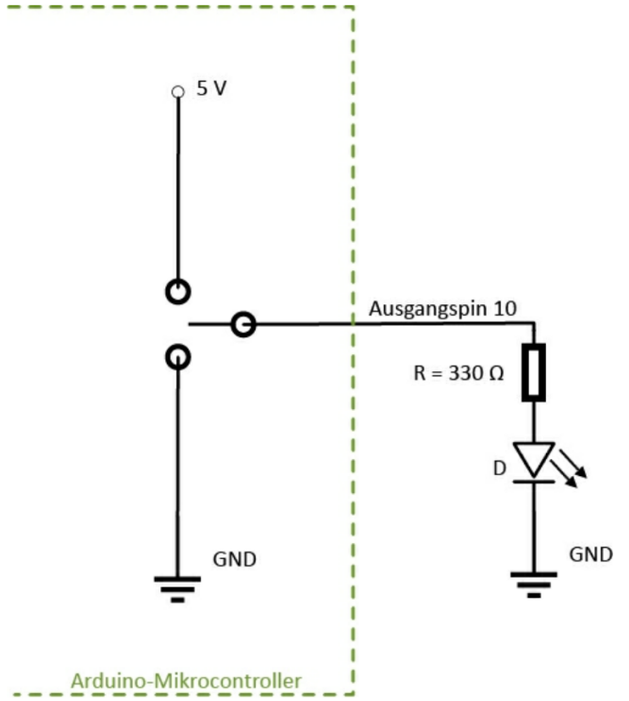
Use Photoelectric Barrier

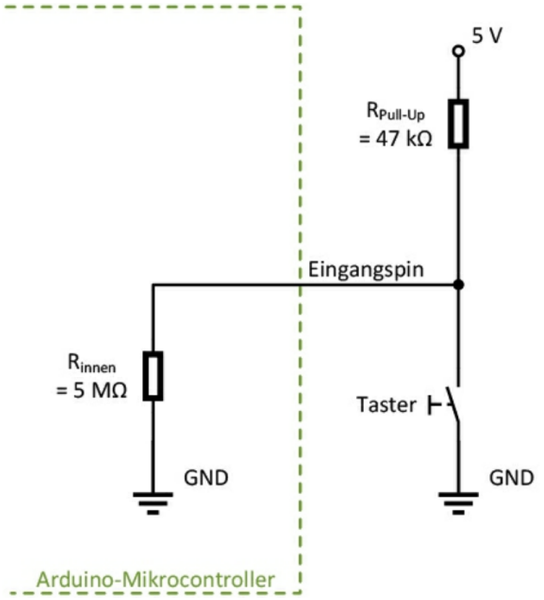
Michael Gerndt
Technische Universität München

I/O Pins

- Digital Outputs
  - Switches pin between LOW and HIGH
  - Be careful of maximum current
  - E.g. trigger an LED



- Digital Inputs
  - Sense LOW and HIGH
  - Be careful that input is always defined.
  - E.g. button

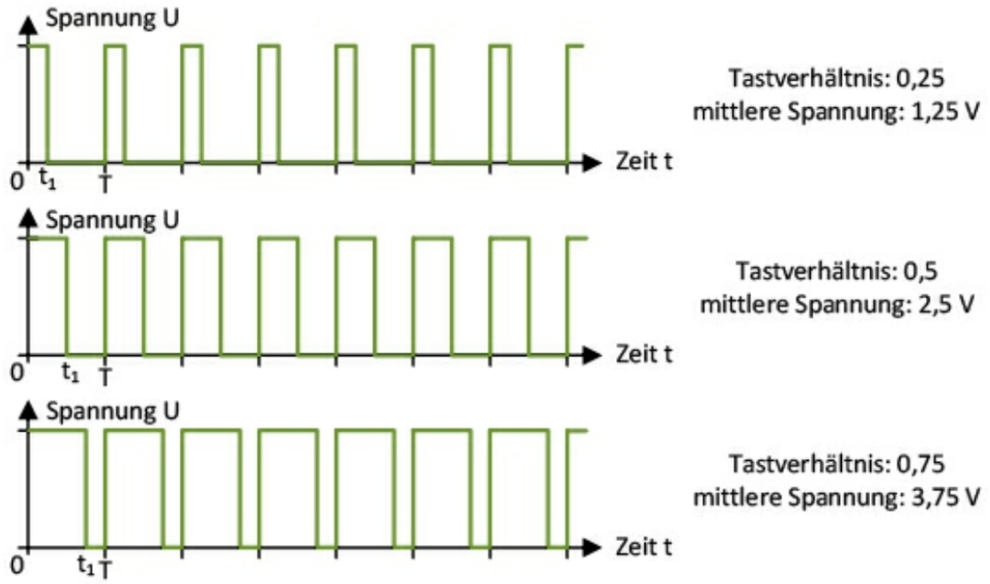


GPIO Handling

```
esp_err_t gpio_set_level(gpio_num_t gpio_num, uint32_t level)
// level is 0 or 1

int gpio_get_level(gpio_num_t gpio_num)

esp_err_t gpio_set_direction(gpio_num_t gpio_num, gpio_mode_t mode)
//GPIO_MODE_INPUT, GPIO_MODE_OUTPUT
//pins 34 - 39 can only be used for input

esp_err_t gpio_pullup_en   (gpio_num_t gpio_num)
esp_err_t gpio_pullup_dis  (gpio_num_t gpio_num)
esp_err_t gpio_pulldown_en (gpio_num_t gpio_num)
esp_err_t gpio_pulldown_dis(gpio_num_t gpio_num)

//pins 34 - 39 do not have software pullup/pulldown resistors
```

I/O Pins

- Analog Digital Converter (ADC)
  - Input is a voltage. 18 channels.
  - E.g. The range of 0..3,3V is represented as 0..4096 for a 12 bit resolution
- Digital Analog Converter (DAC)
  - Converts digital value to analog voltage.
  - Two 8 bit converters connected to Pins 25 and 26.
  - Based on a 3.3 V for maximum
- Pulse Width Modulation (PWM)
  - Mainly for LED control on ESP32
  - Digital output is pulsed according to the digital input (0...255)
  - 16 channels



I/O Pins

- Serial Interface (UART or RS-232)
  - Fixed baud rate (bit/s)
  - 2 lines (input and output)
  - Connects two systems
- I2C - Inter-Integrated Circuit Bus
  - Serial bus
  - 2 lines: data (SDA) and clock (SCL)
  - Connecting master with multiple slaves identified by addresses
- SPI - Serial Peripheral Interface (SPI)
  - Data transmission on separate input and output line
    - SCK: clock
    - MISO: master in slave out
    - MOSI: master out slave in
  - Requires additional signal to select a slave on the bus.

FreeRTOS

- ESP IDF is based on the Free Real Time Operating System (FreeRTOS)
  - freertos.org
  - Maintenance taken over by Amazon from Real Time Engineers ltd.
    - Integration of sensors with the Greengrass edge runtime
  - It is free: No need to publish your code if developed with FreeRTOS, no fees
- Basically a runtime system linked to the application
  - Managing resources: CPU, memory, timers, IO
  - Supports soft and hard realtime requirements

FreeRTOS

- Terminology
  - What is usually called a **thread** is a **task** in FreeRTOS
- Provides
  - Multitasking: time slicing with task preemption
    - The task scheduler is the heart of FreeRTOS
    - Soft and hard realtime constraints are supported through task prioritization
  - Synchronization: locks, semaphores, ...
  - Software timers: countdown clocks triggering an attached callback function
  - HEAP management
- Write embedded software without a kernel
  - For simple applications may be easier: Arduino style (setup and loop routine)
  - In case of multiple tasks to be run with a certain frequency it becomes complex. The frequency has to be implemented in the code of the loop, e.g., display task, read sensors, handle input from Wifi ...

Naming Conventions

- Variables
  - ‘c’ for char, ‘s’ for int16_t (short), ‘l’ int32_t (long), and ‘x’ for BaseType_t and any other non-standard types (structures, task handles, queue handles, etc.).
  - If a variable is unsigned, it is also prefixed with a ‘u’. If a variable is a pointer, it is also prefixed with a ‘p’. For example, a variable of type uint8_t will be prefixed with ‘uc’, and a variable of type pointer to char will be prefixed with ‘pc’.
- Functions
  - Functions are prefixed with both the type they return, and the file they are defined within.
  - vTaskPrioritySet() returns a void and is defined within task.c. 'v' for void.
  - xQueueReceive() returns a variable of type BaseType_t and is defined within queue.c.
  - pvTimerGetTimerID() returns a pointer to void and is defined within timers.c.

Naming Conventions

- Macros
  - port (for example, portMAX_DELAY): portable.h or portmacro.h
  - task (for example, taskENTER_CRITICAL()): task.h
  - pd (for example, pdTRUE): projdefs.h
  - config (for example, configUSE_PREEMPTION): FreeRTOSConfig.h
  - err (for example, errQUEUE_FULL): projdefs.h

Task States

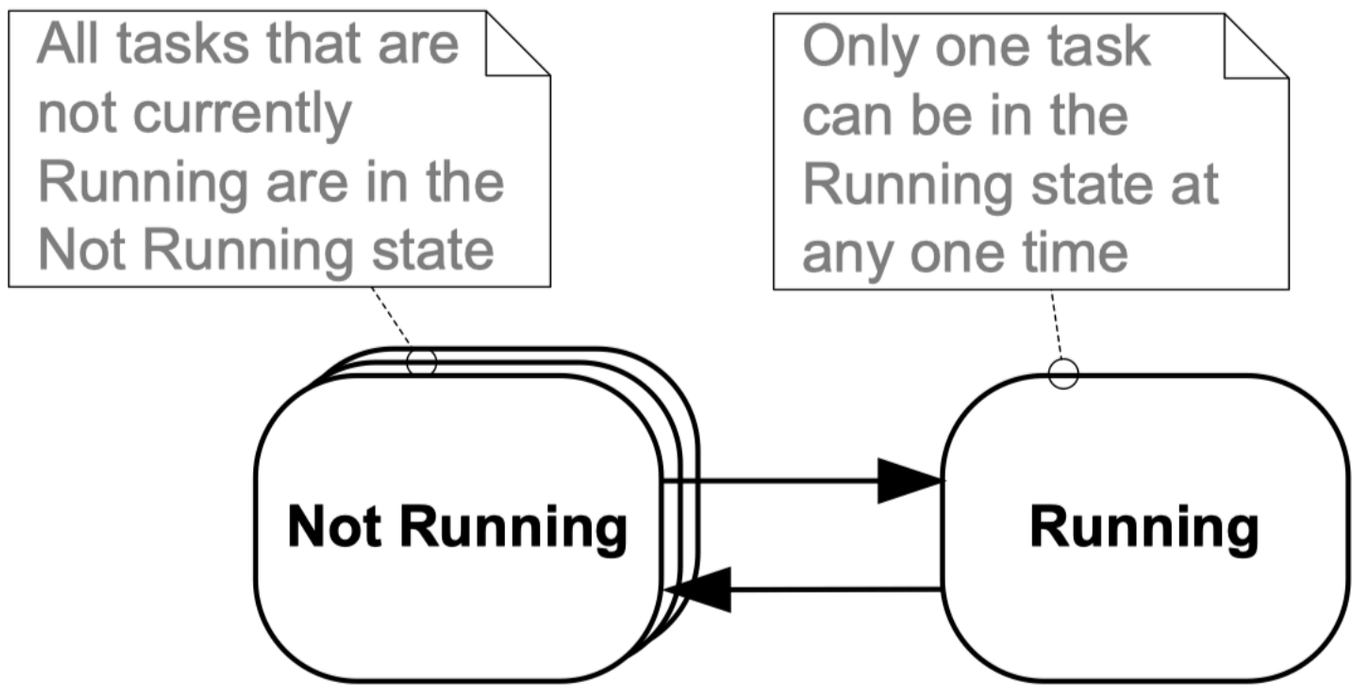


Figure 9. Top level task states and transitions

Declare a Task Function

```
void vTaskFunction( void *pvParameters )
{
char *pcTaskName;
volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */

    /* The string to print out is passed in via the parameter.  Cast this to a
    character pointer. */
    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation.  There is
            nothing to do in here.  Later exercises will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

Listing 18. The single task function used to create two tasks in Example 2

Start a Task in FreeRTOS

```
/* Define the strings that will be passed in as the task parameters.  These are
defined const and not on the stack to ensure they remain valid when the tasks are
executing. */
static const char *pcTextForTask1 = "Task 1 is running\r\n";
static const char *pcTextForTask2 = "Task 2 is running\r\n";

int main( void )
{
    /* Create one of the two tasks. */
    xTaskCreate(    vTaskFunction,          /* Pointer to the function that
                                               implements the task. */
                    "Task 1",               /* Text name for the task.  This is to
                                               facilitate debugging only. */
                    1000,                   /* Stack depth - small microcontrollers
                                               will use much less stack than this. */
                    (void*)pcTextForTask1,  /* Pass the text to be printed into the
                                               task using the task parameter. */
                    1,                      /* This task will run at priority 1. */
                    NULL );                 /* The task handle is not used in this
                                               example. */

/* Create the other task in exactly the same way.  Note this time that multiple
tasks are being created from the SAME task implementation (vTaskFunction).  Only
the value passed in the parameter is different.  Two instances of the same
task are being created. */
xTaskCreate( vTaskFunction, "Task 2", 1000, (void*)pcTextForTask2, 1, NULL );
```

Task Scheduling

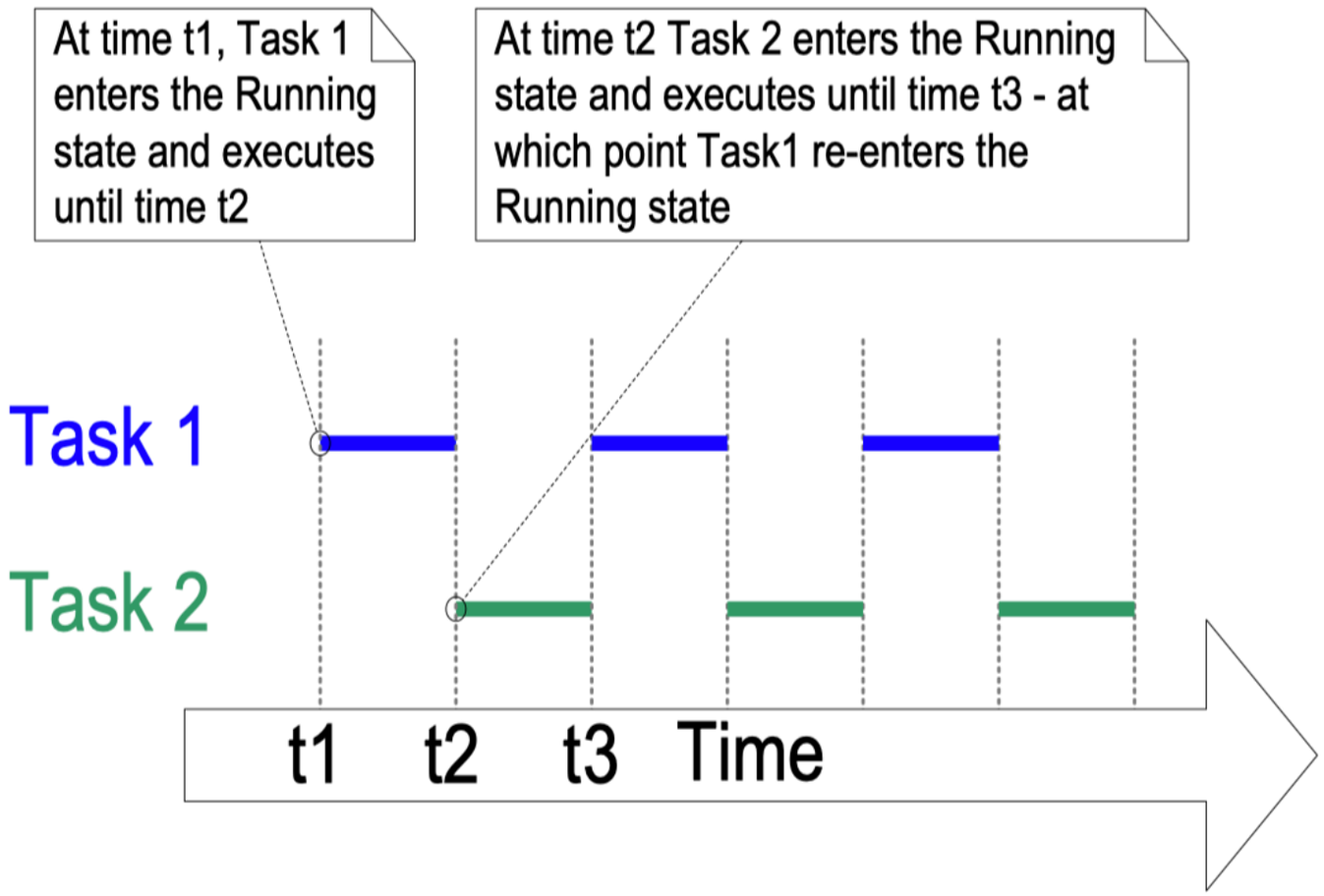


Figure 11. The actual execution pattern of the two Example 1 tasks

Time Slicing

- The interrupt frequency is defined in FreeRTOSConfig.h via `configTICK_RATE_HZ`.
  - It is set to `CONFIG_FREERTOS_HZ` and this is defined in sdkconfig as 100Hz.
- Time slicing is an optional feature of FreeRTOS.
- Other scheduling points:
  - Wait for a certain time: **vTaskDelay(1000 / portTICK_PERIOD_MS);**
  - Wait for an event: xQueueReceive(...)
  - Wait for a synchronization: xSemaphoreTake(...)
  - Tasks suspends itself: taskYIELD()
  - A higher priority task becomes ready after being blocked.

Start a Task in FreeRTOS

```
29
30  static void main_task(void* arg){
31    while(1){
32
33      vTaskDelay(2000 / portTICK_RATE_MS);
34    }
35  }
36
37  xTaskCreate(main_task, "main_task", 1024, NULL, 10, NULL);
38
```

Prioritized Pre-emptive Scheduling with Time Slicing

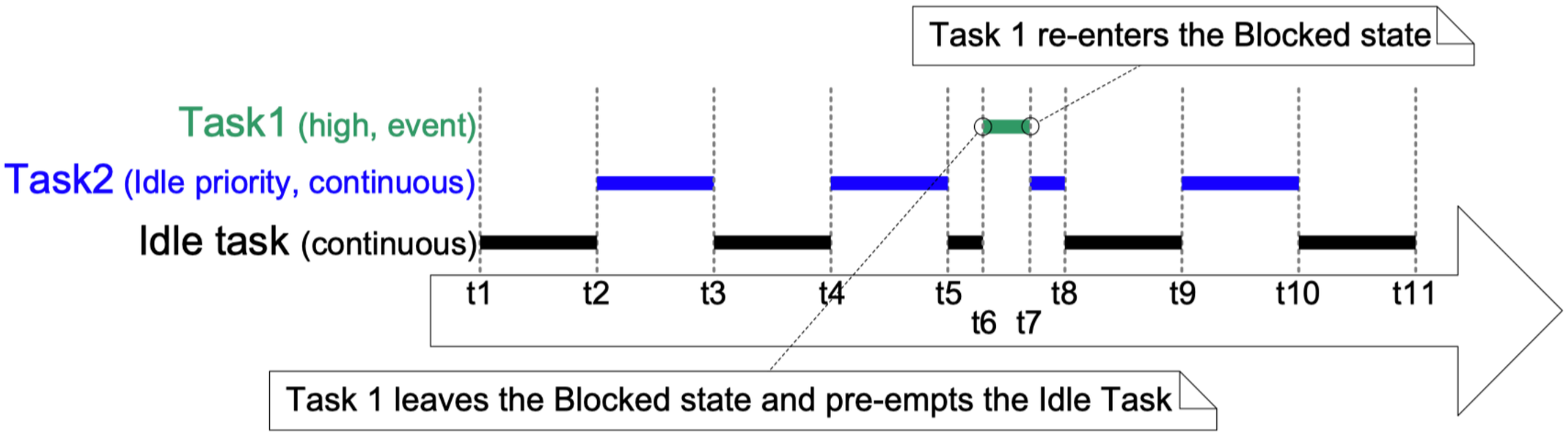


Figure 27 Execution pattern highlighting task prioritization and time slicing in a hypothetical application in which two tasks run at the same priority

Full Task State Diagram

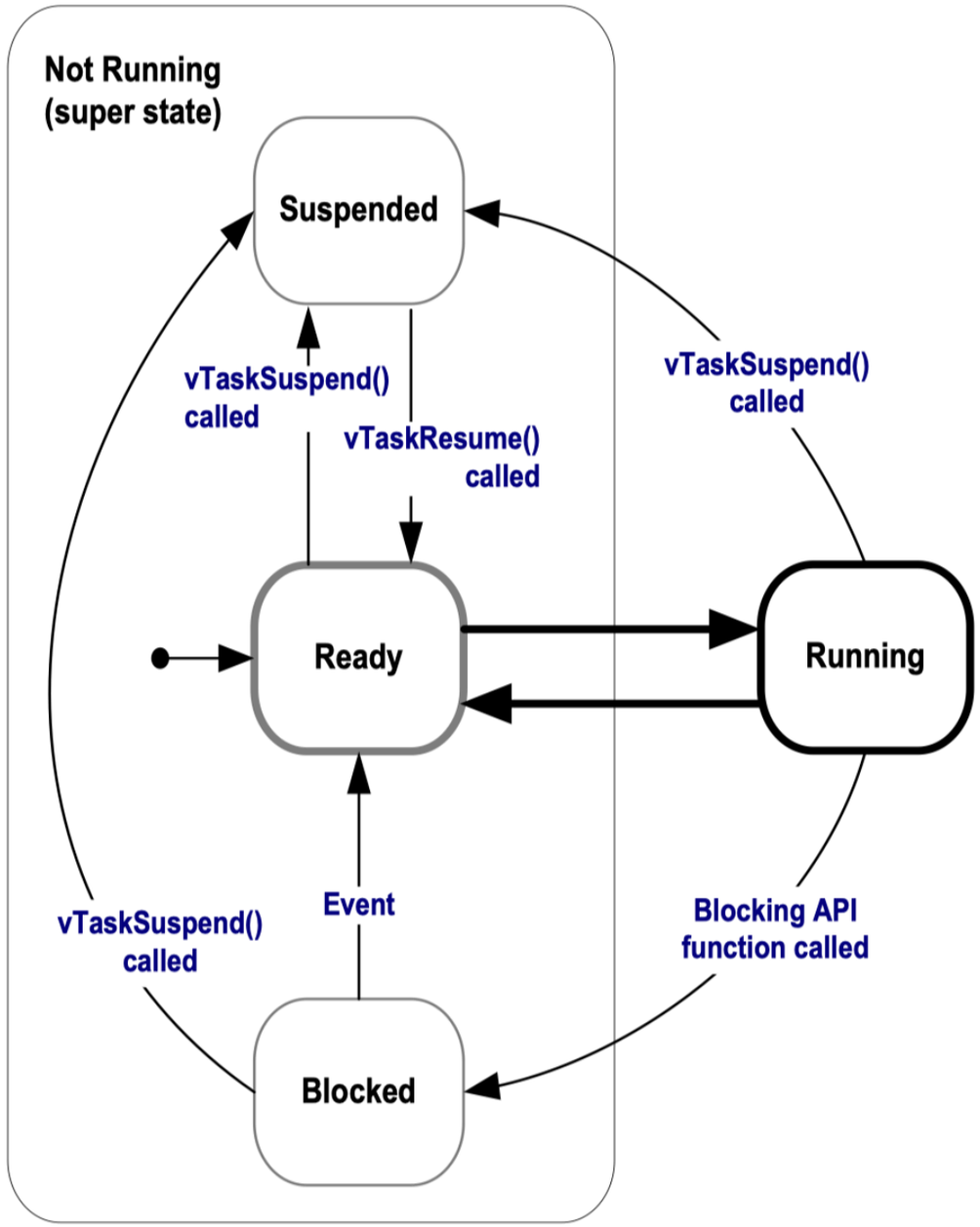


Figure 15. Full task state machine

Stack and Heap

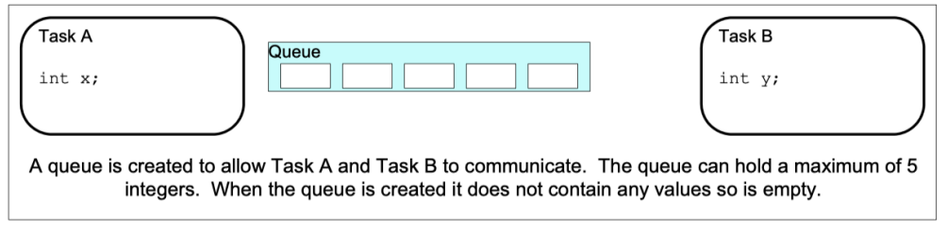
- Stack: dynamic memory allocated by program control flow
- Heap: dynamic memory allocated by function calls

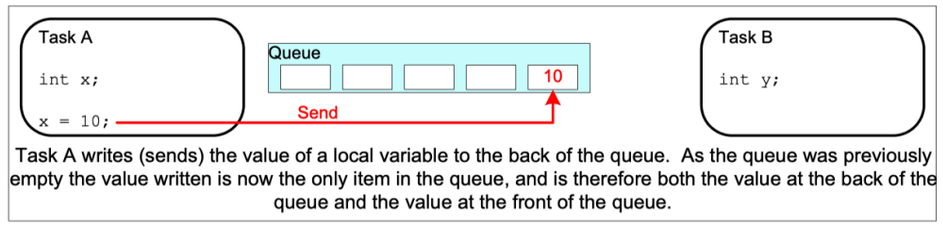
- Each task has it own stack. The stack is allocated on the heap when the task is started. Its size is determined in `xTaskCreate( ... )`.

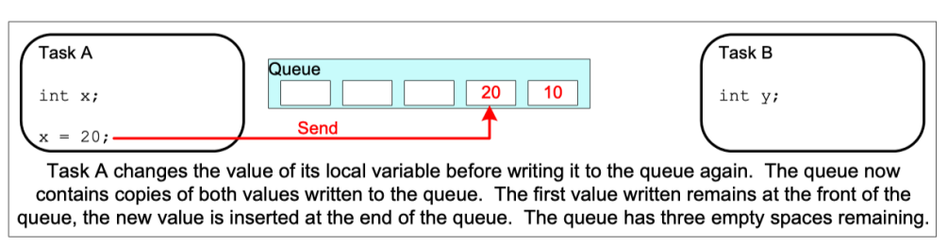
- Common problem is a stack overflow.
  - Programmer might not know how much stack is required. Check the free stack space via int32_t freestack=uxTaskGetStackHighWaterMark(NULL);
  - Automatic detection of stack overflow by one of two methods:
    - Checking the stack pointer
    - Filling the last 20 bytes of the stack with a bit pattern. Whenever this pattern changes, a stack overflow is signaled.

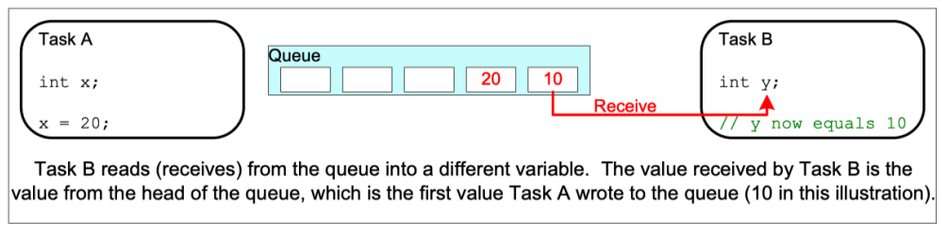
FreeRTOS queues

- Queues are used to communicate between tasks.
- maximum queue elements and size of an element is specified at creation.
- Data are copied into the queue.
- Important: it is threadsafe.









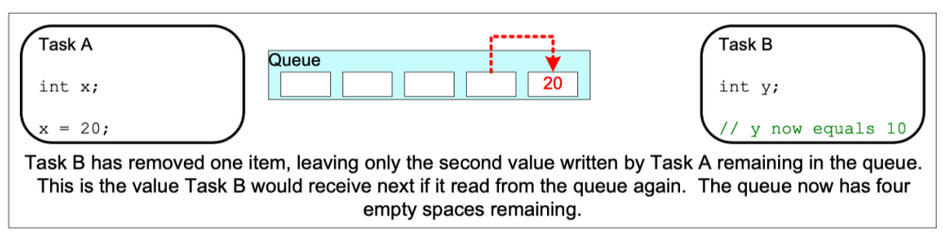


Figure 31. An example sequence of writes to, and reads from a queue

FreeRTOS queues

- `QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength, UBaseType_t uxItemSize );`
  - Creates a queue and returns a handle.
- `BaseType_t xQueueSendToBack( QueueHandle_t xQueue, const void * pvItemToQueue, TickType_t xTicksToWait );`
  - Inserts an element at the end.
  - Waits at most the number of ticks if the queue is full.
- `BaseType_t xQueueReceive( QueueHandle_t xQueue, void * const pvBuffer, TickType_t xTicksToWait );`
  - Will receive the first element into the specified buffer.
  - Waits at most the number of ticks if the queue is empty.
  - return values: pdPASS and errQUEUE_EMPTY

Error Handling

ESP_ERROR_CHECK(x)

Macro which can be used to check the error code, and terminate the program in case the code is not ESP_OK. Prints the error code, error location, and the failed statement to serial output.

Disabled if assertions are disabled.

ESP_ERROR_CHECK_WITHOUT_ABORT(x)

In comparison with ESP_ERROR_CHECK(), this prints the same error message but isn't terminating the program.

Debugging

- Debugging the ESP32
  - printf(...)
  - ESP_LOGx(...)
  - gdb via ESP PROG

ESP Logging

- ESP IDF provides logging macros
  - `ESP_LOGI(TAG,"Free stack space: %d",freestack);`
  - `static const char* TAG = "MyModule"`
  - Verbosity levels: error (lowest), warning, info, debug verbose (highest)
  - `#include "esp_log.h"`
- Logging can be controlled at compile time and run time.
  - Compile time
    - Default log verbosity set via menuconfig (component config/log output)
    - Can be overwritten by `#define LOG_LOCAL_LEVEL ESP_LOG_VERBOSE`
  - Run time
    - `esp_log_level_set("*", ESP_LOG_ERROR)`
    - `esp_log_level_set("MyModule", ESP_LOG_WARN)`
- Logging to Host via JTAG

Counting People in Seminar Room

- Use two photoelectric barriers to detect incoming and outgoing people.
- Two barriers at the doorframe: b_out and b_in.
  - A person entering first breaks b_out and then b_in.
  - Leaving person first breaks b_in and then b_out.
- KY-010 sensor

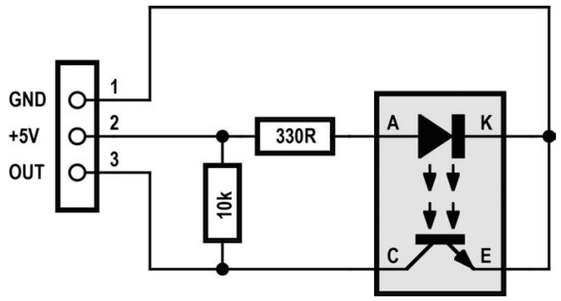
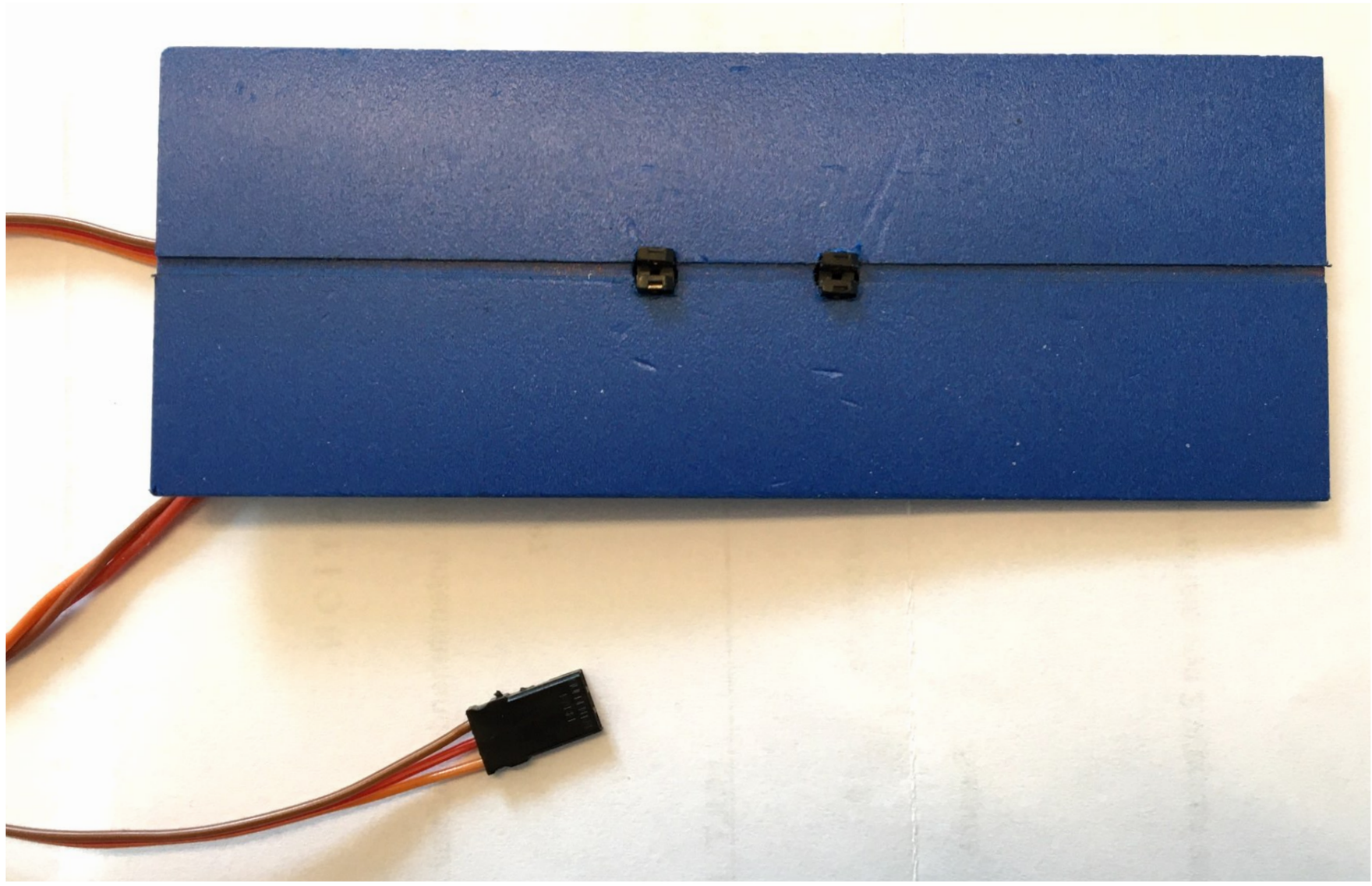


Abbildung 3-63: KY-010-Schaltplan

- LOW voltage at OUT when not activated and HIGH when activated (broken).

Assembled Barriers



Brown: GND
Red: VCC
Orange: Signal

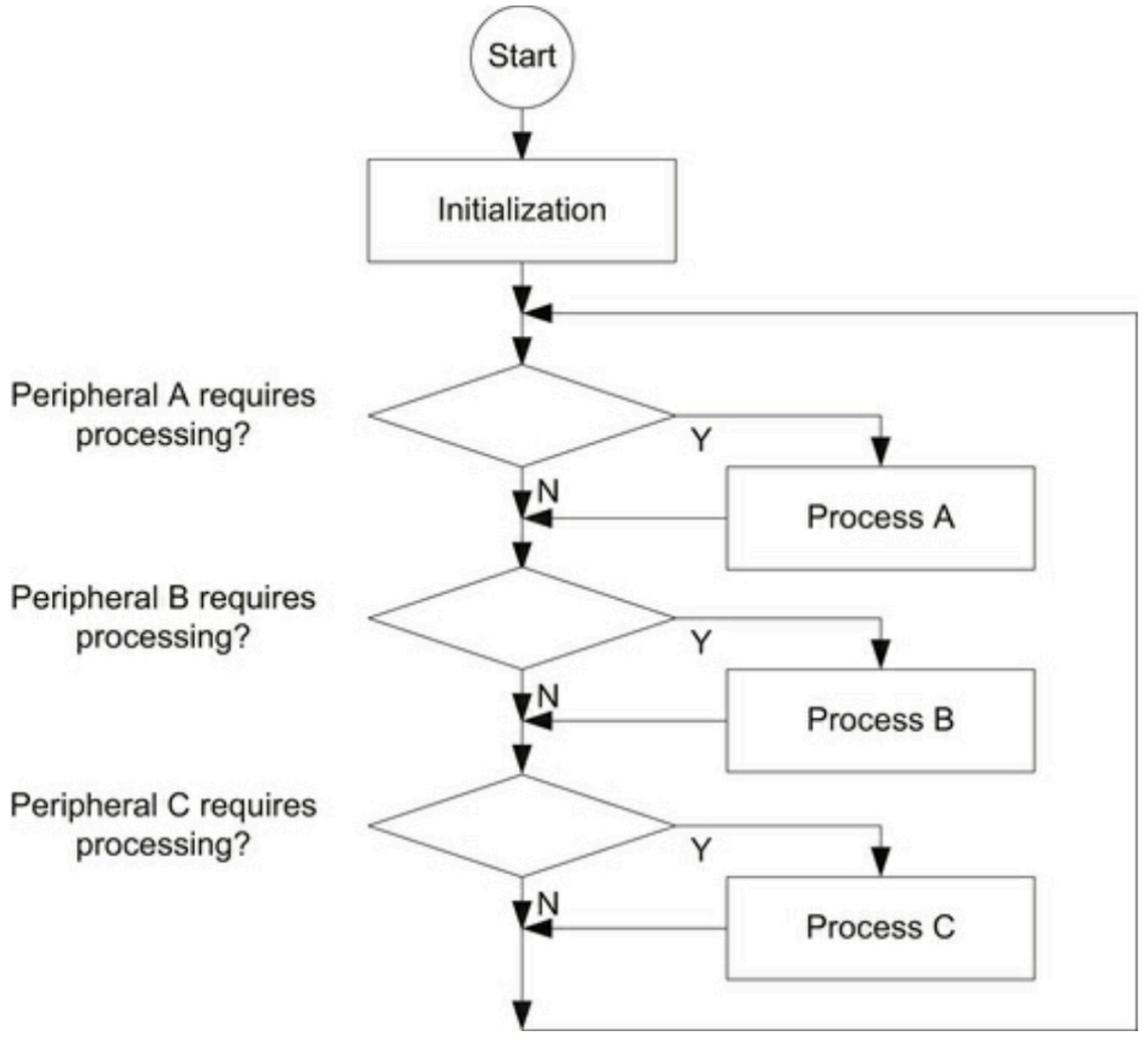
Simulation via Two Buttons

Challenges

- Activating the barrier might create spurious signals (bouncing effect).
- Be careful about the speed of people entering the room.
- Be careful about the gap between different people.
- Think about these challenges and report which challenges your solution tolerates.

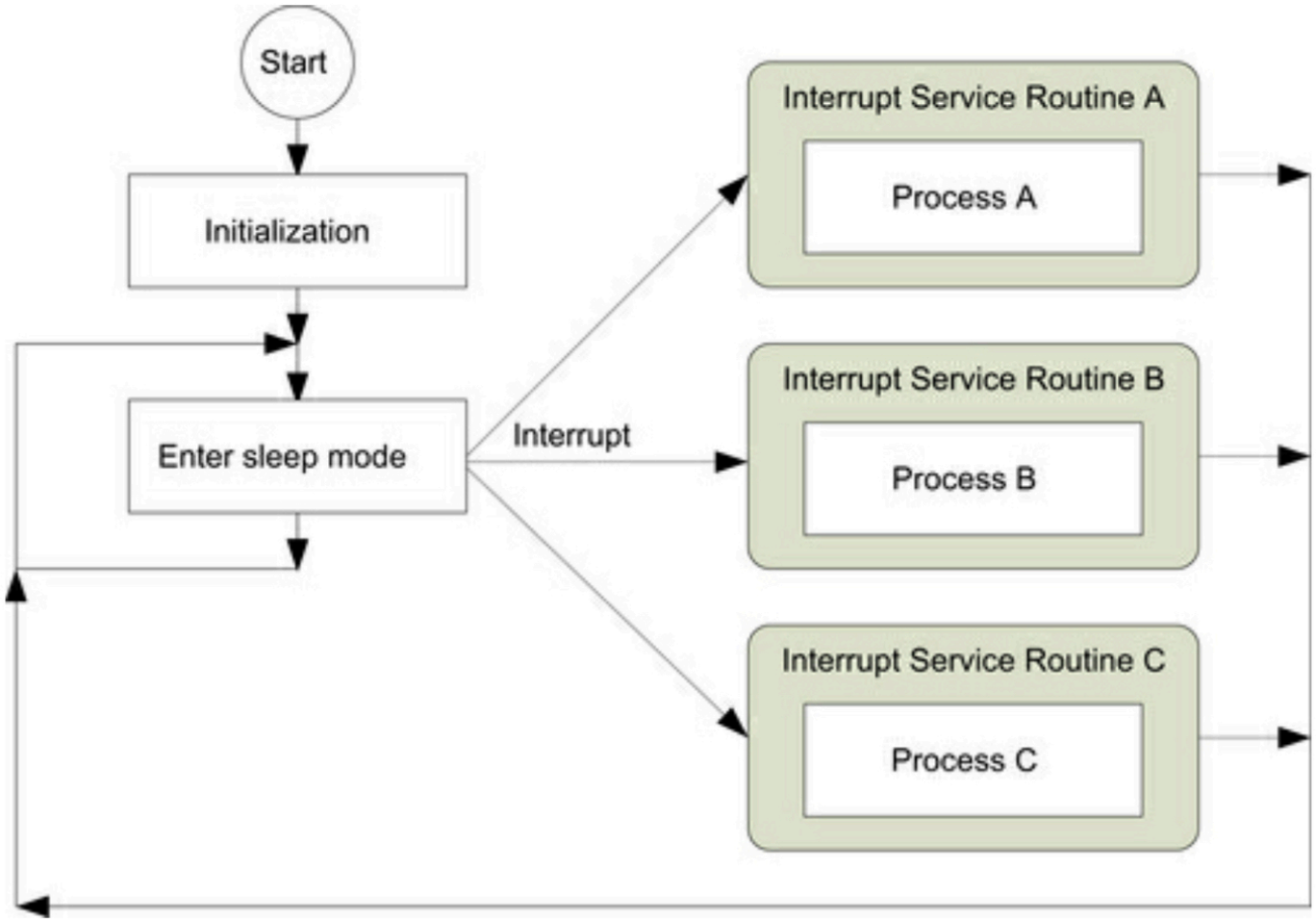
Software Design

- Polling



Software Design

- Interrupt



Interrupt Handling

```
esp_err_t gpio_install_isr_service(int intr_alloc_flags)

//ESP_INTR_FLAG_IRAM: interrupt handler needs to be installed
 with IRAM_ATTR in the instruction RAM
```

Interrupt Handling

```
esp_err_t gpio_set_intr_type(gpio_num_t gpio_num,
                             gpio_int_type_t intr_type)

//GPIO_INTR_POSEDGE, ... _NEGEDGE, ..._ANYEDGE,
//..._LOW_LEVEL, ..._HIGH_LEVEL
```

```
esp_err_t gpio_isr_handler_add(gpio_num_t gpio_num, gpio_isr_t
 isr_handler, void *args)
```

```
void IRAM_ATTR outISR(void* arg){
   ets_printf("Interrupt OUT.\n");
   ...
}
```

Debouncing

- Debouncing
  - https://www.digikey.de/en/articles/how-to-implement-hardware-debounce-for-switches-and-relays

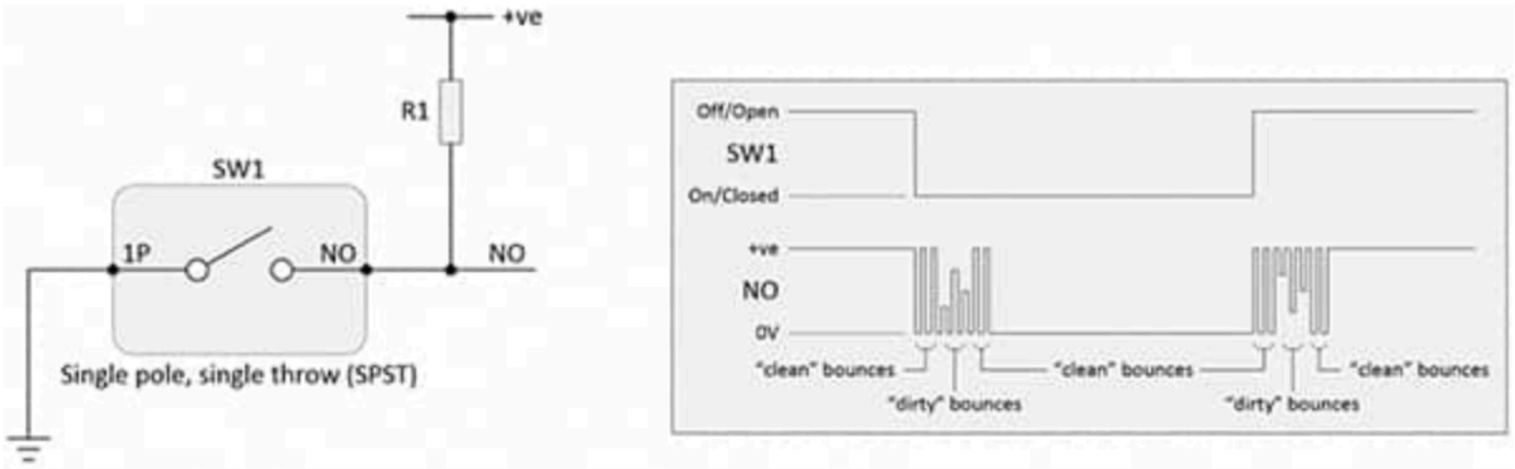


Figure 1: In the case of an SPST-NO toggle switch, bouncing may occur both when the switch is activated and deactivated. (Image source: Max Maxfield)

-

Hardware Debouncing

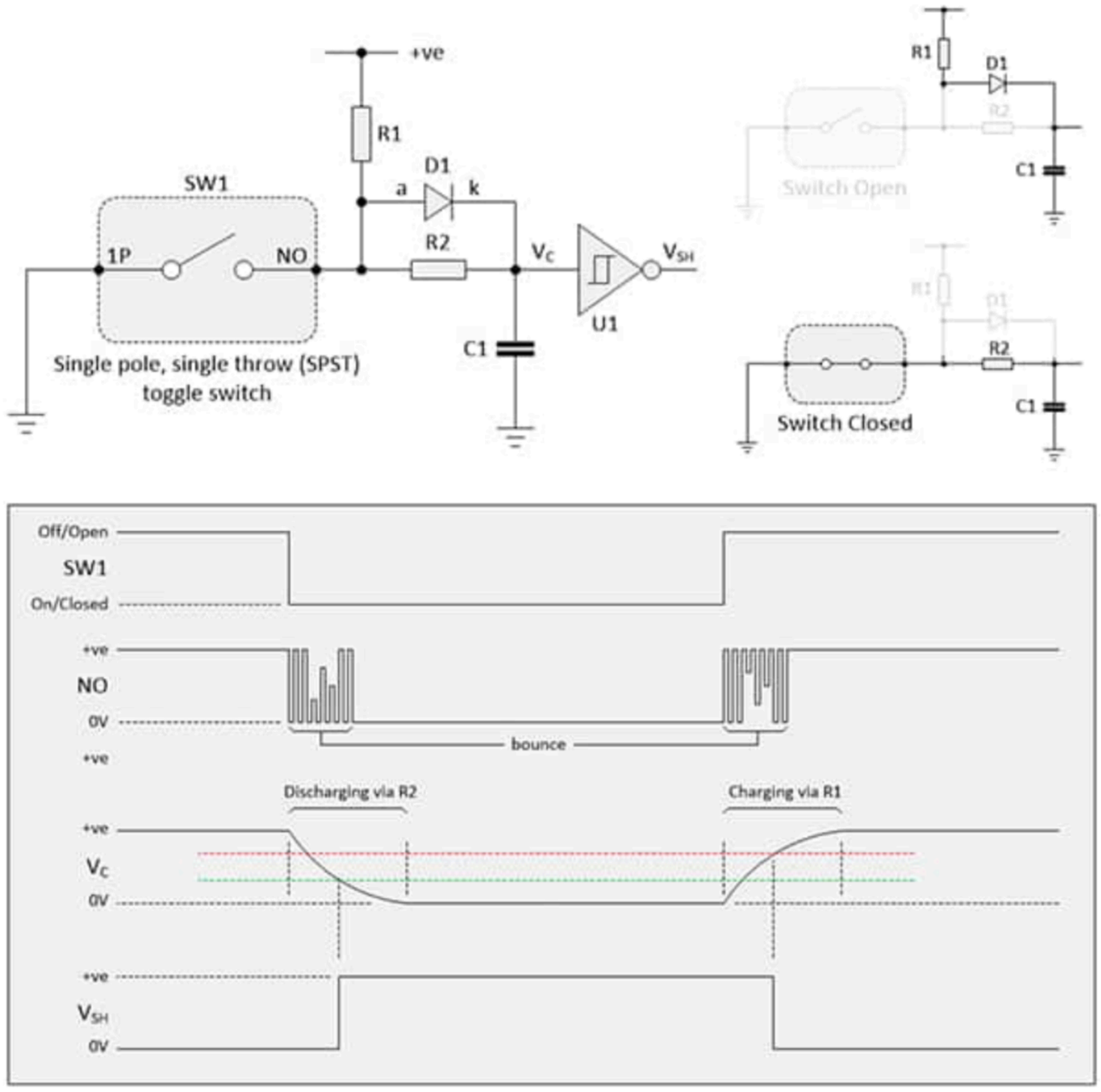


Figure 3: When using an RC network to debounce an SPST switch (top), the addition of the diode (D1) forces the capacitor (C1) to charge via resistor R1 and discharge via resistor R2. (Image source: Max Maxfield)

Software Debouncing

- Software Debouncing
  - E.g. wait until the signal is stable for some time

Tips

- Print inside of interrupt handler
  - ets_printf("Interrupt OUT\n");
- Time
  - #include esp_timer.h
  - esp_timer_get_time(): time in usec since boot

OLED Display

- 0,96 Zoll I2C OLED Display 128x64 pixel
- Pins:
  - GND -> GND
  - VCC -> 3.3 V
  - SCL->SCL
  - SDA->SDA
- Address
  - 0x3C

OLED Display

```
#include "ssd1306.h"

void initDisplay(){
  ssd1306_128x64_i2c_init();
  ssd1306_setFixedFont(ssd1306xled_font6x8);
}

void textDemo(){
  ssd1306_clearScreen();
  ssd1306_printFixedN(0,  0, "Normal text", STYLE_NORMAL,1);
  ssd1306_printFixed(0, 16, "Bold text", STYLE_BOLD);
  ssd1306_printFixed(0, 24, "Italic text", STYLE_ITALIC);
  ssd1306_negativeMode();
  ssd1306_printFixed(0, 32, "Inverted bold", STYLE_BOLD);
  ssd1306_positiveMode();
  delay(3000);
  ssd1306_clearScreen();
}

void app_main(void){
  ...
  initDisplay();
  textDemo();
  ...
}
```

Develop Interrupt based version

1. Install the OLED display (I2C pins 21 and 22).
2. Design your interrupt handling to count people. Use a variable `volatile uint8_t count` for the current number of people in the room.
3. Write a function `showRoomState()` that prints the count on the display.
4. Call `showRoomState()`from an own task. Do not write to the display in another task.
5. Carefully check corner cases, e.g., one peaks into the room but does not enter.