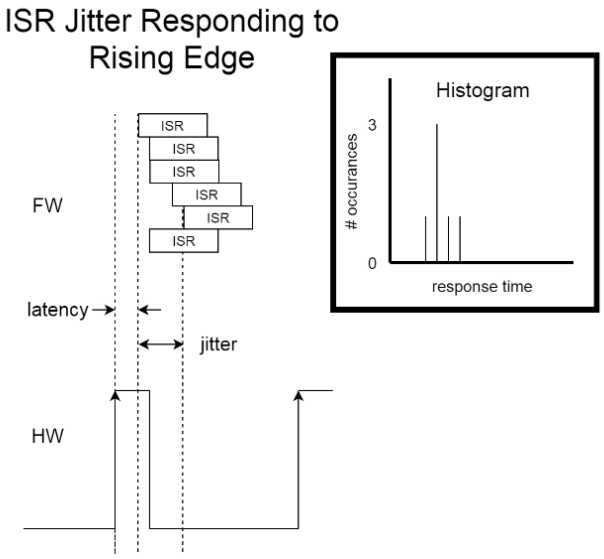
RTOS and Embedded systems

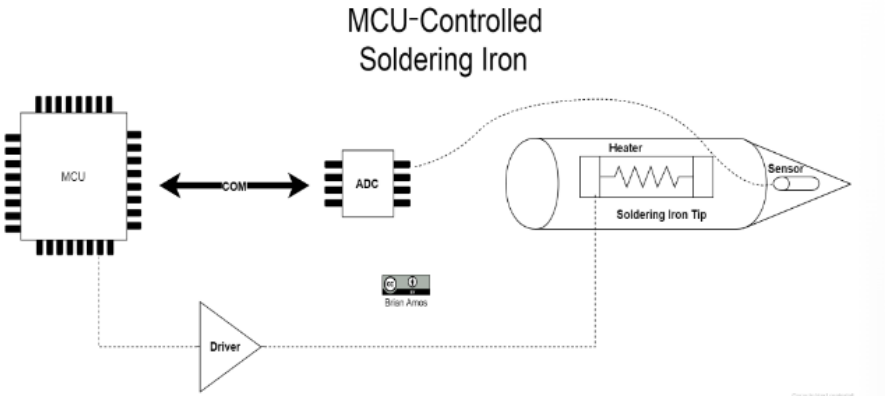
Core idea: Firmware that runs a scheduling kernel on an MCU is RTOS-based firmware. The introduction of the scheduler and some RTOS-primitives allows tasks to operate under the illusion they have the processor to themselves (discussed in detail in Chapter 2, Understanding RTOS Tasks)

There are different ways to minimize latency and jitter for critical ISRs. In ARM Cortex-M-based MCUs, interrupt priorities are flexible – a single interrupt source can be assigned different priorities at runtime:

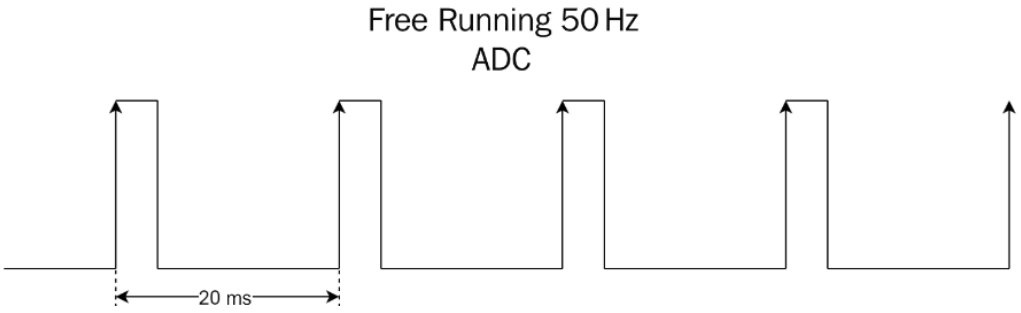


The analyzer or oscilloscope that is going to be reading an ADC at a rate of tens of GHz! The raw ADC readings will likely be converted into the frequency domain and graphically displayed on a high-resolution front panel dozens of times a second. A system like this requires huge amounts of processing to be performed and must adhere to extremely tight timing requirements, if it is to function properly.Somewhere in the middle of the spectrum, you'll find systems such as closed-loop motion controllers, which will typically need to execute their PID control loops between hundreds of Hz to tens of kHz in order to provide stability in a fast-moving system.

MCU

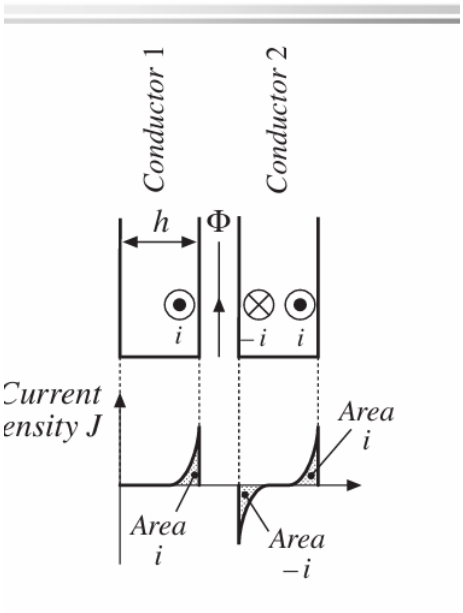


The MCU also needs to be running the control algorithm to calculate the updated values for the heater output at 5 Hz (200 ms). Both of these cases (although not particularly fast) are examples of real-time requirements:



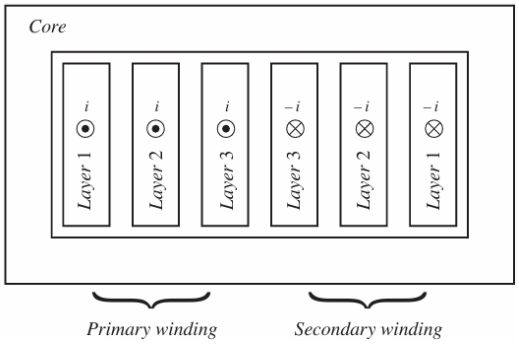
Wiring and magnetic fields examples in cobblestone and how to calculate visually some important measures in an aerospace electrical system:

A multi-layer foil winding, with h > . Each layer carries net current i(t).

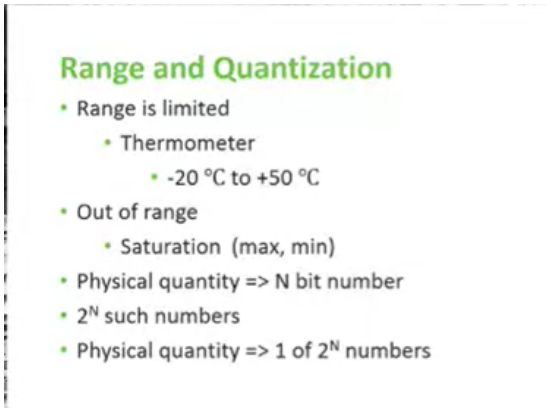


Notes: Arduino examples, magnetic fields examples, cooper loos, Hands-On RTOS with Microcontrollers.

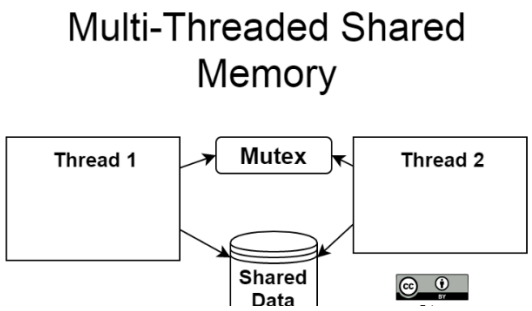
Cross-sectional view of two-winding transformer example. Primary turns are wound in three layers. For this example, let's assume that each layer is one turn of a flat foil conductor. The secondary is a similar three-layer winding. Each layer carries net current i(t). Portions of the windings that lie outside of the core window are not illustrated. Each layer has thickness h > .



In other systems, such as the flight control of a UAV or motion control in industrial process control, failing to run the control algorithm in a timely manner could result in something more physically catastrophic, such as a crash. In this case, the consequences are potentially life-threatening



RTOS guide with C++



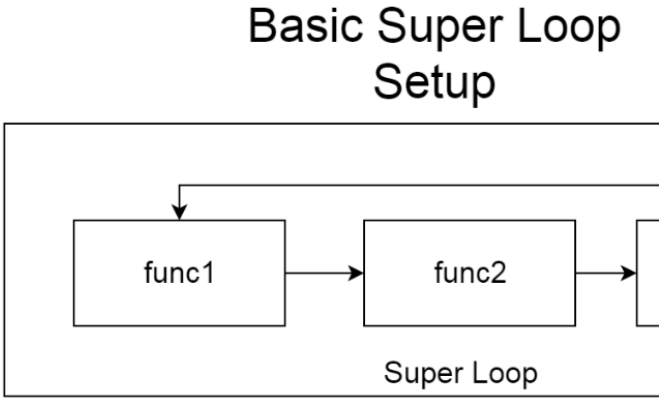
Key points

- The application code is less complex.It is easier to understand—the same primitives are used regardless of the programmer, making it easier to understand code created by different people.The is better hardware portability—with the proper precautions, the code can be run on any hardware supported by the OS without modification.

- The following code represents the basic idea of a super loop. Take a look at this before moving on to the more detailed explanations:void main (void)

```
void main ( void ){ while(1) { func1(); func2(); func3();
//do useful stuff, but don't return //(otherwise, where would we go. . what
would we do. . .?!) }
```

Functions: while loop never returns – it goes on forever executing the same three functions (this is intended). The three innocent-looking function calls can hide some nasty surprises in a real-time system. This main loop that never returns is generally referred to as a super loop. It's always fun to think super because it has control over most things in the system – nothing gets done in the following diagram unless the super loop makes it happen. This type of setup is perfect for very simple systems that need to perform just a few tasks that don't take a considerable amount of time. Basic super loop structures are extremely easy to write and understand; if the problem you're trying to solve can be done with a simple super loop, then use a simple super loop.



The frequency of how often func3 checks the flag is dependent on how long func1 and func2 take to execute. A well designed and responsive super loop will typically execute very rapidly, checking for events more often than they occur (callout B). When an external event does occur, the loop doesn't detect the event until the next time func3 executes (callouts A, C, and D). Notice that there is a delay between when the event is generated and when it is detected by func3.

MCUs also include dedicated hardware for generating interrupts. Interrupts provide signals to the MCU that allow it to jump directly to an interrupt service routine (ISR) when the event occurs. This is such a critical piece of functionality that ARM Cortex-M cores provide a standardized peripheral for it, called the nested vectored interrupt controller.

hHow do interrupts fit into a super loop in a way that better achieves the illusion of parallel activity? The code inside an ISR is generally kept as short as possible, in order to minimize the amount of time spent in the interrupt. This is important for a few reasons. If the interrupt occurs very often and the ISR contains a lot of instructions, there is a chance that the ISR won't return before being called again. For communication peripherals such as UART or SPI, this will mean dropped data (which obviously isn't desirable). Another reason to keep the code short is because other interrupts also need to be serviced, which is why it's a good idea to push off any responsibility to the code that isn't running inside an ISR context

Types of microcontrollers SMT32 and cores;

This microcontroller book starts by introducing you to the concept of RTOS and compares some other alternative methods for achieving real-time performance. Once you've understood the fundamentals, such as tasks, queues, mutexes, and semaphores, you'll learn what to look for when selecting a microcontroller and development environment. By working through examples that use an STM32F7 Nucleo board, the STM32CubeIDE, and SEGGER debug tools, including SEGGER J-Link, Ozone, and SystemView, you'll gain an understanding of preemptive scheduling policies and task communication.

RTOS Guide for LIDAR and Navigation Systems

1. What is an RTOS?

An RTOS is an operating system designed to manage tasks in real time, with predictable response times. It is essential for LIDAR and navigation systems that require precision in managing tasks such as data acquisition and communication.

2. Key Components of an RTOS

2.1 Tareas (Tasks)

Tasks are independent units of execution, similar to processes in a traditional operating system. In a LIDAR system, one task could be in charge of acquiring data from the sensors, another for processing it, and a third for managing communication with other devices.

```
void vTaskFunction(void *pvParameters) {
    while(1) {
        // Code that executes the task
    }
}
```

- Explanation: The cycle `while(1)` ensures that the task is executed continuously. The parameter `pvParameters` allows you to pass data to the task, such as specific configurations.

2.2 Scheduler

The scheduler manages the order in which tasks are executed. In LIDAR systems, the scheduler is crucial to prioritize critical tasks, such as real-time data acquisition.

- Function to start the scheduler:

```
vTaskStartScheduler();
```

- Explanation: This function starts the task scheduler, which organizes and executes tasks according to their priority.

2.3 Task Priorities

Each task has a priority that determines its execution order. In LIDAR systems, tasks such as data acquisition may have a higher priority than sending results, for example.

- Assigning priority to a task:

```
xTaskCreate(vTaskFunction, "TaskName", STACK_SIZE, NULL, PRIORITY, NULL);
```

- Explanation: The function `xTaskCreate` create a task with a stack of size `STACK_SIZE` and a priority defined by `PRIORITY`. The name
- `TaskName` It is optional and serves to identify the task.

2.4 Interrupts

Interrupts are external signals that can stop the execution of a task to execute another function. In LIDAR systems, an interruption could occur when a sensor detects an obstacle.

- Interrupt handling function:

```
void vInterruptHandler(void) {
    // Code to handle the interrupt
}
```

- Explanation: This code is executed when an interrupt occurs, allowing the system to respond quickly to important events.

2.5 Traffic lights and Mutex

Semaphores and mutexes are used to synchronize access to shared resources between tasks. For example, if two tasks in a LIDAR system need to access sensor data, a traffic light ensures that only one task does so at a time.

- Traffic light use:

```
xSemaphoreTake(xSemaphore, portMAX_DELAY);
```

- Explanation: `xSemaphoreTake` take a traffic light. If another process already has the semaphore, the task waits until it is available. `portMAX_DELAY` indicates that the task will wait indefinitely.

2.6 Queues

Queues allow tasks to communicate with each other by sending and receiving messages. In a LIDAR system, a task that acquires data could send it to a queue for another task to process.

```
xQueueSend(xQueue, &data, portMAX_DELAY);
```

- Explanation: `xQueueSend` send `data` to the queue `xQueue`. If the queue is full, the task waits until space is available.

2.7 Timers

Timers allow a function to be executed at regular intervals. In a LIDAR system, they could be used to make measurements at fixed intervals.

- Timer Settings

```
TimerHandle_t xTimer = xTimerCreate("Timer", pdMS_TO_TICKS(1000), pdTRUE, 0, vTimerCallback);
```

Explanation: This function creates a timer that fires every second (1000 milliseconds), calling the function `vTimerCallback`.

3. Application of RTOS in a LIDAR System

3.1 Sensor Data Acquisition

A task dedicated to data acquisition can be executed with high priority, ensuring that sensor data is processed in real time.

- Acquisition task example:

```
void vTaskAcquisition(void *pvParameters) {
    while(1) {
        // Get LIDAR sensor data
        GetLIDARData();
        vTaskDelay(pdMS_TO_TICKS(100));
    }
}
```

- Explanation: The task acquires data every 100 ms. `vTaskDelay` causes the task to wait before the next execution.

3.2 Data Processing

Once acquired, the data can be processed by another task that runs at a lower priority, since processing is not as critical as acquisition.

- Processing task example:

```
void vTaskProcessing(void *pvParameters) {
    while(1) {
        // Process data
        ProcessLIDARData();
        vTaskDelay(pdMS_TO_TICKS(200));
    }
}
```

.3 Communication and Control

An additional task can be responsible for communication with other systems or for controlling actuators based on the processed data.

---

4. Optimization and Considerations

4.1 Load Balance

A well-designed RTOS should balance the load between tasks, preventing less critical tasks from blocking more important ones. Correctly adjusting priorities is key.

4.2 Memory Usage

In embedded systems, it is essential to properly manage the available memory. Correctly defining the size of each task's stacks and queues is essential.

---

5. Conclusion

This guide provides you with a solid foundation for implementing an RTOS in LIDAR systems, describing how to configure tasks, handle interrupts, and synchronize tasks. You can adapt and expand this approach to other types of navigation systems, always ensuring that the design is robust and efficient.

1. FreeRTOS

FreeRTOS is one of the most popular RTOS implementations, especially in embedded systems. It is compatible with a wide variety of microcontrollers, including AVRs, and is flexible enough to be adapted to LIDAR systems.

FreeRTOS Integration with AVR and LIDAR Microcontrollers

To work with FreeRTOS and AVR in a LIDAR system, you must include the FreeRTOS library in your project and configure the environment for AVR microcontrollers.

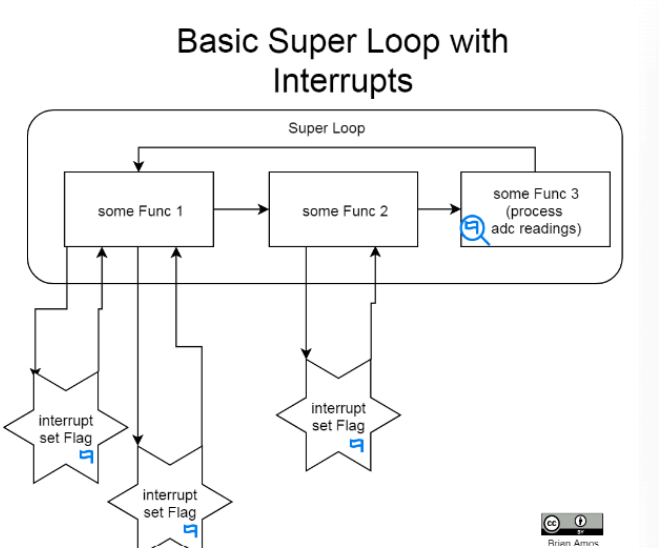
1. Installation and Configuration:
    - Download FreeRTOS from your official repository.
    - Set the `Makefile` or use a compatible IDE

I'm going to integrate this code snippet into the RTOS guide, focusing on how structures and function pointers can be used to drive devices like LEDs within an RTOS-based embedded system. This approach is applicable to LIDAR systems where LEDs could be used for visual or status signals.

---

6. Device Management with RTOS: LEDs

In embedded systems, it is common to handle peripherals such as LEDs to indicate states or alerts. For example, in a LIDAR system, LEDs could indicate whether the sensor is active or if there is an error in data acquisition.



Using Structures and Function Pointers

The following code snippet defines a simple interface for driving LEDs using function pointers, allowing great flexibility in the implementation of the LED driver.

```
#ifndef INTERFACES_ILED_H_
#define INTERFACES_ILED_H_

#ifdef __cplusplus
 extern "C" {
#endif

// Define a function pointer to control LEDs
typedef void (*iLedFunc)(void);

/**
 * This structure contains pointers to functions to turn LEDs on and off
 */
typedef struct
{
    /**
     *On function turns the LED on, regardless of driver logic
     */
```

```
    const iLedFunc On;

    /**
     * Off function turns off the LED, regardless of controller logic
     */
    const iLedFunc Off;
} iWidth;

#ifdef __cplusplus
 }
#endif

#endif /* INTERFACES_ILED_H_ */
```

Code Explanation:

- Pointer to Function (`iLedFunc`): This definition creates a type of function pointer, which points to functions that take no parameters and return nothing. In this case, it will be used to turn the LEDs on and off.
- Structure `iLed`: This structure contains two function pointers:
    - On: Points to the function that turns on the LED.
    - Off: Points to the function that turns off the LED.

This way of defining LED control logic allows different hardware controllers (or even simulations) to implement their own on and off logic, without modifying the main code.

Example of Use in a LIDAR System with FreeRTOS

In a LIDAR system, you could use this structure to control an LED that indicates when the sensor is acquiring data or when there is an error.

```
void TurnOnLed(void) {
    // Logic to turn on the LED (e.g., write to a GPIO pin)
}

void TurnOffLed(void) {
    // Logic to turn off the LED
```

```
}

iLed lidarStatusLed = {
    .On = TurnOnLed,
    .Off = TurnOffLed
};

void vTaskLIDAR(void *pvParameters) {
    while(1) {
        // Data acquisition simulation
        lidarStatusLed.On();  // Turn on the LED while acquiring data
        AcquireLIDARData();   // Function that acquires the data
        lidarStatusLed.Off(); //Turn off the LED when the acquisition is
complete
        vTaskDelay(pdMS_TO_TICKS(500)); // Wait 500 ms before next acquisition
    }
}
```

Integration with RTOS:

- Task `vTask LIDAR`: This task acquires data from the LIDAR sensor. During acquisition, the LED lights up, indicating that the system is operational.
- Using Semaphores or Mutexes: If multiple tasks can access the LEDs, you could use semaphores or mutexes to avoid conflicts.

Considerations:

1. Modularity: By using function pointers, you can easily change the LED driver implementation without modifying the main logic.
2. RTOS: This approach integrates well with FreeRTOS, allowing LEDs to be efficiently controlled by system tasks.

Conclusion

This section shows how to drive simple devices such as LEDs in an RTOS system, applicable to LIDAR systems. Using pointers to functions and structures allows you to write modular and reusable code while maintaining flexible control over peripherals.

Detailed and structured explanation of this code in relation to an RTOS system. The objective is to understand how memory regions and blocks are defined and managed in an embedded system using a microcontroller, in this case under a real-time operating system.

---

7. Memory Management in RTOS

Memory management is a critical aspect in embedded systems and even more so in RTOS (Real-Time Operating System) systems, where efficient use of resources is key to ensuring that critical tasks are executed at the right time. This code defines several regions and blocks of memory in an embedded environment with RTOS.

Memory Region Definitions

1. Define Total Memory:

```
define memory with size = 4G;
```

Here a total memory space of 4 GB is being defined. In practice this value may vary depending on hardware, but is a general statement for the total size available.

```
define symbol _estack = 0x20080000;
```

1. I symbol `_estack` defines the start of the stack at address `0x20080000`. This address is in the RAM region and is where the stack usage starts.
2. RAM Memory Region:

```
define region RAM = [0x20000000 size 0x80000];
```

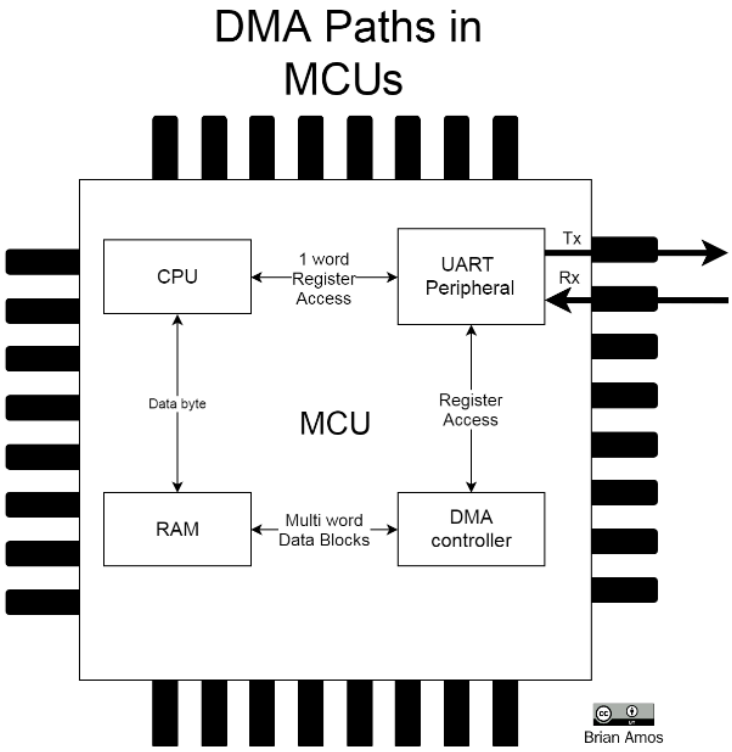
   - Start: `0x20000000`
   - Size: `0x80000` (512 KB)

2. Here the region of RAM memory is defined, which is used to store variables, data, and structures that are needed at runtime.
3. FLASH Memory Region:

```
define region FLASH = [0x8000000 size 0x200000];
```

   - Start: `0x08000000`
   - Size: `0x200000` (2 MB)

2. The FLASH region is where program code and constant data that does not change during execution are stored. FLASH is read-only memory.



The CPU must detect when an individual byte (or word) has been received, either by polling the UART register flags, or by setting up an interrupt service routine that will be fired when a byte is ready. After the byte is transferred from the UART, the CPU can then place it into RAM for further processing.Steps 1 and 2 are repeated until the entire message is received.When DMA is used in the same scenario, the following happens: The CPU configures the DMA controller and peripheral for the transfer.The DMA controller takes care of ALL transfers between the UART peripheral and RAM. This requires no intervention from the CPU.The CPU will be notified when the entire transfer is complete and it can go directly to processing the entire byte stream.

Memory Blocks

Memory blocks are used to organize the arrangement of program sections and other important data within the RAM and FLASH regions.

1. Constructors (ctors) and Destructors (dtors):

```
define block ctors { section .ctors, section .ctors.*, block with
alphabetical order { init_array } };
define block dtors { section .dtors, section .dtors.*, block with
reverse alphabetical order { fini_array } };
```

   - ctors: Stores the constructors (initialization of static variables) in the section `.ctors` and `init_array`.
   - dtors: Store the destructors in the section `.dtors` and `fini_array`.
2. These blocks organize the functions that are called at the beginning and end of the program, ensuring that they are executed in the correct order.
3. Bloques para Thread Local Storage (TLS):

```
define block tbss  { section .tbss,  section .tbss.*  };
define block tdata { section .tdata, section .tdata.* };
define block tls   { block tbss, block tdata };
```

1. Thread Local Storage (TLS) stores specific data for each thread (task). Two blocks are defined here:
   - tbss: Uninitialized data for TLS.
   - tdata: Initialized data for TLS.
2. Heap and Stack Blocks:
3. Heap: The heap is used for dynamic memory allocation at runtime. Its size and alignment are defined with the variables `__HEAPSIZE__` and `alignment = 8`.
4. Stack: The stack is used to store local variables and manage function calls. Its size is defined with `__STACKSIZE__`.

Section Initialization

1. Sections that are not initialized:

```
do not initialize  { section .non_init };
```

1. These sections are initialized by copying the values from FLASH to RAM. The section `.fast` is an example of this.

Location of Sections in Memory

The following code organizes which sections should be placed in RAM and which sections should go in FLASH:

1. Place in FLASH:

```
place in FLASH { section .isr_vector };
keep { section .isr_vector };
place in FLASH { section .init, section .init.*, section .text,
section *.text.*, section .rodata, section .rodata.*, section
.segger.*, block exidx, block ctors, block dtors };
```

- .isr_vector: The interrupt vector table is placed in FLASH, since it does not change during program execution.
- Code: Sections `.text`, `.hot`, and `.rodata` (read-only code and data) are also placed in FLASH.

2. Place in RAM:

```
place in RAM { section .data, section .data.*, section .RamFunc* };
place in RAM { section .bss, section .bss.* };
place in RAM { block heap, block tls, section .non_init, section
.fast };
```

- data: Initialized variables are placed in RAM.
- .bss: Uninitialized variables are also placed in RAM.
- Heap and TLS: Heap memory (heap) and thread local storage (TLS) are placed in RAM.

```
define symbol __stack_end__ = _estack;
```

The symbol `__stack_end__` is defined as the value of `_estack`, indicating the end of the stack.

Conclusion

This code shows how the memory regions and blocks of an embedded system with RTOS are defined and organized. Correct memory layout is essential to ensure the stability and performance of an embedded system, especially in critical applications such as lidar and navigation systems.

With this type of configuration, it is ensured that each section of the program is correctly located in RAM or FLASH, according to its characteristics, and that the system operates efficiently in real time.

In an embedded system with an RTOS (Real-Time Operating System), tasks use RAM in an organized and optimized way for different purposes. Below I explain what each type of task does in RAM and how they manage memory:

1. Section `.data` - Initialized Variables

- Purpose: Stores global and static variables that have been initialized in the source code before execution.

- Use in RAM: When the system boots, the initial values of these variables are copied from FLASH memory to RAM, so that they can be modified during execution.
  Example:

```
int global_var = 10;  // Stored in .data
```

- areas in RTOS: System tasks can use these global variables to share data between them. The RTOS manages its access so that several tasks do not access the same variable at the same time without control, preventing race conditions.

2. Section `.bss` - Uninitialized Variables

- Purpose: Stores global and static variables that are not initialized or initialized to 0. These variables do not take up space in FLASH, but are simply allocated space in RAM.
- Usage in RAM: At program startup, the RTOS or system loader allocates space for these variables in RAM and initializes them to zero.
  Example:

```
int counter;  // Stored in .bss and initialized to 0 by default
```

- areas in RTOS: Tasks that depend on variables that are initialized during execution, such as counters or buffers, will use this area of RAM.

3. Section `.stack` - How many

- Purpose: The stack is used to store local variables, task context (such as CPU registers), and return addresses when calling functions.
- Usage in RAM: Each task in an RTOS generally has its own space on the stack. When a task is executed, the stack is used to store local variables and to make function calls.

  Example:

```
void tarea1(void) {
    int local_var = 5;  // Stored on the stack
}
```

- areas in RTOS: Stacks are crucial in managing task context. When the RTOS switches from one task to another (context switch), it saves the state of the current task (registers, stack pointer) and restores that of the next task on the stack, so that it continues from where it stopped.

4. Section `.heap` - Heap

- Purpose: The heap is used for dynamic memory allocation during program execution. It is used when the data size is not known at compile time.
- Usage in RAM: Heap memory is requested and freed dynamically through functions such as `malloc()` and `free()`. It is a more flexible memory area, but needs manual or automatic management to avoid fragmentation.

- Example:

```
int* ptr = (int*)malloc(sizeof(int));  // Memory is allocated
on the heap
```

- Tasks in RTOS: Tasks that need memory space for data that changes size or is not known in advance (such as dynamic data buffers) use the heap. However, its use must be carefully managed, as misuse could cause memory leaks.

5. Section `TLS` - Thread Local Storage

- Purpose: TLS (Thread Local Storage) is a special section in RAM reserved for storing variables that are specific to each task (thread). These variables are not shared between tasks.
- Usage in RAM: Each task has its own TLS space to store local variables that only that task can modify or read.
  Example:

```
__thread int tls_var = 0;  // See all in TLS
```

- Tasks in RTOS: Each task has its own TLS space. This is useful in multithreaded systems, as it allows each task to have its own data without interfering with that of other tasks.

6. Section `.tdata` and `.tbss` - Task Specific Data

- Purpose: These sections manage the data that is initialized and uninitialized for tasks (threads) of an RTOS.
  - .tdata: Stores local variables of tasks that have been initialized.
  - .tbss: Stores local variables of uninitialized tasks.

- RAM Usage: Similar to `.data` and `.bss`, but in this case they are specific to tasks and not to the entire program.

  Example:

```
static __thread int task_specific_var = 100;  // Se almacena en
.tdata
```

areas in RTOS: Each task has its own data that can be initialized in `.tdata` the `.tbss`, thus preventing tasks from interfering with each other.

7. Section `.isr_vector` - Vector of Interruptions

- Purpose: The interrupt vector table contains the addresses of the interrupt handling (ISR) functions. This table is usually in FLASH memory, but the ISR code runs in RAM.
- Use in RAM: Interrupt handlers use the stack of the current task to save state and process the interrupt. Temporary variables and context are stored on the stack.
- Tasks in RTOS: Although interrupts are handled outside of RTOS tasks, the RTOS can use synchronization mechanisms (such as semaphores or queues) to have an ISR wake up or notify a task in response to an interrupt.

Conclusion

In an RTOS, RAM is organized into different sections to efficiently manage resources. Each task in an RTOS has its own spaces in RAM (such as stack and TLS) and some sections are shared by all tasks (such as `.data` and `.bss`). This design allows tasks to operate independently, with private or shared data, while the RTOS manages context switching and resource access safely and efficiently.

Understanding how each task uses RAM will allow you to design and optimize RTOS-embedded systems, such as LIDAR systems, more efficiently.

This linker script file (`.ld`) defines the memory layout and sections for an STM32F767ZI microcontroller with 2 MB FLASH memory and 512 KB RAM. This script is key to building a project because it specifies how code and data segments are organized and placed in memory. Below, I detail the meaning of each important part:

1. Entry Point

```
ENTRY(Reset_Handler)
```

- Purpose: Defines the entry point of the program, which in this case is the function `Reset_Handler`. It is where execution begins after a microcontroller reset.

2. Establishment of Stack and Heap

```
_estack = 0x20080000;    /* end of RAM */
_Min_Heap_Size = 0x200;  /* 512 bytes para heap */
_Min_Stack_Size = 0x400; /* 1 KB para stack */
```

- _ stack: Defines the highest point of the stack in RAM (in this case, at the end of the 512 KB of RAM, that is, at address `0x20080000`).
- _Min Heap_Size and _Min Stack_Size: Defines the minimum sizes for the heap (512 bytes) and stack (1 KB), used for dynamic allocation and local variables in tasks.

3. Memory Areas (MEMORY)

```
MEMORY { RAM (xrw) : ORIGIN = 0x20000000, LENGTH = 512K FLASH (rx) : ORIGIN = 0x8000000, LENGTH = 2048K }
```

- RAM: Declared as a 512 KB block from the address `0x20000000`. The `xrw` It means that it has execute, read and write permissions.
- FLASH: Defines a 2 MB block of flash memory starting at `0x08000000`. Used for read and execute only (`rx`), which corresponds to the storage of the code and constants.

4. Output Sections (SECTIONS)

Output sections indicate where different types of data and code are placed within memory areas. Some key sections are:

a) Section `.isr_vector`

```
.isr_vector : { . = ALIGN(4); KEEP(*(.isr_vector)) /* Startup code */ . = ALIGN(4); } >FLASH
```

- Purpose: This section stores the interrupt vector, which contains the addresses of the interrupt handling routines (ISR). It is placed in the FLASH.

b) Section `.text`

```
.text : { . = ALIGN(4); *(.text) /* .text sections (code) */ *(.text*) /* .text* sections (code) */ . = ALIGN(4); _etext = .; /* Marks the end of the code */ } >FLASH
```

- Purpose: The program code and any other read-only data is placed here. It is stored in FLASH so that it is not modified during execution.
- _etext: Marks the end of the section `.text`.

c) Section `.rodata`

```
.rodata : { . = ALIGN(4); *(.rodata) /* Read-only data and constants */ . = ALIGN(4); } >FLASH
```

- Purpose: Stores read-only data such as constants and strings. It's also in FLASH.

d) Section `.data`

```
.data : { . = ALIGN(4); _sdata = .; /* Start of data section */ *(.data) /* Initialized variables */ *(.data*) . =ALIGN(4); _agea = .; /* End of data section */ } >RAM AT> FLASH
```

- Purpose: Stores the global and static variables that have been initialized. They are placed in RAM during execution, but copied from FLASH during boot.
- _sdata and _edata: Mark the beginning and end of the section `.data`.

e) Section `.bss`

```
.bss : { _sbss = .; /* Start of the .bss section */ *(.bss) /* Uninitialized variables */ *(.bss*) *(COMMON) . =ALIGN(4); _ebss = .; /* End of .bss section */ } >RAM
```

- Purpose: Stores uninitialized global and static variables. These variables are initialized to 0 when the program starts. They are placed directly in RAM.
- _sbss and _ebss: Mark the beginning and end of the section `.bss`.

f) Section `.heap` and `.stack`

```
._user_heap_stack : { . = ALIGN(4); PROVIDE ( end = . ); PROVIDE ( _end = . ); . = . + _Min_Heap_Size; . = . + _Min_Stack_Size; . = ALIGN(4); } >RAM
```

- Purpose: This section ensures that there is enough RAM space for the program heap and stack. Makes sure there are at least `_Min_Heap_Size` bytes for the heap and `_Min_Stack_Size` bytes for the stack.

5. Section `DISCARD`

```
/DISCARD/ : { libc.a ( * ) libm.a ( * ) libgcc.a ( * ) }
```

- Purpose: This section automatically discards any unwanted content from the standard libraries that are not needed to reduce the size of the binary.

Conclusion

This script defines the memory organization on an STM32F767ZI microcontroller. FLASH is used to store program code and read-only data, while RAM is used for global variables, the heap, and the stack. symbols like `_sdata`, `_sbss`, `_edata`, etc., help the loader to correctly initialize the data sections when the program starts.

This configuration kernel belongs to an STM32 project, probably created or managed through STM32CubeMX, for an STM32F767ZITx microcontroller. The file contains settings related to various peripherals, such as ETH (Ethernet), FREERTOS, USART3, and USB OTG FS, in addition to the clock settings and GPIO pins for the microcontroller. Some key points of this setup:

1. Peripherals enabled: Interfaces such as ETH (Ethernet using LAN8742A driver) and USB OTG_FS are being used for USB functionality. Support for FREERTOS is also configured.
2. GPIO Pin Configuration: GPIO pins are configured for their respective functions such as RMII reference clock pins for Ethernet, USB data pins and debug pins such as TMS, TCK (Serial Wire Debug). Additionally, some pins are intended for LEDs and buttons on the board.
3. Clock Configuration: The clock is configured to obtain specific frequencies for each block of the system, such as the AHB and APB buses, which power different peripherals, including Ethernet, USB, and other components.
4. FreeRTOS: The FreeRTOS real-time operating system is configured to execute a predetermined task.
5. NVIC Interrupt System: Critical interrupts such as HardFault, SysTick, MemoryManagement, and others are enabled and prioritized.

This file is essential to configure the hardware environment before generating the initialization code with STM32CubeMX or any other development tool. It is noted that this is a complex project that involves Ethernet and USB communication, in addition to handling multiple tasks with FreeRTOS.

The code describes the configuration of an STM32 microcontroller (specifically the STM32F767ZITx), and contains various configurations for peripherals such as LED, USB, Ethernet (RMII), and the FreeRTOS stack. Here I explain the most relevant elements at the level of LED sensors and peripherals in the context of a real-time operating system (RTOS).

1. FreeRTOS

- FREERTOS.IPParameters=Tasks01: FreeRTOS is a real-time operating system that manages multiple tasks. This parameter configures system tasks, such as `defaultTask`, and assigns resources as a priority (`24`), stack size (`128`), and the function that executes the task (`StartDefaultTask`).
- Use in Sensors and LEDs: The RTOS manages the time and resources of the microcontroller, ensuring that sensors (such as the user button) and LEDs (LD2, LD3) can be monitored and controlled efficiently, even if other tasks are running in parallel.

2. LED configuration (PB14 and PB7):

- PB14.GPIO_Label=LD3 [Network]: This configures pin PB14 as a digital output to drive the LD3 red LED. `PB14.Signal=GPIO_Output` specifies that this pin will operate in GPIO output mode, meaning that the program will be able to turn this LED on or off.

- PB7.GPIO_Label=LD2 [Blue]: Similarly, pin PB7 controls the blue LED LD2, also in GPIO output mode. Both LEDs can be used to signal states or events in the system.
- Use in FreeRTOS: Typically, you can create a task in FreeRTOS to drive the LEDs. For example, an LED can flash to indicate that the system is operational or turn on/off in response to sensor events (such as a signal from a temperature or motion sensor).

These are the types of systems where an RTOS is needed. Guaranteeing that the most time-critical tasks are always running when necessary and scheduling lower priority tasks to run whenever spare time is available is a strong point of preemptive schedulers. In this type of setup, the critical sensor readings could be pushed into their own task and assigned a high priority – effectively interrupting anything else in the system (except ISRs) when it was time to deal with the sensor. That complex communication stack could be assigned a lower priority than the critical sensor.

3. User Button (PC13):

- PC13.GPIO_Label=USER_Btn [B1]: Pin PC13 is configured as a GPIO input representing the user button `USER_Btn`. The system can read the state of this pin to detect if the button has been pressed.
- GPXTI13: This signal allows interrupts, that is, an interrupt can be generated in the system when the button is pressed, activating a specific task in FreeRTOS.
- Use in FreeRTOS: The button can be used as a trigger to change the state of the LEDs or initiate some task (for example, reboot the system or activate a special mode). In an RTOS system, this action could be mapped to an ISR (Interrupt Service Routine) that wakes up or unlocks a task.

4. Ethernet (RMII):

- RMII (Ethernet) Peripherals: Pins such as PA1, PA2, PA7, PC1, PC4, etc., are configured to handle the RMII (Reduced Media Independent Interface) interface that allows Ethernet communication through an external PHY (`LAN8742A`). This is essential in embedded systems that need network connectivity.
- ETH.MediaInterface=ETH_MEDIA_INTERFACE_RMII: Configures the media interface as RMII, a common physical interface for Ethernet on STM32 systems.
- Use in FreeRTOS: In more complex systems, FreeRTOS can handle networking tasks, such as sending/receiving packets using a TCP/IP stack, controlled by the Ethernet RMII configuration. This type of interface is vital if the microcontroller needs to communicate with other systems or networks.

5. USB OTG (On-The-Go):

- PA9.GPIO_Label=USB_VBUS: This pin detects if there is power on the USB bus, which is necessary to operate as a USB device. The microcontroller can act as a USB device or host (for example, to read USB devices or be detected as a mass storage device).
- PA11.GPIO_Label=USB_DM, PA12.GPIO_Label=USB_DP: These are the pins of the USB data lines. They are configured to handle USB signals in device mode (`Device_Only`).
- Use in FreeRTOS: USB applications, such as uploading files or communicating with USB peripherals, can be managed in FreeRTOS tasks, allowing the system to respond to USB events while executing other concurrent tasks (such as reading sensors or LED control).

6. Peripheral Clock and Speed Configuration (RCC):

- RCC.AHBFreq_Value=216000000: Sets the AHB (Advanced High-performance Bus) bus speed to 216 MHz, which is essential for the overall performance of the microcontroller. This ensures that peripherals, such as Ethernet and USB, have sufficient bandwidth.
- RCC.APB1Freq_Value=54000000: The APB1 bus operates at 54 MHz, being responsible for handling lower speed peripherals, such as USART, which could be used for serial communication in FreeRTOS systems.

7. USART (PD8, PD9):

- PD8.GPIO_Label=STLK_RX, PD9.GPIO_Label=STLK_TX: These pins are configured for UART (Asynchronous) communication, allowing data transmission and reception through USART3.
- Use in FreeRTOS: FreeRTOS can handle serial communication, for example to debug the system or communicate sensor data to a computer. In FreeRTOS code, you can have a task that sends periodic data over USART.

Conclusion:

This code configures several key peripherals (LEDs, USB, Ethernet, USART) and sensors (User Button) that can then be controlled in the context of a FreeRTOS system. FreeRTOS manages concurrent tasks so that the microcontroller can, for example, monitor the user button, control LEDs, handle USB and Ethernet communication, and operate UART peripherals, all at the same time. This allows us to design efficient, modular embedded systems with multitasking capabilities.

```
The main RTOS file from src: #include <FreeRTOS.h>
```

```
#include <Nucleo_F767ZI_Init.h>

#include <stm32f7xx_hal.h>

#include <Nucleo_F767ZI_GPIO.h>

#include <task.h>

#include <SEGGER_SYSVIEW.h>
```

classic example of a system with FreeRTOS running on an STM32 microcontroller (specifically the Nucleo-F767ZI) that handles various tasks and LEDs through multitasking. I explain each part of the code, focusing on how the peripherals (LEDs) interact with FreeRTOS, and some important concepts for its operation:

1. Includes

```
#include <FreeRTOS.h> #include <Nucleo_F767ZI_Init.h> #include <stm32f7xx_hal.h>
#include <Nucleo_F767ZI_GPIO.h> #include <task.h> #include <SEGGER_SYSVIEW.h>
```

- FreeRTOS.h: FreeRTOS main header, which includes data types and functions for the real-time operating system.
- Nucleo_F767ZI_Init.h and Nucleo_F767ZI_GPIO.h: These headers are specific to the Nucleo-F767ZI board and contain the hardware configuration (GPIOs for the LEDs).
- SEGGER_SYSVIEW.h: Used for system debugging and tracing using SEGGER SystemView, which is a tool for monitoring the real-time behavior of the FreeRTOS system.

2. Function Prototypes

```
void Task1(void *argument); void Task2(void *argument); void Task3(void *argument); void lookBusy(void);
```

Here the prototypes of the tasks are declared (`Task1`, `Task2`, `Task3`) and an auxiliary function (`lookBusy`). These tasks will be managed by the FreeRTOS kernel.

3. Function `main`

```
int main(void) { const static uint32_t stackSize = 128; HWinit(); SEGGER_SYSVIEW_Conf(); HAL_NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_4);
```

- `stackSize`: Each task has its own stack. A stack size is defined `128 * 4 = 512 bytes` for each task. It's a reasonable amount for simple tasks.
- `HWinit()`: Initializes the board hardware (peripherals such as LEDs, etc.).
- `SEGGER_SYSVIEW_Conf()`: Configure the plot with SEGGER SystemView.
- `HAL_NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_4)`: Configures interrupt priority grouping. FreeRTOS depends on this to correctly handle task priorities and interrupts.

Then the tasks are created with the function `xTaskCreate`:

```
if (xTaskCreate(Task1, "task1", stackSize, NULL, tskIDLE_PRIORITY + 2, NULL) == pdPASS)
```

- `xTaskCreate(Task1, "task1", stackSize, NULL, tskIDLE_PRIORITY + 2, NULL)`: Create `Task1` with a priority of `tskIDLE_PRIORITY + 2`. The tasks `Task2` and `Task3` They are created similarly.
- `vTaskStartScheduler()`: Start the FreeRTOS scheduler. Once the scheduler is started, control of program flow is taken over by FreeRTOS.

If any task cannot be created, the code goes into an infinite loop:

```
while(1) { }
```

4. Tasks (`Task1`, `Task2`, `Task3`)

Each task follows the same basic pattern, but with different LEDs and messages.

`Task1`:

```
void Task1(void *argument) { while(1) {      SEGGER_SYSVIEW_PrintfHost("hey there!\n");   GreenLed.On();     vTaskDelay(105/ portTICK_PERIOD_MS);  GreenLed.Off();     vTaskDelay(100/ portTICK_PERIOD_MS); } }
```

- This task turns the green LED on and off (`GreenLed.On()` and `GreenLed.Off()`) with a delay of approximately 105 ms on and 100 ms off.
- `vTaskDelay()`: Causes the task to suspend for the specified time (105 or 100 milliseconds). This allows other tasks to run.

`Task2`:

```
void Task2( void* argument ) {  while(1)     {
SEGGER_SYSVIEW_PrintfHost("task 2 says 'Hi!'\n");        BlueLed.On();
vTaskDelay(200 / portTICK_PERIOD_MS);              BlueLed.Off();
vTaskDelay(200 / portTICK_PERIOD_MS);        } }
```

- Similar a `Task1`, this task turns the blue LED on and off (`BlueLed.On()` and `BlueLed.Off()`) with a delay of 200 ms between states.

`Task3`:

```
void Task3( void* argument ) {  while(1)     {            lookBusy();
SEGGER_SYSVIEW_PrintfHost("task3\n");              RedLed.On();
vTaskDelay(500/ portTICK_PERIOD_MS);         RedLed.Off();
vTaskDelay(500/ bytTICK_PERIOD_MS);   } }
```

- This task turns the red LED (`RedLed.On()` and `RedLed.Off()`) with a delay of 500 ms. Also calls the function `lookBusy()` before turning on the LED.

5. Function `lookBusy`

```
void lookBusy( void ) {  volatile uint32_t __attribute__((unused)) dontCare = 0;
for(int i = 0; i < 50E3; i++)   {            dontCare = i % 4;  }
SEGGER_SYSVIEW_PrintfHost("looking busy\n"); }
```

- This function simulates a task that "does something" but with no real effect. In the loop, it simply performs an arithmetic operation to occupy CPU time.

- `SEGGER_SYSVIEW_PrintfHost()`: Sends a message to appear in the SEGGER SystemView debugging console.

Conclusion

The code defines three tasks that manage three LEDs (green, blue and red). Each task is executed independently, with a specific delay between the LEDs turning on and off. FreeRTOS allows these tasks to be executed concurrently, dividing the CPU time between them.

- Task 1 (Task1): The green LED flashes.
- Task 2 (Task2): The blue LED flashes.
- Task 3 (Task3): Flashes the red LED and simulates a workload (`lookBusy`).

Each task behaves cooperatively, releasing CPU control to other tasks by `vTaskDelay()`, ensuring effective multitasking.

examples and clarifications so that you understand how the code works in the context of initializing and deinitializing peripherals on an STM32 board, using the HAL (Hardware Abstraction Layer) from STMicroelectronics.

1. `HAL_MspInit()`

This function initializes the MSP (Microcontroller Support Package) support system. It is called by the initialization code at startup to enable global clocks and set system interrupt priorities.

c

Copy code

```
void HAL_MspInit(void) { /* Enable clock for power controller and system controller */ __HAL_RCC_PWR_CLK_ENABLE(); __HAL_RCC_SYSCFG_CLK_ENABLE(); /*
```

```
Setting interrupt priority for PendSV_IRQn */ HAL_NVIC_SetPriority(PendSV_IRQn, 15, 0); }
```

- `__HAL_RCC_PWR_CLK_ENABLE()`: Enable the clock for the power controller.
- `__HAL_RCC_SYSCFG_CLK_ENABLE()`: Enables the clock for the system controller (SYSCFG).
- `HAL_NVIC_SetPriority(PendSV_IRQn, 15, 0)`: Sets the priority of the PendSV interrupt (one of the system exceptions) to a low priority (15).

---

2. `HAL_ETH_MspInit(ETH_HandleTypeDef* heth)`

This function initializes the hardware resources for the Ethernet (ETH) controller on the microcontroller. Configure the GPIO pins and enable the associated clocks.

```
void HAL_ETH_MspInit(ETH_HandleTypeDef* heth) { if (heth->Instance == ETH) { /* Enable the clock for the Ethernet peripheral */ __HAL_RCC_ETH_CLK_ENABLE(); /* Configuration of the GPIO pins used by the ETH peripheral */ GPIO_InitTypeDef GPIO_InitStruct = {0}; /* Configuration of the pins of the different ports */ GPIO_InitStruct.Pin = RMII_MDC_Pin | RMII_RXD0_Pin | RMII_RXD1_Pin; GPIO_InitStruct.Mode = GPIO_MODE_AF_PP; GPIO_InitStruct.Pull = GPIO_NOPULL; GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH; GPIO_InitStruct.Alternate = GPIO_AF11_ETH; HAL_GPIO_Init(GPIOC, &GPIO_InitStruct); /* Additional configuration for the remaining ETH pins */ // Similar configurations for GPIOA, GPIOB, and GPIOG. } }
```

- Peripheral clock: `__HAL_RCC_ETH_CLK_ENABLE()` enables the clock for the Ethernet module.

- Pin configuration: Each pin block, e.g. `RMII_MDC_Pin`, is associated with a function of the Ethernet peripheral. The pins are set to GPIO_MODE_AF_PP (alternate mode with push-pull output), without pull-up/down resistor.

---

3. `HAL_UART_MspInit(UART_HandleTypeDef* huart)`

This function initializes the hardware resources used for UART communication (USART3 in this case).

c

Copy code

```
void HAL_UART_MspInit(UART_HandleTypeDef* huart) { if (huart->Instance == USART3) { /* Enable the clock for the USART3 peripheral */ __HAL_RCC_USART3_CLK_ENABLE(); /* Configuration of the TX and RX pins */ GPIO_InitTypeDef GPIO_InitStruct = {0}; GPIO_InitStruct.Pin = STLK_RX_Pin | STLK_TX_Pin; GPIO_InitStruct.Mode = GPIO_MODE_AF_PP; GPIO_InitStruct.Pull = GPIO_NOPULL; GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH; GPIO_InitStruct.Alternate = GPIO_AF7_USART3; HAL_GPIO_Init(GPIOD, &GPIO_InitStruct); } }
```

- USART3: The UART communication peripheral (USART3) is activated using `__HAL_RCC_USART3_CLK_ENABLE()`.
- TX and RX Pins: The transmit (TX) and receive (RX) pins are configured in alternative mode with high speed and without pull-up/pull-down resistors.

---

4. `HAL_PCD_MspInit(PCD_HandleTypeDef* hpcd)`

This function initializes the USB OTG FS (On-The-Go Full Speed) peripheral on the microcontroller.

```
void HAL_PCD_MspInit(PCD_HandleTypeDef* hpcd) { if (hpcd->Instance == USB_OTG_FS) { /* Enable clock for USB OTG FS peripheral */ __HAL_RCC_USB_OTG_FS_CLK_ENABLE(); /* Pin configuration for USB OTG FS */ GPIO_InitTypeDef GPIO_InitStruct = {0}; GPIO_InitStruct.Pin = USB_SOF_Pin | USB_ID_Pin | USB_DM_Pin | USB_DP_Pin; GPIO_InitStruct.Mode = GPIO_MODE_AF_PP; GPIO_InitStruct.Pull = GPIO_NOPULL; GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH; GPIO_InitStruct.Alternate = GPIO_AF10_OTG_FS; HAL_GPIO_Init(GPIOA, &GPIO_InitStruct); } }
```

- USB OTG FS: `__HAL_RCC_USB_OTG_FS_CLK_ENABLE()` Enables clocking for the USB OTG FS peripheral.
- USB Pins: The pins are configured for their respective functions with the corresponding alternate mode.

Feature Summary

- MSP Init/DeInit: These functions are responsible for configuring the hardware resources (pins, clocks, etc.) that a specific peripheral needs. For example, `HAL_UART_MspInit()` initializes the hardware associated with the UART, such as the GPIO pins and the clock.
- GPIO_InitTypeDef: It is a structure that defines the configuration parameters for the GPIO pins. The most important fields include `Pin`, `Mode`, `Pull`, `Speed`, and `Alternate` to define the pin configuration, its speed and the alternative function.

This file is crucial for configuring peripherals on STM32 microcontrollers, as it manages the physical resources (such as pins and clocks) that allow the peripherals to function correctly.

Interrupt Service Routines (ISRs) *Interrupt Service Routines*) of an RTOS system (FreeRTOS) on an STM32F7 microcontroller. Let's break down some key points to understand it better:

1. Cortex-M7 Processor Interruption and Exception Handlers:

- The file contains interrupt handlers for the Cortex-M7 processor. These are default functions that are invoked in response to various types of processor exceptions, such as memory errors, bus faults, and others.
- For example:
    - `NMI_Handler()` handles non-maskable interrupt (*Non-Maskable Interrupt*), which is a higher priority interrupt.
    - `HardFault_Handler()` runs when a critical hardware failure occurs.
    - `MemManage_Handler()` handles memory management errors.
    - `BusFault_Handler()` Handles memory access errors due to bus failures.
    - `UsageFault_Handler()` is executed if an undefined instruction or illegal state occurs.

2. SysTick_Handler:

- This is one of the most important parts for the RTOS. The SysTick_Handler is responsible for handling system timer interrupts, which are essential for context switching in a multitasking system like FreeRTOS.
- FreeRTOS uses `xPortSysTickHandler()` to manage the *scheduling* of tasks. This handler is invoked periodically to change the state of tasks, allowing the RTOS to manage the execution of multiple tasks concurrently.
- The block `#if (INCLUDE_xTaskGetSchedulerState == 1 )` checks if the FreeRTOS scheduler has already been started. If it has been, it invokes the handler of the *tick* from FreeRTOS (`xPortSysTickHandler`).

3. Failure Handling:

- The functions like `HardFault_Handler()`, `MemManage_Handler()`, `BusFault_Handler()`, and `UsageFault_Handler()` they go into infinite loops (`while(1)`) once the corresponding fault occurs. This is common in embedded systems to stop execution when a critical error occurs.

4. User Personalization:

- Blocks labeled `USER CODE` They are reserved for the developer to insert their own code. These comments (`/* USER CODE BEGIN... */`) help keep user code separate from code automatically generated by tools like STM32CubeMX, allowing future regenerations to not overwrite user code.

This code is associated with the core of the real-time operating system (RTOS), where the interrupt handler and FreeRTOS functions play a key role in multitasking, timing, and error management in the system.

RTOS architecture

1. RTOS Kernel: It is the core of the system and manages tasks, interrupts and resources. FreeRTOS is a preemptive kernel, meaning it can interrupt a running task to execute another task of higher priority.
2. Tasks: Each task is an independent thread of execution. These tasks have defined priorities, which in FreeRTOS can range from 0 (the lowest priority) to a maximum value, defined in `configMAX_PRIORITIES`. In this example, tasks can have up to 56 priority levels. Tasks can be in states like *Running* (running), *Blocked* (waiting for an appeal) o *Suspended* (waiting to be resumed).
3. Interrupts: Interrupt service routines (ISRs) respond to external events, such as hardware interrupts. In the code, functions like `SysTick_Handler` They handle time interrupts (SysTick). ISRs in FreeRTOS can invoke safe functions from interrupts as long as their priority is set correctly (`configMAX_SYSCALL_INTERRUPT_PRIORITY`).
4. Scheduling: The FreeRTOS scheduler is responsible for allocating CPU time to tasks based on their priorities. This is done with the kernel scheduler, which executes the task with the highest available priority.
5. SysTick Configuration: The Timer `SysTick` is used to generate system timing interrupts, allowing FreeRTOS to schedule tasks at regular intervals. In this case, a rate of 1000 ticks per second (1ms per tick) is configured, defined in `configTICK_RATE_HZ`.
6. Synchronization: To coordinate tasks and shared resources, FreeRTOS provides mechanisms such as semaphores, mutexes, and queues, which allow synchronization between tasks or between tasks and interrupts.

FreeRTOS Technical Details

1. Tasks and Priorities:
    - `configMAX_PRIORITIES (56)` defines how many priority levels tasks can have. Higher priority tasks will interrupt lower priority tasks.
    - `configMINIMAL_STACK_SIZE (128)` indicates the minimum stack size that is assigned to a task.
2. Interruptions:
    - `configKERNEL_INTERRUPT_PRIORITY (15)` defines the lowest priority for interrupts managed by the RTOS kernel.
    - `configMAX_SYSCALL_INTERRUPT_PRIORITY (5)` Sets the maximum priority that an interrupt that calls safe FreeRTOS functions can have. That is, any ISR that makes use of FreeRTOS APIs must have a priority between 5 and 15 (lower numbers indicate higher priority).
3. Buses y Pines:
    - In STM32 systems with Cortex-M7, data can travel through several buses, such as the APB (Advanced Peripheral Bus) or AHB (Advanced High-performance Bus), which connect the peripherals to the CPU.
    - Operations that involve communication with the outside world, such as handling GPIO pins, SPI communication, or UART, are configured through these buses. However, the specific details of which pin the data travels to depend on the peripheral you are configuring.
4. FreeRTOS configurations:
    - `configUSE_MUTEXES`, `configUSE_COUNTING_SEMAPHORES` and `configUSE_RECURSIVE_MUTEXES` They enable the use of synchronization mechanisms such as semaphores and mutexes.
    - `configUSE_TIMERS` enables the use of software timers, which can also be used to create timed events in the system.

Byte Flow and Peripherals

- Communication buses such as the AHB, APB or the DMA (Direct Memory Access) bus are responsible for transferring data between the memory and the peripherals (UART, SPI, ADC, etc.).
- The input/output (GPIO) pins are configured for direct communication with external hardware. The bytes that travel on these buses will depend on the type of peripheral that is configured. For example:
    - In a UART system, data travels from RAM to the UART controller and then to the GPIO pins configured as TX (transmit) and RX (receive).
    - In SPI, bytes travel over the SPI bus to the MOSI/MISO pins.

Examples of Functions in FreeRTOS

1. vTaskDelay: Suspends the current task for a period of time defined by the number of ticks.
2. xTaskCreate: Creates a new task with its own stack and priority.
3. xSemaphoreTake: Used to acquire a semaphore or block if it is not available.
4. xQueueSend/xQueueReceive: Send or receive data through queues, used to communicate data between tasks.

These are the key concepts of RTOS operation, the relationship with communication buses and the management of tasks and peripherals.

UART Configuration Example with FreeRTOS

1. Hardware Configuration

First, make sure your project is configured correctly on STM32CubeMX. Select the USART peripheral and configure it. Here are some basic steps:

- Select the USART port (for example, USART2).
- Configure parameters such as speed (baud rate), number of data bits, parity and stop bits.
- Generate the code and open your IDE (for example, STM32CubeIDE).

2. UART Initialization Code

In the generated initialization file, you will find the function `MX_USART2_UART_Init()`. Here the UART is configured. You don't need to change anything, but make sure the settings are correct.

3. Create the UART Task

Next, we create a task that will send data over the UART. Here is an example:

```
#include "FreeRTOS.h" #include "task.h" #include "usart.h" // Make sure to include the header generated for UART #define UART_TX_QUEUE_LENGTH 5
#define UART_TX_QUEUE_ITEM_SIZE sizeof(char) // Glue to send data via UART static QueueHandle_t uartTxQueue; void vTaskSendUART(void *pvParameters) { char message[20]; for(;;) {
// Here you can change the message as needed sprintf(message, "Hello FreeRTOS!\r\n"); // Send the message to the queue for (int i = 0; i < strlen(message); i++) { xQueueSend(uartTxQueue, &message[i], portMAX_DELAY); } // Delay the task for 1000 ms vTaskDelay(pdMS_TO_TICKS(1000)); } } void UART_Send_Character(char character) {
// Wait until the queue has data if (xQueueReceive(uartTxQueue, &character, portMAX_DELAY) == pdTRUE) { HAL_UART_Transmit(&huart2, (uint8_t *)&character, 1, HAL_MAX_DELAY); } } // Main function int main(void) { // Initialize hardware HAL_Init(); SystemClock_Config(); MX_GPIO_Init(); MX_USART2_UART_Init(); // Create the queue for UART uartTxQueue = xQueueCreate(UART_TX_QUEUE_LENGTH, UART_TX_QUEUE_ITEM_SIZE); // Create the UART send task xTaskCreate(vTaskSendUART,
```

```
"TaskSendUART", 128, NULL, 1, NULL); // Start the scheduler vTaskStartScheduler();
// Infinite loop (should never get here) for(;;); }
```

4. Explanation of the Code

- Transmission Queue: A queue is created `uartTxQueue` to send characters to the UART. This helps to manage data asynchronously.
- Task `vTaskSendUART`: This task generates a message and sends it character by character to the queue `uartTxQueue` every second.
- Function `UART_Send_Character`: This function receives a character from the queue and sends it through the UART. Use the function `HAL_UART_Transmit` from the STM32 HAL library.
- Initialization: In `main()`, we initialize the hardware, create the queue and the task, and then start the FreeRTOS scheduler.

5. SysTick configuration

The function code `SysTick_Handler` should already be configured correctly in `stm32f7xx_it.c`, and FreeRTOS will take care of managing the time ticks.

Summary

This is a basic example of how to configure a UART and use FreeRTOS to send data. The queue allows asynchronous communication, which is essential in an RTOS system, since tasks can be executed independently of data transmission.