HCMC University Of Technology, VNU-HCM
Faculty of Computer Science & Engineering

– Embedded System Lab 02 –

ESP32 GPIO and FreeRTOS task

Pham Hoang Anh & **Huynh Hoang Kha**

Goal In this lab, students are expected to be able to:

- Read the input from or write the output to an GPIO pin.
- Create, schedule, and destroy FreeRTOS tasks on the ESP32 development board.

Content

- FreeRTOS task
- ESP32 GPIO

Prerequisite Requirement

- Have basic knowledge of operating systems and microcontroller programming.
- Finished lab 01

Grading policy

- 40% in-class performance
- 60% report submission

1 FreeRTOS task

1.1 A task in FreeRTOS

FreeRTOS allows an unlimited number of tasks to be run as long as hardware and memory can handle it. As a real time operating system, FreeRTOS is able to handle both cyclic and acyclic tasks. In RTOS, a task is defined by a simple C function, taking a `void*` parameter and returning nothing (`void`).

Several functions are available to manage tasks:

```
//task creation
vTaskCreate();

//destruction
vTaskDelete();

//priority management
uxTaskPriorityGet();
vTaskPrioritySet();

//delay/resume
vTaskDelay();
vTaskDelayUntil();
vTaskSuspend();
vTaskResume();
vTaskResumeFromISR();

//start the FreeRTOS scheduler
vTaskStartScheduler();
```

Code 1: FreeRTOS task management functions

More options are available to user, for instance to create a critical sequence or monitor the task for debugging purpose.

1.2 Life cycle of a task

This section will describe more precisely how can a task evolve from the moment it is created to when it is destroyed. In this context, we will consider to be available only one micro controller core, which means only one calculation, or only one task, can be run at a given time. Any given task can be in one of two simple states : “running” or “not running”. As we suppose there is only one core, only one task can be running at a given time; all other tasks are in the “not running task. Figure 1 gives a simplified representation of this life cycle. When a task changes its state from “Not running” to running, it is said “swapped in” or “switched in” whereas it is called “swapped out” or “switched out” when changing to “Not running” state.

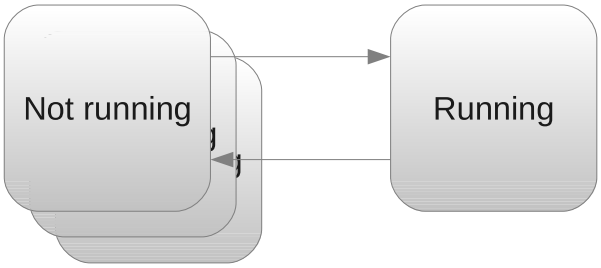

Figure 1: Simplified life cycle of a task : Only one task can be "running" at a given time, whereas the "not running state can be expanded"

As there are several reasons for a task not to be running, the "Not running" state can be expanded as shows Figure 2. A task can be preempted because of a more priority task, because it has been delayed or because it waits for a event. When a task can runs but is waiting for the processor to be available, its state is said "Ready". This can happen when a task has it needs everything to run but there is a more priority task running at this time. When a task is delayed or is waiting for another task (synchronisation through semaphores or mutextes) a task is said to be "Blocked". Finally, a call to `vTaskSuspend()` and `vTaskResume()` or `xTaskResumeFromISR()` makes the task going in and out the state "Suspend".

It is important to underline that a if a task can leave by itself the "Running" state (delay, suspend or wait for an event), only the scheduler can "switch in" again this task. When a task wants to run again, its state turns to "Ready" an only the scheduler can choose which "Ready" task is run at a given time.

1.3 Creating and deleting a task

A task defined by a simple C function, taking one `void*` argument and returning nothing.

```
1 void ATaskFunction( void *pvParameters );
```

Code 2: A typical task signature

Any created task should never end before it is destroyed. It is common for task's code to be wrapped in an infinite loop, or to invoke `vTaskDestroy(NULL)` before it reaches its final brace. As any code in infinite loop can fail and exit this loop, it is safer even for a repetitive task, to invoke `vTaskDelete()` before its final brace.

A task can be created using vTaskCreate(). This function takes as argument the following list:

- `pvTaskCode`: a pointer to the function where the task is implemented.
- `pcName`: given name to the task. This is useless to FreeRTOS but is intented to debugging purpose only.

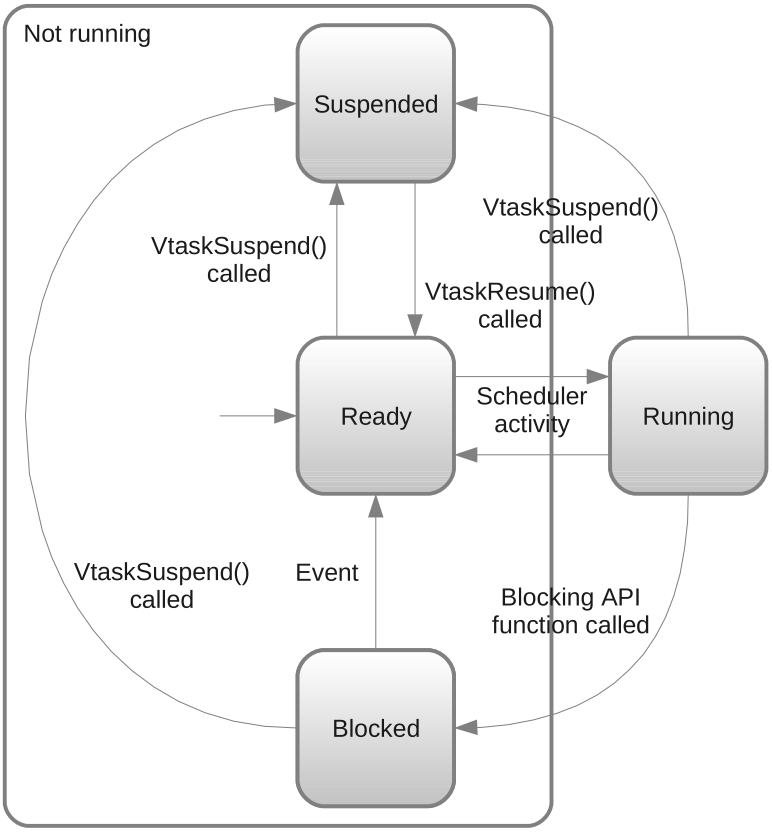

Figure 2: Life cycle of a task

- `usStackDepth`: length of the stack for this task in words. The actual size of the stack depends on the micro controller. If stack with is 32 bits (4 bytes) and usStackDepth is 100, then 400 bytes (4 times 100) will be allocated for the task.

- `pvParameters`: a pointer to arguments given to the task. A good practice consists in creating a dedicated structure, instantiate and fill it then give its pointer to the task.

- `uxPriority`: priority given to the task, a number between 0 and `MAX_PRIORITIES - 1`.

- `pxCreatedTask`: a pointer to an identifier that allows to handle the task. If the task does not have to be handled in the future, this can be leaved `NULL`.

```
portBASE_TYPE xTaskCreate ( pdTASK_CODE pvTaskCode,
      const signed portCHAR * const pcName,
      unsigned portSHORT usStackDepth,
      void *pvParameters,
      unsigned portBASE_TYPE uxPriority,
      xTaskHandle *pxCreatedTask
    );
```

Code 3: Task creation routine

```
void ATaskFunction( void *pvParameters )
{
/* Variables can be declared just as per a normal function.
Each instance of a task created using this function will have
its own copy of the iVariableExample variable. This would not
be true if the variable was declared static - in which case
only one copy of the variable would exist and this copy would
be shared by each created instance of the task. */

 int iVariableExample = 0;

 /* A task will normally be implemented as in infinite loop. */
 for( ;; )
 {
  //The code to implement the task functionality will go here.
 }

 /* Should the task implementation ever break out of the
 above loop then the task must be deleted before reaching
 the end of this function. The NULL parameter passed to
 the vTaskDelete() function indicates that the task to be
 deleted is the calling (this) task. */
 vTaskDelete( NULL );
}
```

Code 4: A typical task

A task is destroyed using `vTaskDelete` routine. It takes as argument `pxCreatedTask` which is given when the task was created. Signature of this routine is given in Code 5 and an example can be found in Code 4.

When a task is deleted, it is responsibility of idle task to free all allocated memory to this task by kernel. Notice that all memory dynamically allocated must be manually freed.

```
void vTaskDelete( xTaskHandle pxTask );
```

Code 5: Deleting a task

1.4 Scheduling

Task scheduling aims to decide which task in "Ready" state has to be run at a given time. FreeRTOS achieves this purpose with priorities given to tasks while they are created. Priority of a task is the only element the scheduler takes into account to decide which task has to be switched in. Every clock tick makes the scheduler to decide which task has to be waken up, as shown in Figure 3 below.

1.4.1 Priorities

FreeRTOS implements tasks priorities to handle multi tasks scheduling. A priority is a number given to a task while it is created or changed manually using `vTaskPriorityGet()` and `vTaskPrioritySet()` (See FreeRTOS manual). There is no automatic management of priorities which mean a task always keeps the same priority

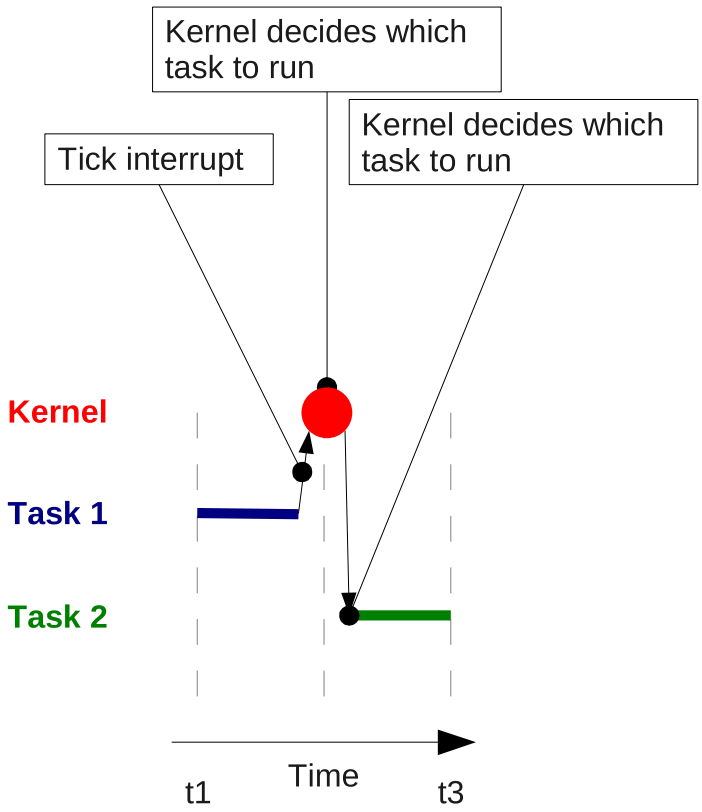

Figure 3: Every clock tick makes the scheduler to run a "Ready" state task and to switch out the running task.

unless the programmer change it explicitly. A low value means a low priority: A priority of 0 is the minimal priority a task could have and this level should be strictly reserved for the idle task. The last available priority in the application (the higher value) is the highest priority available for task. FreeRTOS has no limitation concerning the number of priorities it handles. Maximum number of priorities is defined in `MAX_PRIORITIES` constant in FreeRTOSConfig.h, and hardware limitation (width of the `MAX_PRIORITIES` type). If an higher value is given to a task, then FreeRTOS cuts it to `MAX_PRIORITIES - 1`. Figure 4 gives an example of a application run in FreeRTOS. Task 1 and task 3 are event based tasks (they start when a event occurs, run then wait for the event to occur again), Task 2 is periodic and idle task makes sure there is always a task running.

This task management allows an implementation of Rate Monotonic for task scheduling: tasks with higher frequencies are given an higher priority whereas low frequencies tasks deserve a low priority. Event-based or continuous tasks are preempted by periodic tasks.

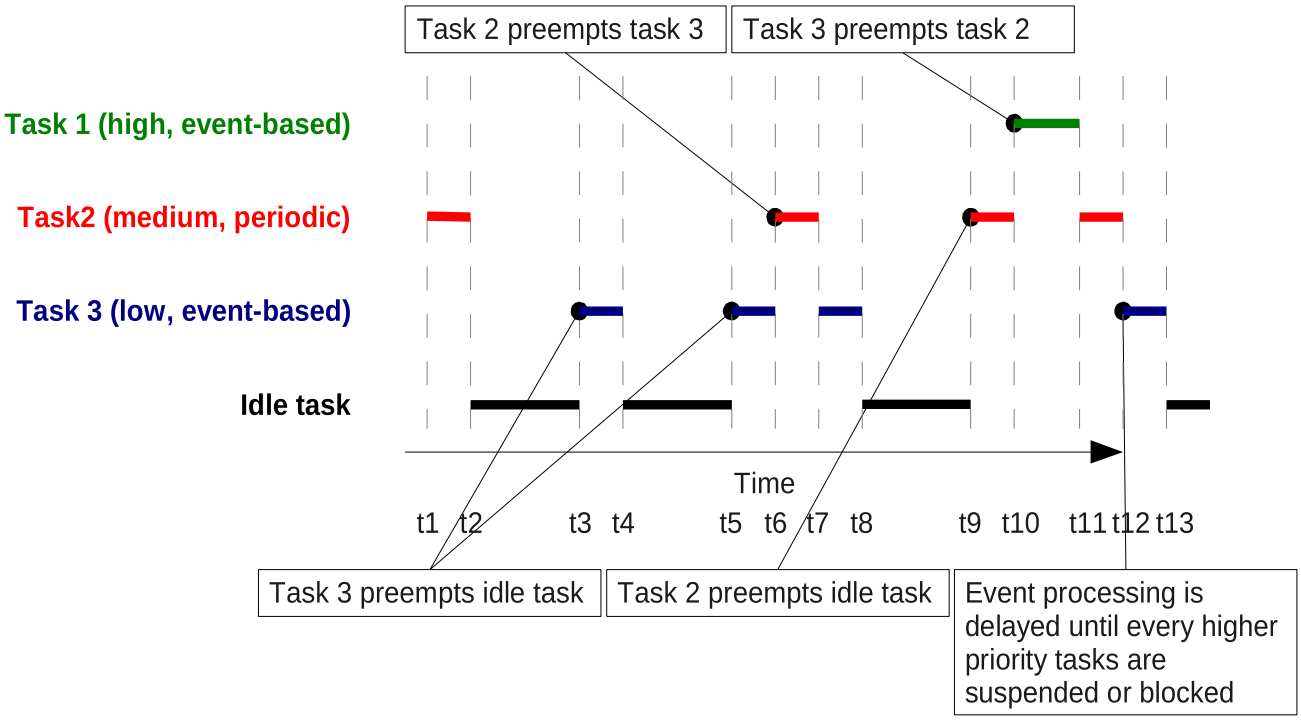

Figure 4: An hypothetic FreeRTOS application schedule

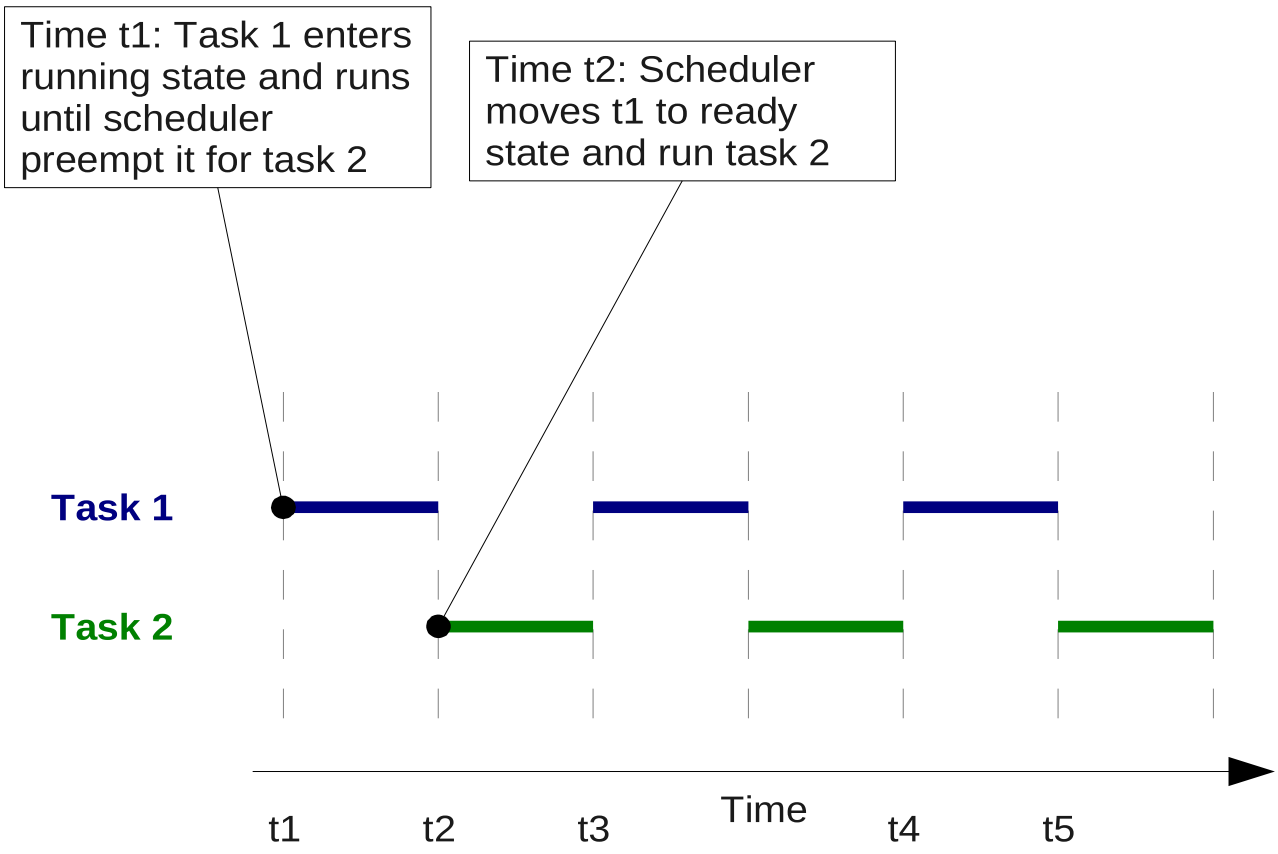

Figure 5: Two tasks with a equivalent priority are run after each other in turn

1.4.2 Priority equally tasks

Tasks created with an equal priority are treated equally by the scheduler: If two of them are ready to run, the scheduler shares running time among all of them: at each clock tick, the scheduler chooses a different task among the ready tasks with highest priority. This implements a Round Robin implementation where quantum is the time between each clock tick. This value is available in `TICK_RATE_HZ` constant, in `FreeRTOSConfig.h`

1.4.3 Starvation

There is no mechanism implemented in FreeRTOS that prevents task starvation: the programmer has to make sure there is no higher priority task taking all running time for itself. It is also a good idea to let the idle task to run, since it can handle some important work such as free memory from deleted tasks, or switching the device into a sleeping mode.

2 ESP32 GPIO

The ESP32 is a 3.3V device. You need to be extremely cautious if you are working with 5V (or above) external MCUs or sensors. Unfortunately devices like the Arduino are typically 5V as are USB UART converters and many sensors. This means you are as likely as not to be working in a mixed voltage environment. Under no circumstances should you think you can power or connect to the ESP32 with a direct voltage of more than 3.3V. It is also important to realize that the maximum amount of current you should anticipate drawing from an output GPIO is only 12mA. To use the GPIO functions supplied by the ESP-IDF, we must include the header file called `driver/gpio.h`.

Next we must call `gpio_pad_select_gpio()` to specify that the function of a given pin should be that of GPIO as opposed to some other function. There are 34 distinct GPIOs available on the ESP32. They are identified as follows, except these ones 20, 24, 28, 29, 30 and 31 are omitted

- GPIO_NUM_0 - GPIO_NUM_19
- GPIO_NUM_21 - GPIO_NUM_23
- GPIO_NUM_25 - GPIO_NUM_27
- GPIO_NUM_32 - GPIO_NUM_39

Note that `GPIO_NUM_34 - GPIO_NUM_39` are input mode only. You can not use these pins for signal output. Also, pins 6 (SD_CLK), 7 (SD_DATA0), 8 (SD_DATA1),

9 (SD_DATA2), 10 (SD_DATA3), 11 (SD_CMD) 16 (CS) and 17(Q) are used to interact with the SPI flash chip... you can not use those for other purposes.

When using pSRAM, strapping pins are GPIO0, GPIO2 and GPIO12; TX and RX (as used for flash) are GPIO1 and GPIO3.

The data type called `gpio_num_t` is a C language enumeration with values corresponding to these names. It is recommended to use these values rather than attempt to use numeric values.

When we want to use a GPIO we need to be aware of whether we are using it as a signal input or a signal output. Think of this as the direction setting. We can set the direction with a call to `gpio_set_direction()`.

```
//Set the GPIO_NUM_17 as OUTPUT
gpio_set_direction(GPIO_NUM_17, GPIO_MODE_OUTPUT);

//Set the GPIO_NUM_17 as INPUT
gpio_set_direction(GPIO_NUM_17, GPIO_MODE_INPUT);
```

Code 6: Setting ESP32 GPIO direction example

If we have set our GPIO as output, we can now set its signal value to be either 1 or 0. We do that by calling `gpio_set_level()`.

```
gpio_pad_select_gpio(GPIO_NUM_17);
gpio_set_direction(GPIO_NUM_17, GPIO_MODE_OUTPUT);
while(1) {
    printf("Off\n");
    gpio_set_level(GPIO_NUM_17, 0);
    vTaskDelay(1000 / portTICK_RATE_MS);
    printf("On\n");
    gpio_set_level(GPIO_NUM_17, 1);
    vTaskDelay(1000 / portTICK_RATE_MS);
}
```

Code 7: An example that toggles a GPIO on and off once a second

As an alternative to setting all the attributes of individual pins, we can set the attributes of one or more pins via a single call using the `gpio_config()` function. This takes a structure called `gpio_config_t` as input and sets the direction, pull up, pull down and interrupt settings of all the pins supplied in a bit mask.

```
gpio_config_t gpioConfig;
gpioConfig.pin_bit_mask = (1 << 16) | (1 << 17);
gpioConfig.mode         = GPIO_MODE_OUTPUT;
gpioConfig.pull_up_en   = GPIO_PULLUP_DISABLE;
gpioConfig.pull_down_en = GPIO_PULLDOWN_ENABLE;
gpioConfig.intr_type    = GPIO_INTR_DISABLE;
gpio_config(&gpioConfig);
```

Code 8: Using GPIO config

In the ESP32 SDK, we can define a GPIO as being pulled-up or pulled-down by using the `gpio_set_pull_mode()` function. This function takes as input the pin number we wish to set and the pull mode associated with that pin.

```
//The GPIO 21 is set to be pulled up
gpio_set_pull_mode(21, GPIO_PULLUP_ONLY);
```

Code 9: Set GPIO pulling mode example

3 Exercises

Given that an ESP-IDF application program has an unique entry point as described in Code 10 below:

```
/**
 * Include required libraries
 * Define some macros
 * Implement some functions
 * ... etc.
**/

//entry function
void app_main(void) {
    //this is the entry point of the program
    //your code goes here
}
```

Code 10: An ESP-IDF application program entry point

Students create 2 tasks and schedule them using FreeRTOS's task management functions:

- A cyclic task printing your student identifier every second.
- An acyclic task polling a button and print "ESP32" every when the button is pressed.

Does the ESP-IDF need the `vTaskStartScheduler()` routine?