My Project

Generated by Doxygen 1.9.1

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 3

Class Documentation

3.1 TaskControlBlock Struct Reference

Represents a task in the task scheduler.

`#include <main.h>`

Public Attributes

- uint32_t **stack_pointer**
- uint32_t **remaining_ticks**
- uint8_t **task_state**
- void(* **task_function**)(void)

3.1.1 Detailed Description

Represents a task in the task scheduler.

This structure contains all the necessary information to manage a task. It includes the stack pointer, the number of ticks remaining until the task can run again, the current state of the task, and a pointer to the task's execution function.

The documentation for this struct was generated from the following file:

- Inc/main.h

Chapter 4

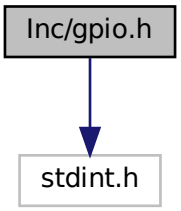
File Documentation

4.1 Inc/gpio.h File Reference

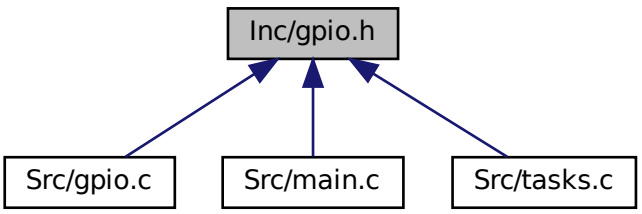
Header file for GPIO control functions.

```
#include <stdint.h>
```

Include dependency graph for gpio.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define **GPIO_PIN_D12** (1 << 12)
- #define **GPIO_PIN_D13** (1 << 13)
- #define **GPIO_PIN_D14** (1 << 14)
- #define **GPIO_PIN_D15** (1 << 15)
- #define **RCC_BASE** 0x40023800
- #define **GPIOD_BASE** 0x40020C00
- #define **RCC_AHB1ENR** (*(volatile uint32_t *)(RCC_BASE + 0x30))
- #define **GPIOD_MODER** (*(volatile uint32_t *)(GPIOD_BASE + 0x00))
- #define **GPIOD_ODR** (*(volatile uint32_t *)(GPIOD_BASE + 0x14))

Functions

- void gpio_init (void)

  Initializes the GPIO port D.
- void toggle_gpio_pin (uint32_t pin)

  Toggles the state of the specified GPIO pin.

4.1.1 Detailed Description

Header file for GPIO control functions.

This file contains function prototypes for GPIO initialization and manipulation.

Author

Bilel

Date

2024-10-28

4.1.2 Function Documentation

4.1.2.1 gpio_init()

```
void gpio_init (
            void  )
```

Initializes the GPIO port D.

This function configures GPIO port D to enable the clock and set pins D12, D13, D14, and D15 as output. The initialization steps include:

1. Enabling the clock for GPIOD by setting the appropriate bit in the RCC_AHB1ENR register.
2. Configuring the mode of the specified pins to output by clearing the relevant bits in the GPIOD_MODER register and setting them to output mode.

The output mode allows these pins to drive external devices or LEDs.

Parameters

None	

Returns

None

4.1.2.2 toggle_gpio_pin()

```
void toggle_gpio_pin (
            uint32_t pin )
```

Toggles the state of the specified GPIO pin.

This function toggles the output state of a specified pin on the GPIOD port by performing an XOR operation on the Output Data Register (ODR).

Parameters

pin	The GPIO pin to toggle. Typically defined as a bitmask corresponding to the desired pin (e.g., `GPIO_PIN_D12`).

- Uses XOR to toggle the specified pin by modifying the corresponding bit in the GPIOD Output Data Register (GPIOD_ODR).
- This function assumes that the specified pin is already configured as an output.

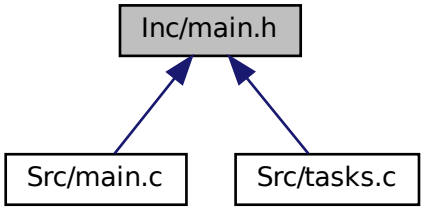
Returns

None

4.2 Inc/main.h File Reference

Header file for the Embedded Scheduler Project.

This graph shows which files directly or indirectly include this file:



Classes

- struct TaskControlBlock

  Represents a task in the task scheduler.

Macros

- #define **TASK_STACK_SIZE** 1024U
- #define **SCHEDULER_STACK_SIZE** 1024U
- #define **RAM_START_ADDRESS** 0x20000000U
- #define **RAM_TOTAL_SIZE** (128 ∗ 1024)
- #define **TASK1_STACK_START** (RAM_START_ADDRESS + RAM_TOTAL_SIZE)
- #define **TASK2_STACK_START** (TASK1_STACK_START - TASK_STACK_SIZE)
- #define **TASK3_STACK_START** (TASK2_STACK_START - TASK_STACK_SIZE)
- #define **TASK4_STACK_START** (TASK3_STACK_START - TASK_STACK_SIZE)
- #define **IDLE_STACK_START** (TASK4_STACK_START - TASK_STACK_SIZE)
- #define **SCHEDULER_STACK_START** (IDLE_STACK_START - SCHEDULER_STACK_SIZE)
- #define **TOTAL_TASKS** 5
- #define **SYSTEM_TICK_RATE_HZ** 1U
- #define **HSI_CLOCK_FREQUENCY_HZ** 16000000U
- #define **xPSR** 0x01000000U
- #define **RUNNING** 0x1
- #define **BLOCKED** 0x0

Functions

- void enable_faults (void)

  Enables system fault handling.
- void systick_T_init (uint32_t)

  Initializes the SysTick timer to generate an interrupt at a specified interval.
- __attribute__ ((naked)) void switch_sp_to_psp(void)

  Switches the stack pointer to the Process Stack Pointer (PSP).
- uint32_t get_psp_value (void)

  Retrieves the current task's Process Stack Pointer (PSP) value.
- void set_psp_value (uint32_t task_psp)

Sets the Process Stack Pointer (PSP) value for the current task.

- void update_next_task (void)

  Updates the current task to the next runnable task.

- void increment_tick (void)

  Increments the global tick count.

- void trig_pendsv (void)

  Triggers a PendSV interrupt.

- void check_blocked_tasks (void)

  Checks the state of all tasks and updates blocked tasks.

4.2.1 Detailed Description

Header file for the Embedded Scheduler Project.

This file contains function prototypes and macro definitions for the main module of the embedded scheduler.

Author

Bilel

Date

2024-10-28

4.2.2 Function Documentation

4.2.2.1 __attribute__()

```
__attribute__ (
             (naked)  )
```

Switches the stack pointer to the Process Stack Pointer (PSP).

This function initializes the Process Stack Pointer (PSP) by calling an external function `get_psp_value` to retrieve the PSP base address. It then updates the stack pointer (SP) to point to PSP by configuring the CONTROL register.

- The function is declared with `__attribute__((naked))` to omit the typical function prologue and epilogue, allowing for direct manipulation of the stack.
- The Link Register (LR) is saved and restored around the call to `get_psp_value`.
- `MSR PSP, R0` is used to set PSP to the value returned by `get_psp_value`.
- Finally, CONTROL register bit 1 is set to switch SP to PSP.

Note

This function should be called only in the privileged mode to ensure proper access to the CONTROL register.

Returns

None

4.2.2.2 check_blocked_tasks()

```
void check_blocked_tasks (
            void  )
```

Checks the state of all tasks and updates blocked tasks.

This function iterates through all tasks and checks if they are in the BLOCKED state. If a task's remaining_ticks matches the current tick count (g_tick_count), the task's state is updated to RUNNING, indicating that it is ready to execute again.

This function is typically called during the task scheduling process to determine which blocked tasks can be unblocked.

Parameters

None	

Returns

None

4.2.2.3 enable_faults()

```
void enable_faults (
            void  )
```

Enables system fault handling.

This function configures the System Handler Control and State Register (SHCSR) to enable memory management, bus, and usage fault exceptions. It modifies the appropriate bits in the SHCSR to activate fault handling for the specified fault types, allowing the system to respond to faults as needed.

Note

This function should be called during system initialization to ensure fault handling is enabled before any tasks are executed.

Returns

None

4.2.2.4 get_psp_value()

```
uint32_t get_psp_value (
            void  )
```

Retrieves the current task's Process Stack Pointer (PSP) value.

This function returns the stack pointer value of the currently running task by accessing the `stack_pointer` field of the `tasks` array at the index of `c_task`.

Parameters

None	

Returns

The current task's stack pointer (PSP) value as a 32-bit unsigned integer.

- This function is useful for managing context switching, as it provides access to the PSP of the current task.
- It is typically called in the context of low-level task management routines where stack manipulation is necessary.

4.2.2.5 increment_tick()

```
void increment_tick (
             void  )
```

Increments the global tick count.

This function increments the global tick counter `g_tick_count` by one, which is typically used for tracking elapsed time in the system.

Parameters

None	

Returns

None

4.2.2.6 set_psp_value()

```
void set_psp_value (
             uint32_t task_psp )
```

Sets the Process Stack Pointer (PSP) value for the current task.

This function updates the `stack_pointer` field of the currently running task (indexed by `c_task`) with the specified `task_psp` value.

Parameters

task_psp	The new stack pointer value to set for the current task.

Returns

None

- Used to save the stack pointer for the active task during context switching.
- This function ensures that each task maintains its own stack pointer value, allowing safe and independent task stack management.

4.2.2.7 systick_T_init()

```
void systick_T_init (
            uint32_t tick )
```

Initializes the SysTick timer to generate an interrupt at a specified interval.

This function configures the SysTick timer to generate an interrupt every `tick` microseconds by setting up the reload value and enabling the counter. The SysTick counter is configured to use the processor clock source and enable the SysTick exception.

Parameters

tick	Specifies the desired tick interval in microseconds. The reload value is calculated based on the HSI clock frequency (e.g., 16 MHz) and the specified interval.

- The reload value is calculated as `(HSI_CLOCK_FREQUENCY_HZ / tick) - 1`.
- The Reload Value Register (SYST_RVR) is cleared for the lower 24 bits, then the calculated reload value is loaded.
- The Control and Status Register (SYST_CSR) is configured to enable the counter, the clock source, and the SysTick exception.

Returns

None

4.2.2.8 trig_pendsv()

```
void trig_pendsv (
            void  )
```

Triggers a PendSV interrupt.

This function sets the PendSV (Pendable Service Interrupt) in the Interrupt Control and Status Register (ICSR) by writing to bit 28. This is typically used in a context-switching mechanism to request a context switch between tasks.

Parameters

None	

Returns

None

4.2.2.9 update_next_task()

```
void update_next_task (
            void  )
```

Updates the current task to the next runnable task.

This function increments the current task index (`c_task`) to point to the next task in a circular manner, skipping any tasks that are in a blocked state. If no tasks are in a runnable state, it defaults to the idle task (index 0).

- The function cycles through all tasks by incrementing the `c_task` index within the bounds of `TOTAL_TASKS`.
- If a running task is found, it breaks out of the loop to set `c_task` to the next runnable task. Otherwise, it falls back to the idle task.

Note

The idle task (task index 0) is chosen only when no other tasks are runnable.

Parameters

None	

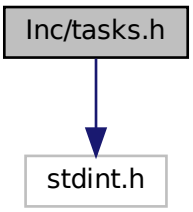
Returns

None

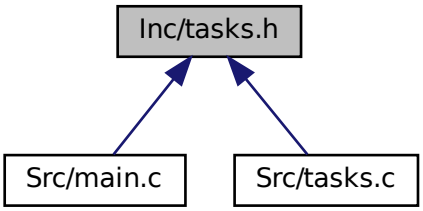
4.3 Inc/tasks.h File Reference

Task management functions for the Embedded Scheduler Project.

```
#include <stdint.h>
```
Include dependency graph for tasks.h:



This graph shows which files directly or indirectly include this file:



Functions

- __attribute__ ((naked)) void sched_stack_init(uint32_t)

  Initializes the Main Stack Pointer (MSP) for the scheduler.
- void init_tasks_stack (void)

  Initializes the task stacks and their respective states.
- void task1_routine (void)

  Task 1 routine.
- void task2_routine (void)

  Task 2 routine.
- void task3_routine (void)

  Task 3 routine.
- void task4_routine (void)

  Task 4 routine.
- void idle_routine (void)

  Idle Task routine.
- void task_delay (uint32_t delay_tick)

  Delays the execution of the current task for a specified number of ticks.

4.3.1 Detailed Description

Task management functions for the Embedded Scheduler Project.

This file defines the TaskControlBlock structure and function prototypes for task management, including task initialization and scheduling.

Author

Bilel

Date

2024-10-28

4.3.2 Function Documentation

4.3.2.1 __attribute__()

```
__attribute__ (
             (naked)  )
```

Initializes the Main Stack Pointer (MSP) for the scheduler.

This function sets the Main Stack Pointer (MSP) to the specified stack address provided as an argument. The function is marked as naked, meaning it does not have a prologue or epilogue generated by the compiler, allowing for direct assembly instructions.

Parameters

stack_sched	The address to set as the Main Stack Pointer (MSP). This address should point to the top of the stack space allocated for the scheduler.

Returns

None

Initializes the Main Stack Pointer (MSP) for the scheduler.

This function initializes the Process Stack Pointer (PSP) by calling an external function `get_psp_value` to retrieve the PSP base address. It then updates the stack pointer (SP) to point to PSP by configuring the CONTROL register.

- The function is declared with `__attribute__((naked))` to omit the typical function prologue and epilogue, allowing for direct manipulation of the stack.
- The Link Register (LR) is saved and restored around the call to `get_psp_value`.
- `MSR PSP, R0` is used to set PSP to the value returned by `get_psp_value`.
- Finally, CONTROL register bit 1 is set to switch SP to PSP.

Note

This function should be called only in the privileged mode to ensure proper access to the CONTROL register.

Returns

None

4.3.2.2 idle_routine()

```
void idle_routine (
             void  )
```

Idle Task routine.

- This routine simulates task scheduling in a multitasking environment.

Parameters

None	

Returns

None

4.3.2.3 init_tasks_stack()

```
void init_tasks_stack (
             void  )
```

Initializes the task stacks and their respective states.

This function initializes the stack pointers for each task and sets the initial state of the tasks to RUNNING. It also prepares the stack frames for each task, setting up the necessary registers for context switching.

The function performs the following steps:

- Sets each task's state to RUNNING.
- Initializes the stack pointer for each task with the address of the allocated stack.
- Prepares the stack frame by storing:
  - xPSR (Program Status Register) to indicate the Thumb state.
  - The address of the task handler function (PC).
  - The link register (LR) value.
  - Initializes general-purpose registers (R0 to R12) to zero.

Parameters

None	

Returns

None

4.3.2.4 task1_routine()

```
void task1_routine (
             void  )
```

Task 1 routine.

- This routine simulates task scheduling in a multitasking environment.

Parameters

None	

Returns

None

4.3.2.5 task2_routine()

```
void task2_routine (
             void  )
```

Task 2 routine.

- This routine simulates task scheduling in a multitasking environment.

Parameters

None	

Returns

None

4.3.2.6 task3_routine()

```
void task3_routine (
            void  )
```

Task 3 routine.

- This routine simulates task scheduling in a multitasking environment.

Parameters

None	

Returns

None

4.3.2.7 task4_routine()

```
void task4_routine (
            void  )
```

Task 4 routine.

- This routine simulates task scheduling in a multitasking environment.

Parameters

None	

Returns

None

4.3.2.8 task_delay()

```
void task_delay (
            uint32_t delay_tick )
```

Delays the execution of the current task for a specified number of ticks.

This function updates the task state of the currently running task (excluding the idle task) to `BLOCKED` and sets a delay period. The PendSV interrupt is triggered to facilitate task switching.

Parameters

delay_tick	The number of system ticks to delay the task.

- If the current task (`c_task`) is not the idle task (task 0), the function sets `remaining_ticks` to the current global tick count plus the specified `delay_tick`.
- The `task_state` of the current task is updated to `BLOCKED`, indicating it should not run until the delay period has expired.
- `trig_pendsv()` is called to initiate a context switch, allowing another task to run.

Note

The idle task (task 0) cannot be delayed.

Returns

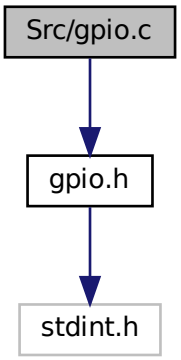
None

4.4 Src/gpio.c File Reference

GPIO control functions for the Embedded Scheduler Project.

```
#include "gpio.h"
```

Include dependency graph for gpio.c:



Functions

- void gpio_init (void)

  Initializes the GPIO port D.
- void toggle_gpio_pin (uint32_t pin)

  Toggles the state of the specified GPIO pin.
- void **delay** (volatile uint32_t count)

4.4.1 Detailed Description

GPIO control functions for the Embedded Scheduler Project.

This file implements functions to initialize and control GPIO pins.

Author

Bilel

Date

2024-10-28

4.4.2 Function Documentation

4.4.2.1 gpio_init()

```
void gpio_init (
             void  )
```

Initializes the GPIO port D.

This function configures GPIO port D to enable the clock and set pins D12, D13, D14, and D15 as output. The initialization steps include:

1. Enabling the clock for GPIOD by setting the appropriate bit in the RCC_AHB1ENR register.
2. Configuring the mode of the specified pins to output by clearing the relevant bits in the GPIOD_MODER register and setting them to output mode.

The output mode allows these pins to drive external devices or LEDs.

Parameters

None	

Returns

None

4.4.2.2 toggle_gpio_pin()

```
void toggle_gpio_pin (
             uint32_t pin )
```

Toggles the state of the specified GPIO pin.

This function toggles the output state of a specified pin on the GPIOD port by performing an XOR operation on the Output Data Register (ODR).

Parameters

pin	The GPIO pin to toggle. Typically defined as a bitmask corresponding to the desired pin (e.g., `GPIO_PIN_D12`).

- Uses XOR to toggle the specified pin by modifying the corresponding bit in the GPIOD Output Data Register (GPIOD_ODR).
- This function assumes that the specified pin is already configured as an output.

Returns

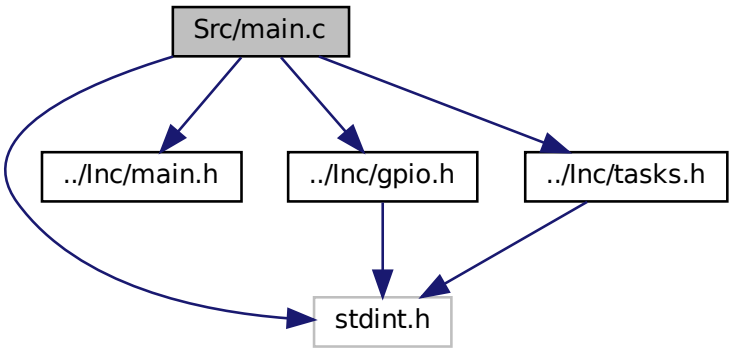
None

4.5 Src/main.c File Reference

Main entry point for the Embedded Scheduler Project.

```
#include <stdint.h>
#include "../Inc/main.h"
#include "../Inc/gpio.h"
#include "../Inc/tasks.h"
```

Include dependency graph for main.c:



Functions

- int **main** (void)
- void increment_tick (void)

  Increments the global tick count.
- void check_blocked_tasks (void)

  Checks the state of all tasks and updates blocked tasks.
- uint32_t get_psp_value (void)

  Retrieves the current task's Process Stack Pointer (PSP) value.
- void set_psp_value (uint32_t task_psp)

  Sets the Process Stack Pointer (PSP) value for the current task.
- void update_next_task (void)

  Updates the current task to the next runnable task.
- __attribute__ ((naked))

  Switches the stack pointer to the Process Stack Pointer (PSP).
- void systick_T_init (uint32_t tick)

  Initializes the SysTick timer to generate an interrupt at a specified interval.
- void enable_faults ()

  Enables system fault handling.
- void trig_pendsv ()

  Triggers a PendSV interrupt.
- void **SysTick_Handler** (void)
- void **HardFault_Handler** (void)
- void **MemManage_Handler** (void)
- void **BusFault_Handler** (void)
- void **UsageFault_Handler** (void)

Variables

- TaskControlBlock **tasks** [TOTAL_TASKS]
- uint8_t **c_task** = 1
- uint32_t **g_tick_count** = 0

4.5.1 Detailed Description

Main entry point for the Embedded Scheduler Project.

This file contains the main function, initializing tasks and hardware configurations required for the embedded scheduler. It sets up system peripherals, initializes the task scheduler, and starts the main loop.

@project Embedded Scheduler Project

Author

Bilel JALOULI

Date

2024-10-28 @license MIT

4.5.2 Function Documentation

4.5.2.1 __attribute__()

```
__attribute__ (
             (naked)  )
```

Switches the stack pointer to the Process Stack Pointer (PSP).

Initializes the Main Stack Pointer (MSP) for the scheduler.

This function initializes the Process Stack Pointer (PSP) by calling an external function `get_psp_value` to retrieve the PSP base address. It then updates the stack pointer (SP) to point to PSP by configuring the CONTROL register.

- The function is declared with `__attribute__((naked))` to omit the typical function prologue and epilogue, allowing for direct manipulation of the stack.
- The Link Register (LR) is saved and restored around the call to `get_psp_value`.
- `MSR PSP, R0` is used to set PSP to the value returned by `get_psp_value`.
- Finally, CONTROL register bit 1 is set to switch SP to PSP.

Note

This function should be called only in the privileged mode to ensure proper access to the CONTROL register.

Returns

None

4.5.2.2 check_blocked_tasks()

```
void check_blocked_tasks (
             void  )
```

Checks the state of all tasks and updates blocked tasks.

This function iterates through all tasks and checks if they are in the BLOCKED state. If a task's remaining_ticks matches the current tick count (g_tick_count), the task's state is updated to RUNNING, indicating that it is ready to execute again.

This function is typically called during the task scheduling process to determine which blocked tasks can be unblocked.

Parameters

None	

Returns

None

4.5.2.3 enable_faults()

```
void enable_faults (
             void  )
```

Enables system fault handling.

This function configures the System Handler Control and State Register (SHCSR) to enable memory management, bus, and usage fault exceptions. It modifies the appropriate bits in the SHCSR to activate fault handling for the specified fault types, allowing the system to respond to faults as needed.

Note

This function should be called during system initialization to ensure fault handling is enabled before any tasks are executed.

Returns

None

4.5.2.4 get_psp_value()

```
uint32_t get_psp_value (
             void  )
```

Retrieves the current task's Process Stack Pointer (PSP) value.

This function returns the stack pointer value of the currently running task by accessing the `stack_pointer` field of the `tasks` array at the index of `c_task`.

Parameters

None	

Returns

The current task's stack pointer (PSP) value as a 32-bit unsigned integer.

- This function is useful for managing context switching, as it provides access to the PSP of the current task.
- It is typically called in the context of low-level task management routines where stack manipulation is necessary.

4.5.2.5 increment_tick()

```
void increment_tick (
             void  )
```

Increments the global tick count.

This function increments the global tick counter `g_tick_count` by one, which is typically used for tracking elapsed time in the system.

Parameters

None	

Returns

None

4.5.2.6 set_psp_value()

```
void set_psp_value (
             uint32_t task_psp )
```

Sets the Process Stack Pointer (PSP) value for the current task.

This function updates the `stack_pointer` field of the currently running task (indexed by `c_task`) with the specified `task_psp` value.

Parameters

task_psp	The new stack pointer value to set for the current task.

Returns

None

- Used to save the stack pointer for the active task during context switching.
- This function ensures that each task maintains its own stack pointer value, allowing safe and independent task stack management.

4.5.2.7 systick_T_init()

```
void systick_T_init (
             uint32_t tick )
```

Initializes the SysTick timer to generate an interrupt at a specified interval.

This function configures the SysTick timer to generate an interrupt every `tick` microseconds by setting up the reload value and enabling the counter. The SysTick counter is configured to use the processor clock source and enable the SysTick exception.

Parameters

tick	Specifies the desired tick interval in microseconds. The reload value is calculated based on the HSI clock frequency (e.g., 16 MHz) and the specified interval.

- The reload value is calculated as `(HSI_CLOCK_FREQUENCY_HZ / tick) - 1`.
- The Reload Value Register (SYST_RVR) is cleared for the lower 24 bits, then the calculated reload value is loaded.
- The Control and Status Register (SYST_CSR) is configured to enable the counter, the clock source, and the SysTick exception.

Returns

None

4.5.2.8 trig_pendsv()

```
void trig_pendsv (
             void  )
```

Triggers a PendSV interrupt.

This function sets the PendSV (Pendable Service Interrupt) in the Interrupt Control and Status Register (ICSR) by writing to bit 28. This is typically used in a context-switching mechanism to request a context switch between tasks.

Parameters

None	

Returns

None

4.5.2.9 update_next_task()

```
void update_next_task (
             void  )
```

Updates the current task to the next runnable task.

This function increments the current task index (`c_task`) to point to the next task in a circular manner, skipping any tasks that are in a blocked state. If no tasks are in a runnable state, it defaults to the idle task (index 0).

- The function cycles through all tasks by incrementing the `c_task` index within the bounds of `TOTAL_TASKS`.
- If a running task is found, it breaks out of the loop to set `c_task` to the next runnable task. Otherwise, it falls back to the idle task.

Note

The idle task (task index 0) is chosen only when no other tasks are runnable.

Parameters

None	

Returns

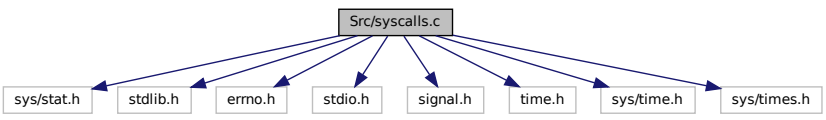
None

4.6 Src/syscalls.c File Reference

STM32CubeIDE Minimal System calls file.

```
#include <sys/stat.h>
#include <stdlib.h>
#include <errno.h>
#include <stdio.h>
#include <signal.h>
#include <time.h>
#include <sys/time.h>
#include <sys/times.h>
```

Include dependency graph for syscalls.c:



Functions

- int **__io_putchar** (int ch) __attribute__((weak))
- int **__io_getchar** (void)
- void **initialise_monitor_handles** ()
- int **_getpid** (void)
- int **_kill** (int pid, int sig)
- void **_exit** (int status)
- **__attribute__** ((weak))
- int **_close** (int file)
- int **_fstat** (int file, struct stat *st)
- int **_isatty** (int file)
- int **_lseek** (int file, int ptr, int dir)
- int **_open** (char *path, int flags,...)
- int **_wait** (int *status)
- int **_unlink** (char *name)
- int **_times** (struct tms *buf)
- int **_stat** (char *file, struct stat *st)
- int **_link** (char *old, char *new)
- int **_fork** (void)
- int **_execve** (char *name, char **argv, char **env)

Variables

- char ** **environ** = __env

4.6.1 Detailed Description

STM32CubeIDE Minimal System calls file.

Author

Auto-generated by STM32CubeIDE

```
For more information about which c-functions
need which of these lowlevel functions
please consult the Newlib libc-manual
```

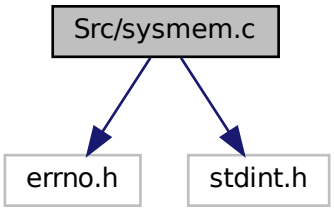
Attention

4.7 Src/sysmem.c File Reference

STM32CubeIDE System Memory calls file.

```
#include <errno.h>
#include <stdint.h>
```

Include dependency graph for sysmem.c:



Functions

- void * _sbrk (ptrdiff_t incr)

  _sbrk() allocates memory to the newlib heap and is used by malloc and others from the C library

4.7.1 Detailed Description

STM32CubeIDE System Memory calls file.

Author

Generated by STM32CubeIDE

```
For more information about which C functions
need which of these lowlevel functions
please consult the newlib libc manual
```

Attention

4.7.2 Function Documentation

4.7.2.1 _sbrk()

```
void* _sbrk (
             ptrdiff_t incr )
```

_sbrk() allocates memory to the newlib heap and is used by malloc and others from the C library

```
* ##########################################################################
* #  .data  #  .bss  #       newlib heap       #          MSP stack          #
* #         #        #                         # Reserved by _Min_Stack_Size #
* ##########################################################################
* ^-- RAM start      ^-- _end                             _estack, RAM end --^
*
```

This implementation starts allocating at the '_end' linker symbol The '_Min_Stack_Size' linker symbol reserves a memory for the MSP stack The implementation considers '_estack' linker symbol to be RAM end NOTE: If the MSP stack, at any point during execution, grows larger than the reserved size, please increase the '_Min_Stack_Size'.

Parameters

incr	Memory size

Returns

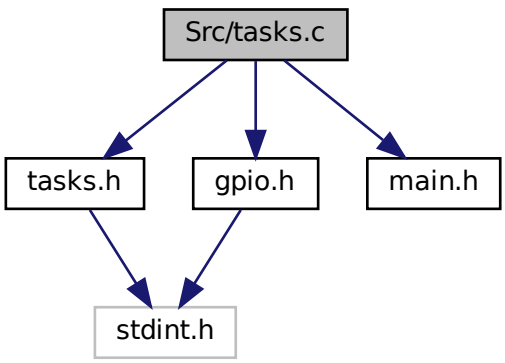
Pointer to allocated memory

4.8 Src/tasks.c File Reference

Implementation of task management functions.

```
#include "tasks.h"
#include "gpio.h"
#include "main.h"
```

Include dependency graph for tasks.c:



Functions

- void **trig_pendsv** ()
- void task1_routine (void)

  Task 1 routine.
- void task2_routine (void)

  Task 2 routine.
- void task3_routine (void)

  Task 3 routine.
- void task4_routine (void)

  Task 4 routine.
- void idle_routine (void)

  Idle Task routine.
- void init_tasks_stack (void)

  Initializes the task stacks and their respective states.
- __attribute__ ((naked))

  Switches the stack pointer to the Process Stack Pointer (PSP).
- void task_delay (uint32_t delay_tick)

  Delays the execution of the current task for a specified number of ticks.

Variables

- TaskControlBlock **tasks** [TOTAL_TASKS]
- uint8_t **c_task**
- uint32_t **g_tick_count**
- uint32_t **psp_of_tasks** [TOTAL_TASKS] = {IDLE_STACK_START ,TASK1_STACK_START , TASK2_↩ STACK_START , TASK3_STACK_START , TASK4_STACK_START}
- void(∗ **task_handlers** [TOTAL_TASKS])(void) = { idle_routine, task1_routine, task2_routine, task3_routine, task4_routine }

4.8.1 Detailed Description

Implementation of task management functions.

This file contains the implementation of task initialization, scheduling, and management for the embedded scheduler.

Author

Bilel

Date

2024-10-28

4.8.2 Function Documentation

4.8.2.1 __attribute__()

```
__attribute__ (
             (naked)  )
```

Switches the stack pointer to the Process Stack Pointer (PSP).

Initializes the Main Stack Pointer (MSP) for the scheduler.

This function initializes the Process Stack Pointer (PSP) by calling an external function `get_psp_value` to retrieve the PSP base address. It then updates the stack pointer (SP) to point to PSP by configuring the CONTROL register.

- The function is declared with `__attribute__((naked))` to omit the typical function prologue and epilogue, allowing for direct manipulation of the stack.
- The Link Register (LR) is saved and restored around the call to `get_psp_value`.
- `MSR PSP, R0` is used to set PSP to the value returned by `get_psp_value`.
- Finally, CONTROL register bit 1 is set to switch SP to PSP.

Note

This function should be called only in the privileged mode to ensure proper access to the CONTROL register.

Returns

None

4.8.2.2 idle_routine()

```
void idle_routine (
             void  )
```

Idle Task routine.

- This routine simulates task scheduling in a multitasking environment.

Parameters

None	

Returns

None

4.8.2.3 init_tasks_stack()

```
void init_tasks_stack (
             void  )
```

Initializes the task stacks and their respective states.

This function initializes the stack pointers for each task and sets the initial state of the tasks to RUNNING. It also prepares the stack frames for each task, setting up the necessary registers for context switching.

The function performs the following steps:

- Sets each task's state to RUNNING.
- Initializes the stack pointer for each task with the address of the allocated stack.
- Prepares the stack frame by storing:
  - xPSR (Program Status Register) to indicate the Thumb state.
  - The address of the task handler function (PC).
  - The link register (LR) value.
  - Initializes general-purpose registers (R0 to R12) to zero.

Parameters

None	

Returns

None

4.8.2.4 task1_routine()

```
void task1_routine (
             void  )
```

Task 1 routine.

- This routine simulates task scheduling in a multitasking environment.

Parameters

None	

Returns

None

4.8.2.5 task2_routine()

```
void task2_routine (
            void  )
```

Task 2 routine.

- This routine simulates task scheduling in a multitasking environment.

Parameters

None	

Returns

None

4.8.2.6 task3_routine()

```
void task3_routine (
            void  )
```

Task 3 routine.

- This routine simulates task scheduling in a multitasking environment.

Parameters

None	

Returns

None

4.8.2.7 task4_routine()

```
void task4_routine (
            void  )
```

Task 4 routine.

- This routine simulates task scheduling in a multitasking environment.

Parameters

None	

Returns

None

4.8.2.8 task_delay()

```
void task_delay (
            uint32_t delay_tick )
```

Delays the execution of the current task for a specified number of ticks.

This function updates the task state of the currently running task (excluding the idle task) to `BLOCKED` and sets a delay period. The PendSV interrupt is triggered to facilitate task switching.

Parameters

delay_tick	The number of system ticks to delay the task.

- If the current task (`c_task`) is not the idle task (task 0), the function sets `remaining_ticks` to the current global tick count plus the specified `delay_tick`.
- The `task_state` of the current task is updated to `BLOCKED`, indicating it should not run until the delay period has expired.
- `trig_pendsv()` is called to initiate a context switch, allowing another task to run.

Note

The idle task (task 0) cannot be delayed.

Returns

None