU_MAX_USER_SIZE defines the slot size for DFU mode.

Memory slots hold .text, .got, .rodata and .data content of the application. .data section will be copied into RAM in the application memory layout later at boot time.

As a consequence, depending on the size of these sections, the number of required slots may vary. You can use objdump or readelf tools to get back the effective size of your application and calculate the effective number of slots needed:

```
$ arm-none-eabi-objdump -h build/armv7-m/wokey/apps/myapp/myapp.elf
build/armv7-m/wokey/apps/sdio/sdio.fw1.elf:      file format elf32-littlearm
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
 0  .text          00002b68  080a0000  080a0000  00010000  2**2
    CONTENTS, ALLOC, LOAD, READONLY, CODE
 1  .got           00000024  080a2b68  080a2b68  00012b68  2**2
```

(continues on next page)

(continued from previous page)

		CONTENTS, ALLOC, LOAD, DATA			
2	.stacking	00001a90	20008000	20008000	00028000 2**0
		ALLOC			
3	.data	00000010	20009a90	080a2b8c	00019a90 2**2
		CONTENTS, ALLOC, LOAD, DATA			
4	.bss	0000428c	20009aa0	00000000	00009aa0 2**2
		ALLOC			

Here, the application requires $0x2b68 + 0x24 + 0x10 = 0x2b9c$, which means 11.164 bytes. For this task, one slot is enough in both modes.

Hint: The Tataouine SDK helps when a task is too big for its configured number of slots, and specifies which section is problematic. You can let it detect slots overlap if needed

Hint: The Tataouine SDK calculates both flash memory and RAM consumption of each task, which also allows to detect RAM overlap

5.2 Syscalls FAQ

Contents

- *Syscalls FAQ*
 - *What is the header to include to get the syscalls prototypes?*
 - *When I declare a device, I always get SYS_E_DENIED?*
 - *When I configure a device, I always get SYS_E_INVALID?*

5.2.1 What is the header to include to get the syscalls prototypes?

Syscalls are implemented as functions in userspace, in the libstd. The header is `syscalls.h`.

5.2.2 When I declare a device, I always get SYS_E_DENIED?

Denying may be the consequence of various causes:

1. You are not in the initialization phase
2. You don't have the permission to register this type of device (see *EwoK permissions*)
3. If you use EXTI for one or more GPIO, you must have the corresponding permission
4. If you require a forced execution of the main thread for one more more ISR, you must have the corresponding permission
5. You have left a field non-configured with a value that means something not permitted in your case (for example EXTI access request for GPIO)

Hint: It is a good idea to memset to 0 a `device_t` structure before configuring it and requesting a device to the kernel.

5.2.3 When I configure a device, I always get `SYS_E_INVALID`?

Returning invalid may be the consequence of various causes:

1. Your `device_t` structure contains some invalid (unset) field(s). When using the Ada kernel, be sure to memset to 0 the structure before using it, the kernel is very strict with the user entries (for obvious security reasons)
2. You try to map a device that is not in the supported device map
3. You try to map a device with an invalid size
4. You have set more IRQ or more GPIOs than the maximum supported in the `device_t` structure

Hint: It is a good idea to memset to 0 a `device_t` structure before configuring it and requesting a device to the kernel, and highly recommended when using the Ada kernel

5.3 Permissions FAQ

Contents

- *Permissions FAQ*
 - *When using a library or a driver, are specific permissions required?*
 - *How to be sure of the requested permissions a driver needs?*
 - *May drivers require non-resource related permissions?*
 - *Why is there a permission for time measurement?*
 - *Why are there three levels of crypto access permission?*
 - *What is `PERM_RES_TSK_RNG`?*

5.3.1 When using a library or a driver, are specific permissions required?

There is no permission needed to link to a given userspace library or driver, but they may require one or more permission to work properly.

For example, the `libconsole` (managing a userspace serial console) requires the `Devices/Buses` permission in order to use the `libusart` and configure the specified U(S)ART correctly.

5.3.2 How to be sure of the requested permissions a driver needs?

When a driver is manipulating a hardware resource (i.e. a device), the associated permission is declared in the device list json file stored in `layouts/arch/socs/soc-devmap-<projname>.json`.

Each device has a permission field which is a string value that can be compared to the effective permission name as managed by EwoK, and configurable in the configuration tool of the application using the driver.

Hint: When writing a driver, it is usually a good idea to specify the requested permission(s) in a README file in the driver sources root path

When manipulating devices or events that are not a part of the layout file (e.g. external interrupts -EXTI) this should be done using dedicated permissions in the application permission list. Most of the permissions are device oriented and, as is, should not be too hard to detect. If the permission is missing at runtime, the kernel will explicitly indicate that the device registration is not permitted.

Hint: For EXTIs, they are usually a part of a bigger device which is globally refused if the permission is not set

5.3.3 May drivers require non-resource related permissions?

This can happen depending on the driver implementation **and** usage.

A typical example is the *usart* driver. This driver can be used by an application in two modes:

- automatically mapped mode
- manually mapped mode

In automatically mapped mode, there is no specific additional permission needed. In manually mapped mode, the userspace task can voluntarily map/unmap the u(s)art device at will during its nominal phase. This behavior permits to manage a potentially big number of devices in a same application without mapping all of them at the same time.

This capacity (i.e. to map and unmap devices) is associated to a permission (PERM_RES_MEM_DYNAMIC_MAP), that is required if the application has configured the driver in this very mode.

Hint: Such non-device related permissions are most of the time dependent on the driver API usage

5.3.4 Why is there a permission for time measurement?

Is there a real good reason for all the tasks to have the ability to precisely measure the time?

When a task has the ability to precisely measure time periods, it has *de-facto* the power to detect the behavior of other tasks (yield time, scheduling behavior, IPC response time and so on), which paves the way to initiate multiple side and covert channels between tasks.

In EwoK, we have decided:

- To associate time measurement ability with a permission
- To define three levels of time measurement permissions, from milliseconds to cycle count precision level

Warning: Take care to define only the adequate level of time measurement permissions for your tasks. They should not have (for nearly all nominal usage) access to cycle accurate time access

5.3.5 Why are there three levels of crypto access permission?

When there is a cryptographic coprocessor, there are various ways to use it:

- Handling secrets (typically injecting secret keys in the device registers)
- Requesting cryptographic processing in black box mode (sending clear text or cipher text and getting back the (un)ciphered content from the device, without knowing the secrets used)
- handling both these modes

In the WooKey project, secrets handling and cryptographic dataplane are separated in two tasks, requesting, for the secret handling, the `PERM_RES_CRYPTOCFG` permission, and for the cryptographic requests, the `PERM_RES_CRYPTouser` permission. This allows to lock any access to the configuration registers (including the registers holding secret keys) to the task handling cryptographic processing.

If you wish to handle both accesses at the same task, you can use the `PERM_RES_CRYPTOFULL` permission, which allows all the requested actions, mapping all the needed device registers in the task memory layout.

5.3.6 What is `PERM_RES_TSK_RNG`?

EwoK implements a KRNG (Kernel-based Random Number Generator) mechanism. This permits to initialize the SSP (Stack Smashing Protection) seed for each task canaries.

When a task is requiring random data, it has two possibilities:

- implement its own software-based RNG
- ask the kernel for random content

When the hardware device hosts a (T)RNG (the STM32F339 hosts a True Random Number Generator), the kernel is using it and is able to distribute trusted randomness to userspace tasks. Why a permission then? It is globally not a good idea to request too much randomness from a RNG source, as it may generate exhaustion, making the RNG source less effective. To avoid this, only tasks that **really** require randomness should be able to ask from the KRNG source, reducing the attack surface of the KRNG.

Hint: There is no permission needed to initialize the tasks SSP mechanism

5.4 Ewok Security

Contents

- *Ewok Security*
 - *Why flash is mapped RX and not Execute only for both user and kernel?*
 - *Is the W^X principle supported?*
 - *Is there SSP mechanism?*
 - *Is there ASLR?*
 - *Are there any shared libraries?*

5.4.1 Why flash is mapped RX and not Execute only for both user and kernel?

This is a constraint due to `.rodata` (read only data sections).

Since `.rodata` must be readable, executable code and such data have to live together in the same flash area. Using different MPU regions to split them would have required too much MPU regions (and the number of regions is very constrained by the hardware unit).

Another solution would be to copy `.rodata` content into RAM, but this suffers from the same MPU limitations issues, with the additional drawback of reducing the available task volatile memory.

5.4.2 Is the W^X principle supported?

The EwoK kernel enforces the W^X mapping restriction principle, which is a strong defense in depth mitigation against userland exploitable vulnerabilities.

Moreover, the Ada kernel integrates SPARK proofs that verify at that there is no region that can be mapped W and X at the same time.

5.4.3 Is there SSP mechanism?

Yes, the kernel handles KRNG source and generates seeds for each task stack smashing protection mechanism. All functions (starting with the `_main()` one) are protected.

5.4.4 Is there ASLR?

There is no ASLR as the amount of accessible memory is too small to generate enough entropy for userspace task memory mapping randomization. Each task has access to approximately 32KB of memory, which is too few for an effective ASLR mechanism.

5.4.5 Are there any shared libraries?

There is no such mechanism, as shared libraries require shared `.text` memory including memory abstraction that only a real MMU can bring efficiently.

In microcontrollers, there is no memory abstraction, and as a consequence, no shared executable content.

5.5 EwoK build process

Contents

- *EwoK build process*
 - *When I switch between C-based and Ada-based kernels, the compilation is not performed?*

5.5.1 When I switch between C-based and Ada-based kernels, the compilation is not performed?

When changing the compilation mode of the kernel, the $\$(OBS)$ objects files list of the kernel is modified to point to the Ada (or C) object files. As a consequence, the clean target does not do its work properly as its variables has changed. To be sure to rebuild the kernel in the other language, you can either:

- delete the kernel/ dir from the build directory
- execute a make distclean before calling the defconfig
- remove the build directory manually