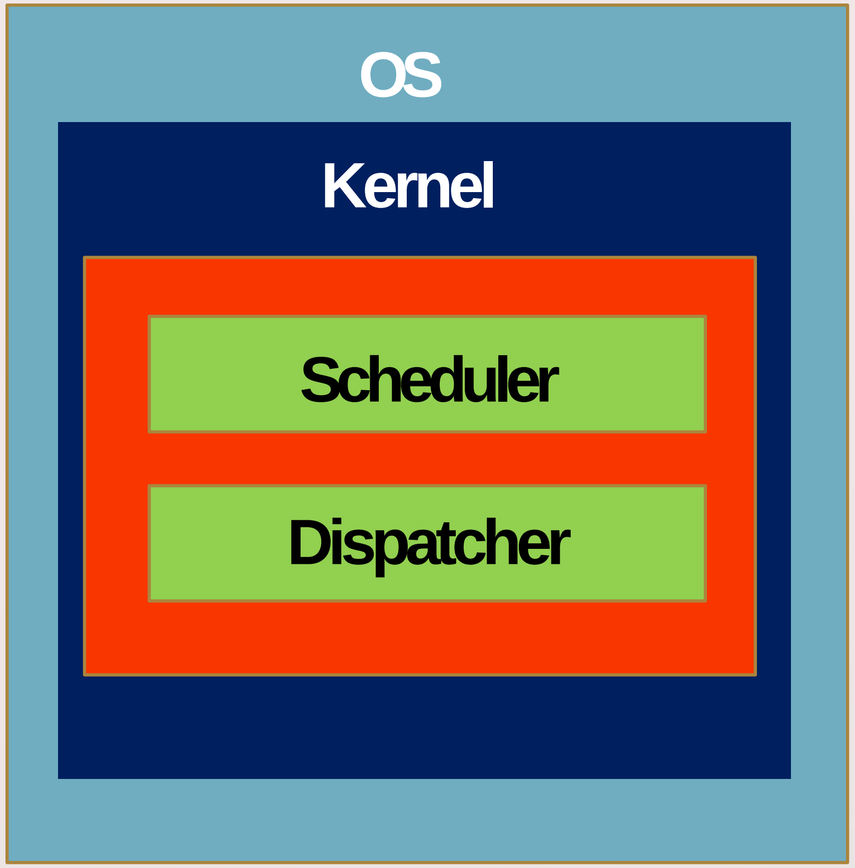
RTOS

SESSION 2

Content

- ## RTOS COMPONENTS :



RTOS COMPONENTS

- **Kernel:**
  It is the manager of tasks that organizes executions and communications between of them.

  - **component for the Kernel:**

    - **Scheduler**
      it decides each ***System Tick*** which task will be run when it is ready and blocks the task when it must wait. According to the Scheduling Algorithm used
    - **Dispatcher**
      It is responsible for executing the decision of the Scheduler, also it Creates the
      Task Control Block "TCB" of the current loaded task then does the
      context switching, then Loads the ready task to the CPU.

RTOS COMPONENTS

- **System Tick**

  RTOS real-time kernel measures time using a tick count variable. Each time the tick count is incremented the real-time kernel must check to see if it is now time to unblock or wake a task.

  A timer interrupt (the RTOS tick interrupt) increments the tick count with strict temporal accuracy - allowing the real-time kernel to measure time to a resolution of the chosen timer interrupt frequency.

RTOS COMPONENTS

- **types of Kernel:**
  - **Preemptive Kernel:**
    - ✓ This type of kernel can take a decision at any time to change the order of execution according to the scheduler that can interrupt any running task
      according to it's Algorithm.
    - ✓ The order of execution is varied.
    - ✓ ISR task is the highest priority task.
    - ✓ this kernel is commonly used because it is higher responsiveness.

RTOS COMPONENTS

- **Non-preemptive Kernel:**
  - ✓ This type is also called Co-operative kernel.
  - ✓ This type of kernel can not change the order of execution.
  - ✓ the scheduler can not interrupt any running task according to the type of Algorithm until the task requests.
  - ✓ The order of execution is fixed.
  - ✓ ISR task is not the highest priority task.
  - ✓ The tasks have to exclusively request from scheduler to leave the CPU.
  - ✓ This type of this kernel can not interrupt tasks at any time even if a higher
    priority task becomes ready until the task requests to leave CPU.
  - ✓ the responsiveness of this type of kernel is LOW.

RTOS CONSTRUCTION COMPONENTS :

- **Objects:**
  Tasks, Semaphores, Mutex, ... Etc.

- **Services:**
  create objects:
  it is used for synchronization between tasks.
  Kill Objects:
  for Kill or delete Tasks.
  Send or Receive Messages:
  it is used for communication between tasks.

RTOS CONSTRUCTION COMPONENTS :

- **Task:**

  It is a normal C function, but its job is independent of other tasks in the system. It must think that it has the CPU for itself only.

IDLE TASK

The default task that runs if there are no tasks in the Running state, its priority is the lowest
priority and it is usually an empty super loop.

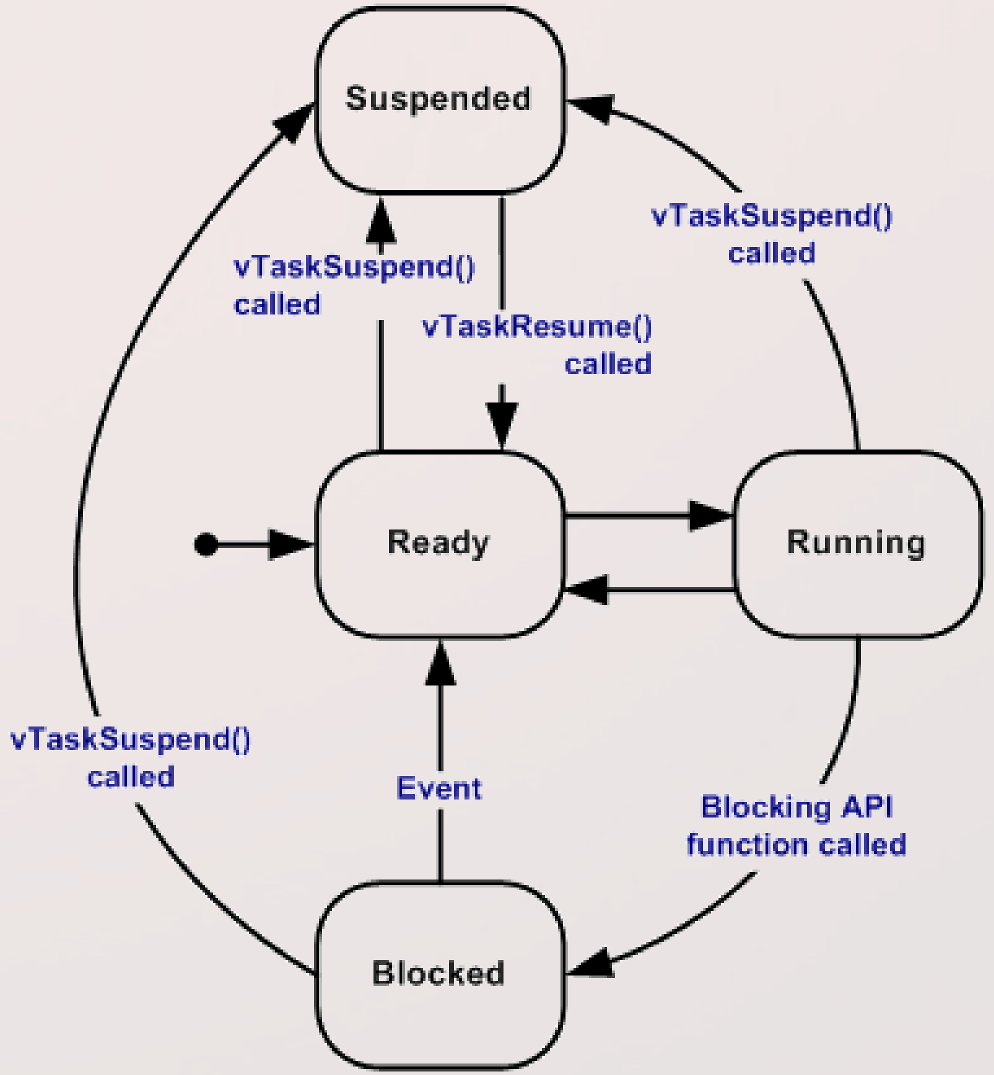
RTOS CONSTRUCTION COMPONENTS :

The software component of each task :

- **C function.**
  the task inner code that determines the functionality of this task
- **Timing Characteristics:**
  - Execution Time.
  - Periodicity.
  - Deadline.
  - Max Blockage Time.
- **Context Switching:**
  Saving the Program Counter “PC”, Process Status Register “PSR” and GPRs.
- **Storage:**
  While Context Switching, all registers must be stored in the memory, so Task Control Block “TCB” will be created to save all characteristics of task.

TASKS STATES

- Ready: task that able to be executed .
- Running: When a task is actually executing .
- Blocked: waiting for event or resource .
- Suspended: tasks that Stopped.



TASKS STATES

- **Running**

  When a task is actually executing it is said to be in the Running state. It is currently utilizing the processor. If the processor on which the RTOS is running only has a single core then there can only be one task in the Running state at any given time.

- **Ready**

  Ready tasks are those that are able to execute (they are not in the Blocked or Suspended state) but are not currently executing because a different task of equal or higher priority is already in the Running state.

- **Suspended**

  Like tasks that are in the Blocked state, tasks in the Suspended state cannot be selected to enter the Running state, but tasks in the Suspended state do not have a time out. Instead, tasks only enter or exit the Suspended state when explicitly commanded to do so through the vTaskSuspend() and xTaskResume().

TASKS STATES

•Blocked

task enter this State when currently waiting for either a temporal or external event or Resource.

For example:

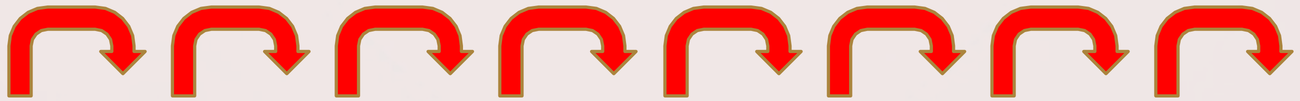
1)if a task calls vTaskDelay() it will block (be placed into the Blocked state) until the delay period has expired (*this is a temporal event*).

2)Tasks can also block waiting for a queue, semaphore, or event group. Tasks in the Blocked state normally have a 'timeout' period, after which the task will be a timeout, and be unblocked, even if the event the task was waiting for has not occurred.

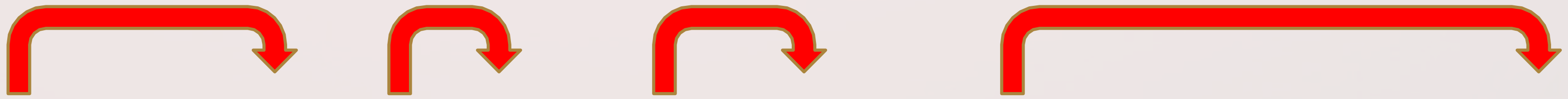
.Tasks in the Blocked state do not use any processing time and cannot be selected to enter the Running state.

Tasks Periodicity Classifications:

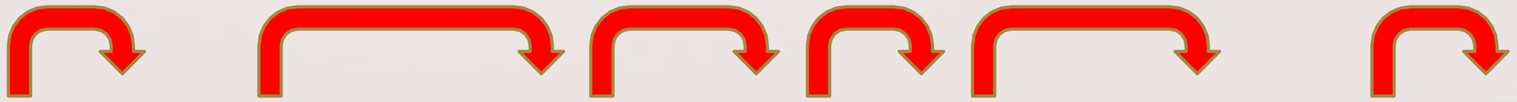
- **PERIODIC**: a task repeated by fixed Period and mustn't miss it's deadline.



- **APERIODIC**: a task arrives unexpectedly and can miss it's deadline.



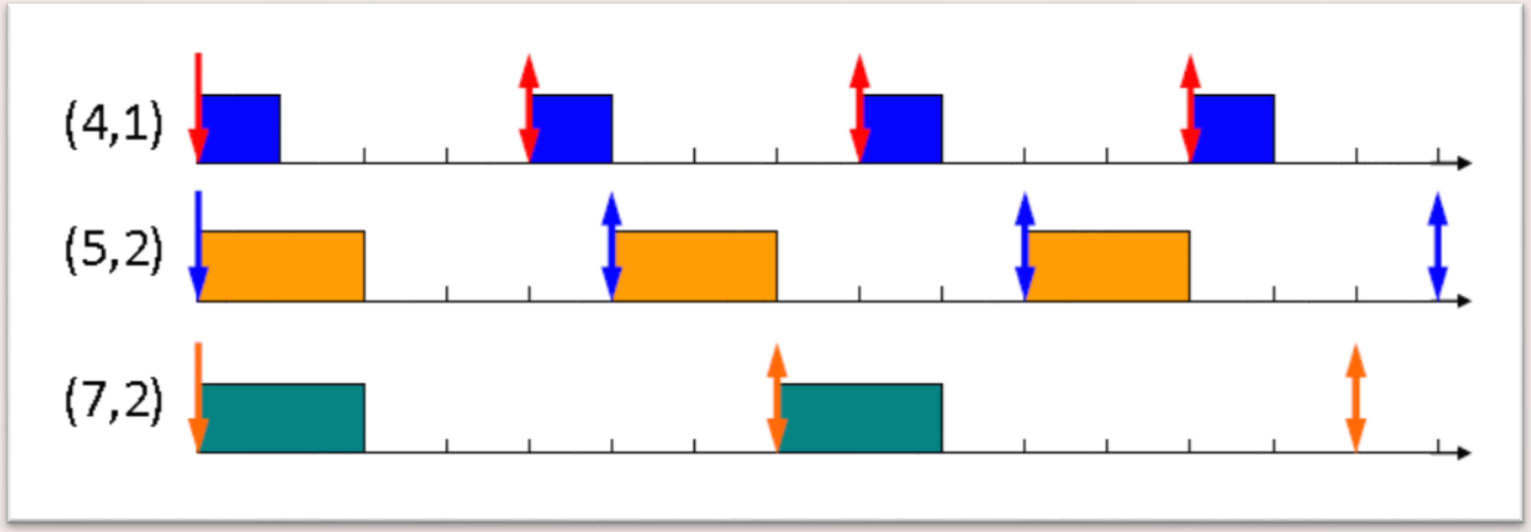
- **SPORADIC**: a task arrives unexpectedly and must meet it's deadline.



SCHEDULING

SCHEDULABILITY

Property indicating whether a set of real-time tasks at a real-time system can meet their deadlines or not.



SCHEDULING ALGORITHM:

It is the Technique used by the scheduler for deciding which task must be run, this decision is taken according to the type of Algorithm.

SOME SCHEDULER ALGORITHMS USED IN REAL-TIME SYSTEMS ARE:

1• Round-robin

2• Rate-Monotonic (RM)

3• Earliest deadline first

(EDF) 4• Fixed Priority

5• First Come, First Serve “FCFS” the task that comes first will be executed first.

6• Shortest Job First “FJT” the task that has the shortest execution time will be run first.

7• Shortest Remaining Time “SRT” is the task that has shortest remaining time runs first.

DEADLINE : the maximum latency allowed for task .

- It decide if the task is schedulable or not with other task parameters like (Priority – Periodicity- Execution time,..etc)
-If no deadline specified so, Then the deadline is assumed to be (deadline = periodicity).

- **HARD-DEADLINE**
  - **missing deadline consider failure very serious consequences may occur if the deadline is missed**
  - **Validation is essential: can all the deadlines be met, even under worst-case scenario?**
  - Deterministic **guarantees**

- **SOFT-DEADLINE**
  - **missing deadline only degrades the performance**
  - **Best effort approaches / statistical guarantees**

– PREEMPTIVE SCHEDULERS

At which the higher priority Tasks can interrupt the lower priority require more system memory in order to save the context

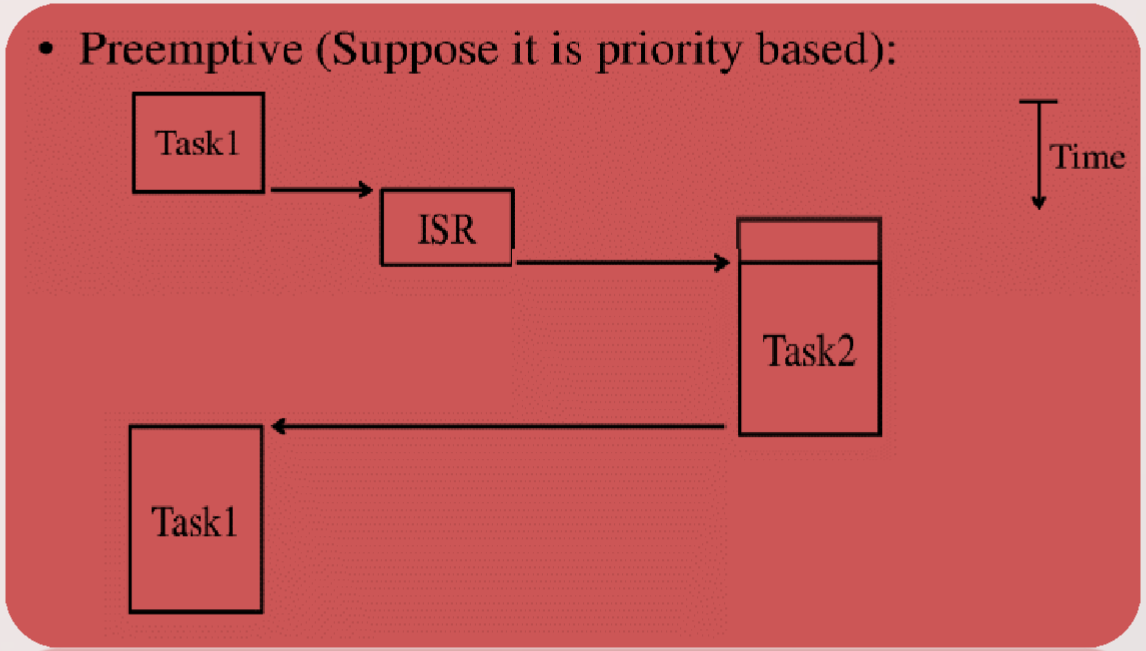
Simulate the following

Two tasks ,
preemptive,
WCET=2,
task1 P=D=3, priority =3
Task2 P=D=6 , priority =2

Result : tasks are schedulable

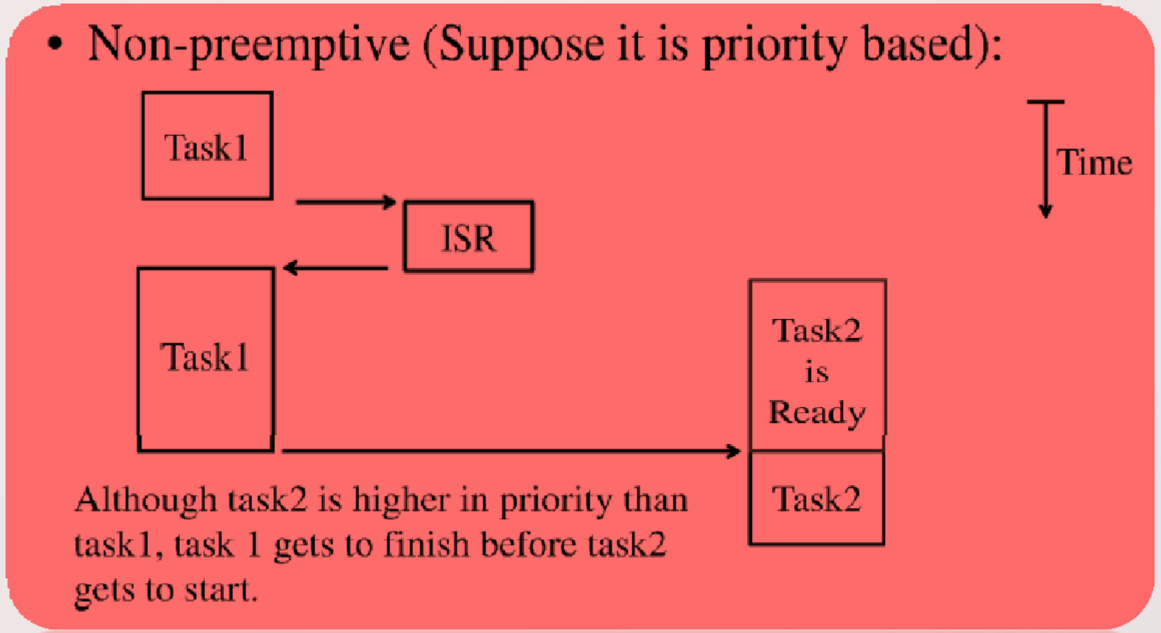
adv

good response time for high-priority tasks
tasks often meet real-time requirements



NON-PREEMPTIVE SCHEDULERS

a lower priority Task has already started the execution of the Task, and a higher priority Task is ready and wants to be executed, but it won't start its execution until the already started lower priority Task ends its execution.



Although task2 is higher in priority than task1, task 1 gets to finish before task2 gets to start.

Simulate the following

Two tasks,
fixed priority, nonpreemptive
WCET=2,
task1 P=D=3
Task2 P=D=6

Result: Task 1 misses its deadline

Disadv

Low response times for high-priority tasks
System tasks might not meet its real-time requirements and miss their Deadlines

ROUND-ROBIN SCHEDULER characteristics:

- Preemptive scheduler

- No priorities

- Tasks are organized in a queue where each task Sliced to a fixed time, it is called a
quantum, each quantum executed mutually between tasks

•Once a process is executed for a given time period, it is preempted and saved in the queue and other task executes for a given time period.

A round-robin scheduler can be easily implemented in FreeRTOS by:

- Enabling preemption
- Creating tasks with same priority
- Enabling time slicing

Adv:

- It doesn't face the issues of starvation
- All the jobs get a fair allocation of CPU
- Simple to implement

DisAdv:

- Its performance heavily depends on time quantum
- Priorities cannot be set for the tasks
- Doesn't give special priority to more important tasks
- Difficult to define quantum value

RATE-MONOTONIC SCHEDULER characteristics:

- Preemptive scheduler
- It assigns priority according to period(shorter period has a higher priority)

Adv:

- It is easy to implement
- Predictable
- Most optimal scheduling Algorithm among other fixed priority schedulers

DisAdv:

- It is very difficult to support aperiodic and sporadic tasks
- Not optimal when tasks period and deadline differ

Earliest Deadline First Scheduling characteristics:

- Preemptive scheduler
- Dynamic priorities assigned online at every scheduling point
- Priorities are assigned based on deadlines, Closer deadline gives higher priorit

ADV:

- No need to define priorities offline
- Most optimal scheduling policy among other dynamic priority schedulers
- High utilization of CPU
- Can handle periodic and aperiodic tasks

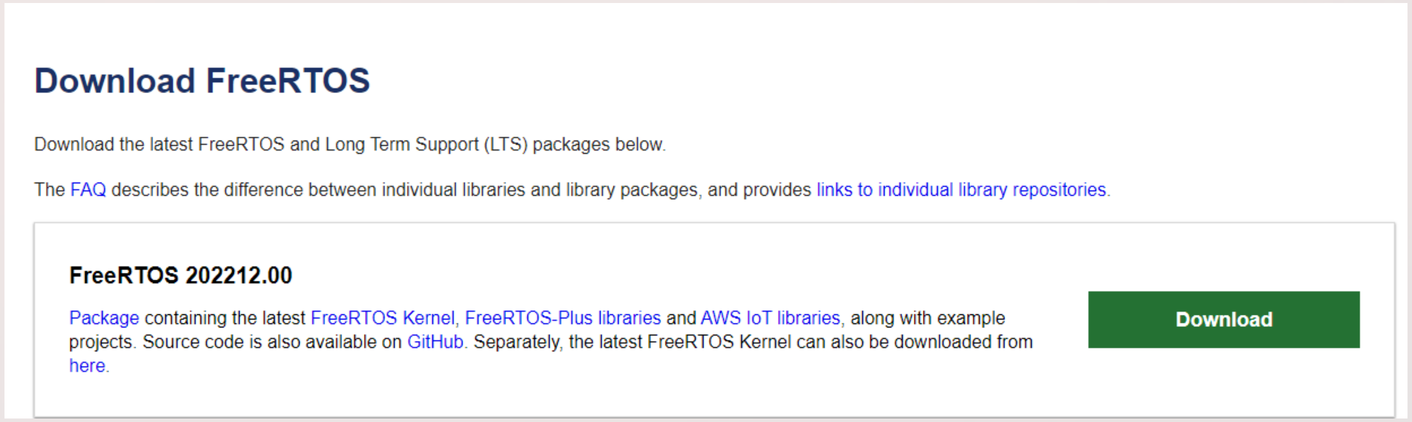
DisAdv:

- Complex in implementation and verification of schedulability
- Less predictable due to dynamic behavior

LET’S START TO EXPLORE FREERTOS….

Add Free RTOS to our program

- Download free RTOS (https://www.freertos.org/a00104.html)
- We need to customize FreeRTOS for our microcomputer


Download FreeRTOS

Download the latest FreeRTOS and Long Term Support (LTS) packages below.

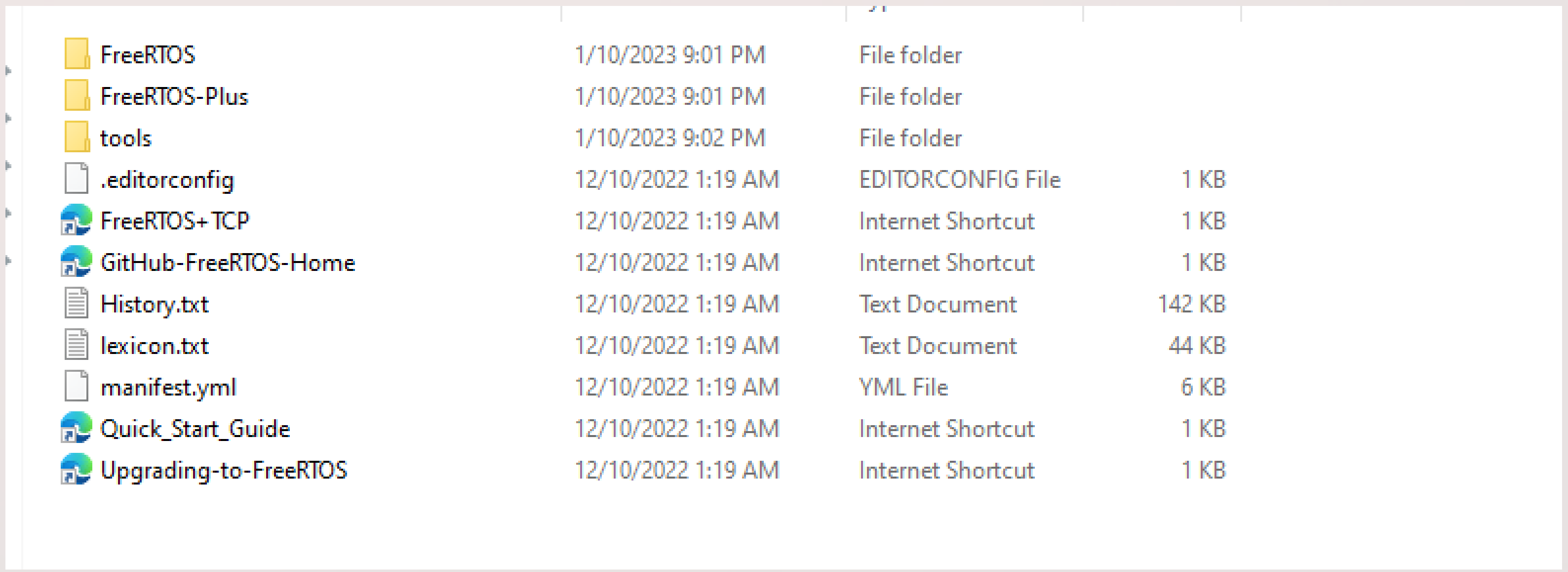
The FAQ describes the difference between individual libraries and library packages, and provides links to individual library repositories.

FreeRTOS 202212.00

Package containing the latest FreeRTOS Kernel, FreeRTOS-Plus libraries and AWS IoT libraries, along with example projects. Source code is also available on GitHub. Separately, the latest FreeRTOS Kernel can also be downloaded from here.

Add Free RTOS to our program

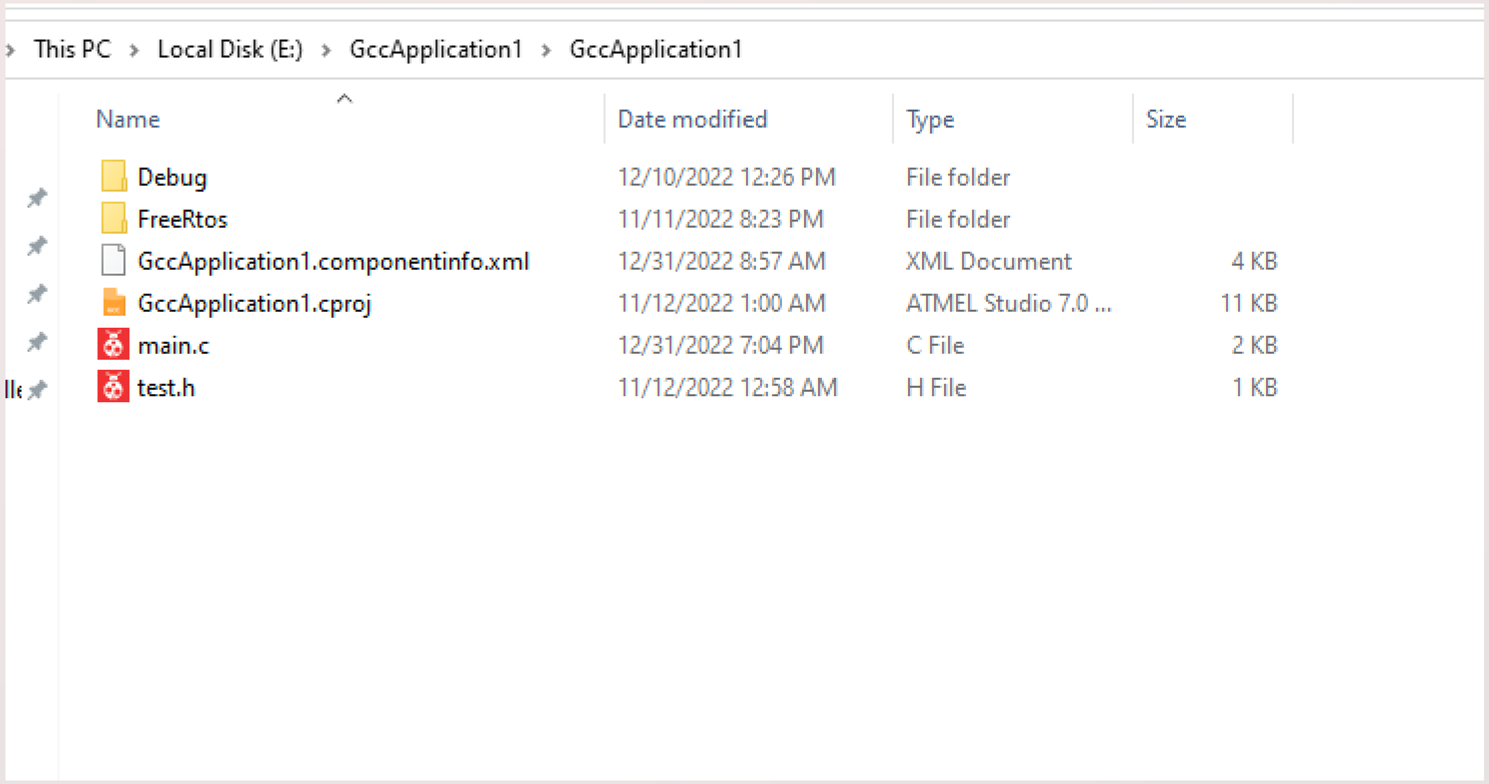
- Create a new project on Atmel studio
- Copy FreeRTOS folder



FreeRTOS	1/10/2023 9:01 PM	File folder	
FreeRTOS-Plus	1/10/2023 9:01 PM	File folder	
tools	1/10/2023 9:02 PM	File folder	
.editorconfig	12/10/2022 1:19 AM	EDITORCONFIG File	1 KB
FreeRTOS+TCP	12/10/2022 1:19 AM	Internet Shortcut	1 KB
GitHub-FreeRTOS-Home	12/10/2022 1:19 AM	Internet Shortcut	1 KB
History.txt	12/10/2022 1:19 AM	Text Document	142 KB
lexicon.txt	12/10/2022 1:19 AM	Text Document	44 KB
manifest.yml	12/10/2022 1:19 AM	YML File	6 KB
Quick_Start_Guide	12/10/2022 1:19 AM	Internet Shortcut	1 KB
Upgrading-to-FreeRTOS	12/10/2022 1:19 AM	Internet Shortcut	1 KB

Add Free RTOS to our program

- Add FreeRTOS to your project folder in the folder that contain the main.c

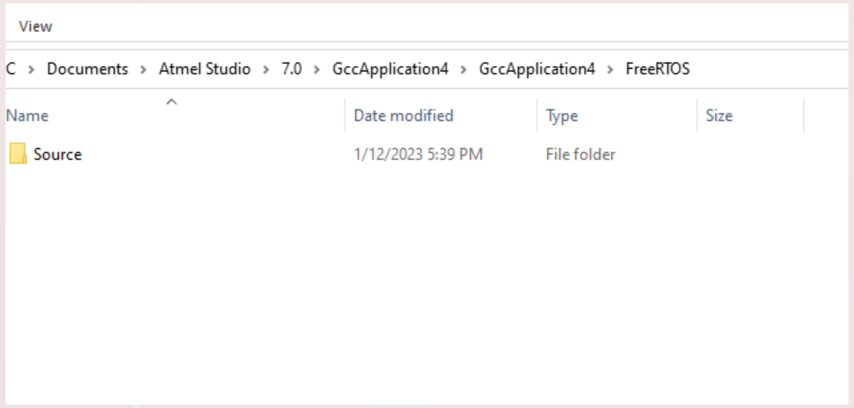


› This PC › Local Disk (E:) › GccApplication1 › GccApplication1

Name	Date modified	Type	Size
Debug	12/10/2022 12:26 PM	File folder	
FreeRtos	11/11/2022 8:23 PM	File folder	
GccApplication1.componentinfo.xml	12/31/2022 8:57 AM	XML Document	4 KB
GccApplication1.cproj	11/12/2022 1:00 AM	ATMEL Studio 7.0 ...	11 KB
main.c	12/31/2022 7:04 PM	C File	2 KB
test.h	11/12/2022 12:58 AM	H File	1 KB

Add Free RTOS to our program

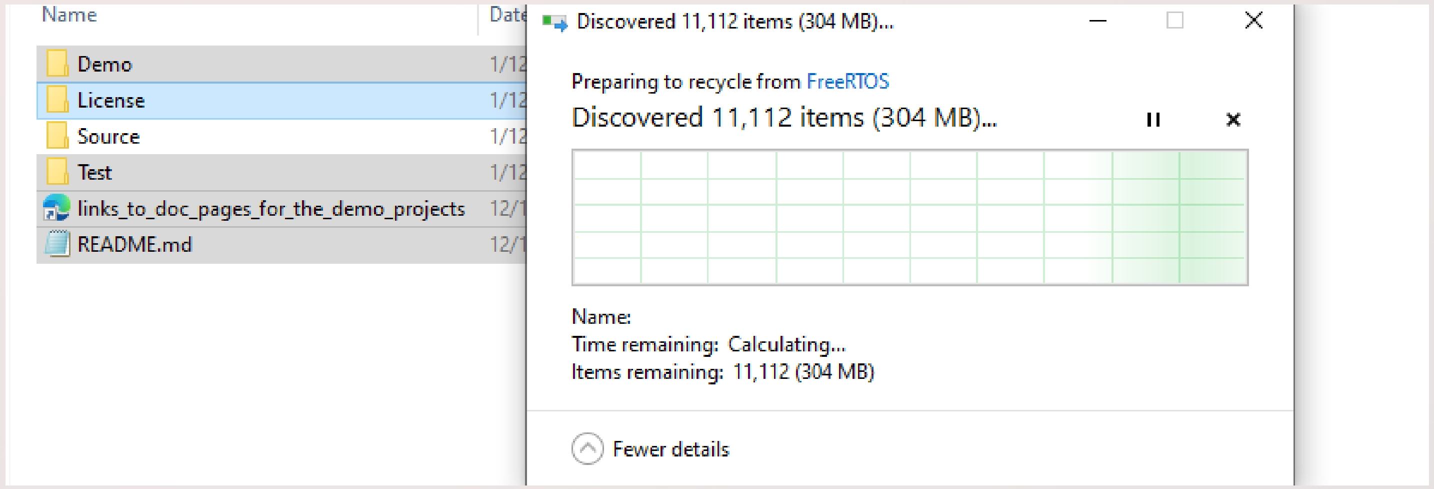
- Inside the FreeRTOS folder we just need the source folder we will delete the other folder



View

C › Documents › Atmel Studio › 7.0 › GccApplication4 › GccApplication4 › FreeRTOS

Name	Date modified	Type	Size
Source	1/12/2023 5:39 PM	File folder	



Name	Date
Demo	1/12
License	1/12
Source	1/12
Test	1/12
links_to_doc_pages_for_the_demo_projects	12/1
README.md	12/1

Discovered 11,112 items (304 MB)...

Preparing to recycle from FreeRTOS

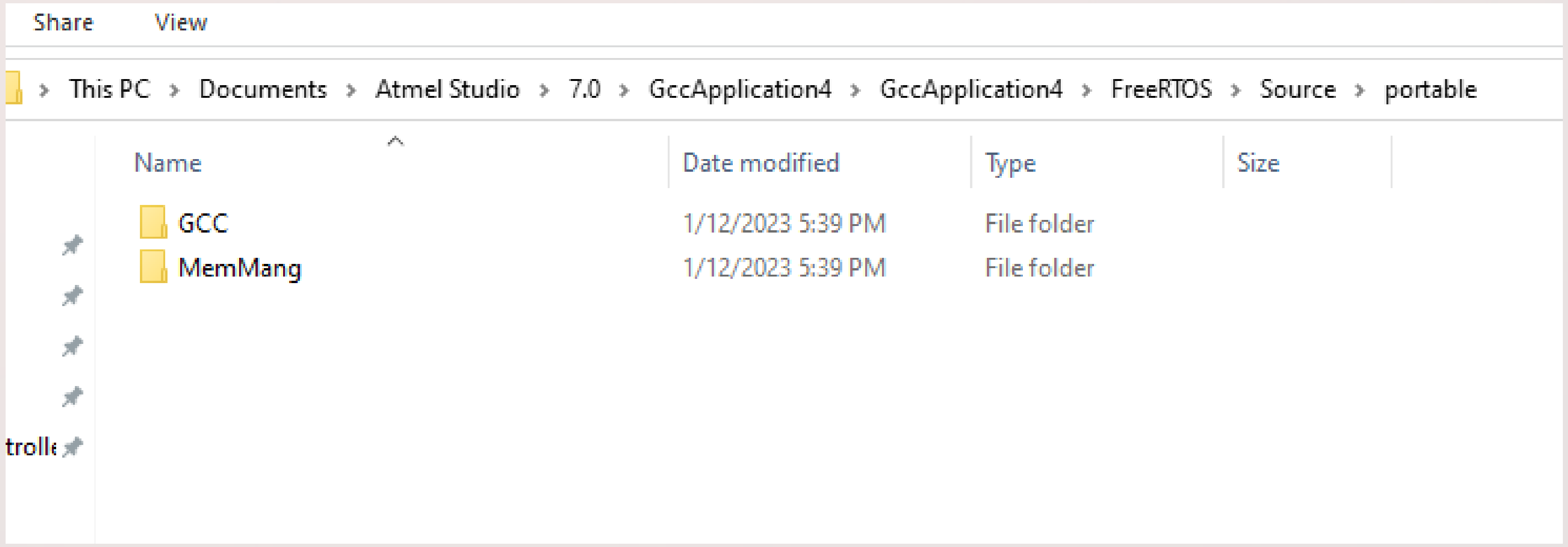
Discovered 11,112 items (304 MB)...

Name:
Time remaining: Calculating...
Items remaining: 11,112 (304 MB)

Fewer details

Add Free RTOS to our program

- Inside Source\portable we just need GCC and MemMang we will delete the other folder

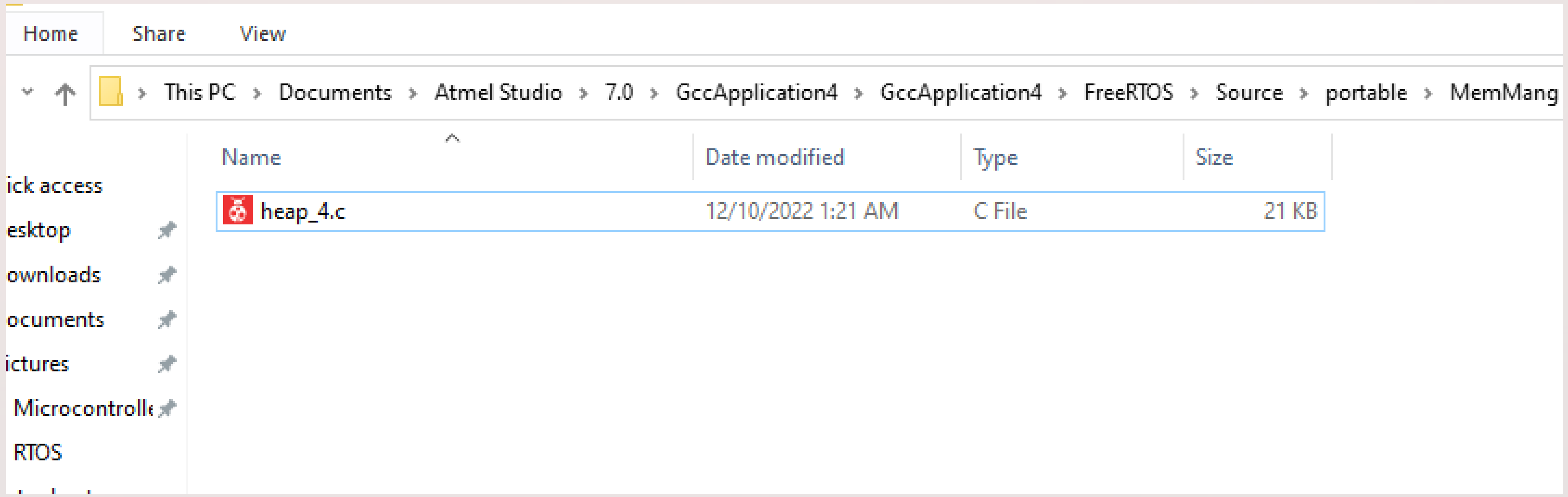


This PC > Documents > Atmel Studio > 7.0 > GccApplication4 > GccApplication4 > FreeRTOS > Source > portable

Name	Date modified	Type	Size
GCC	1/12/2023 5:39 PM	File folder	
MemMang	1/12/2023 5:39 PM	File folder	

Add Free RTOS to our program

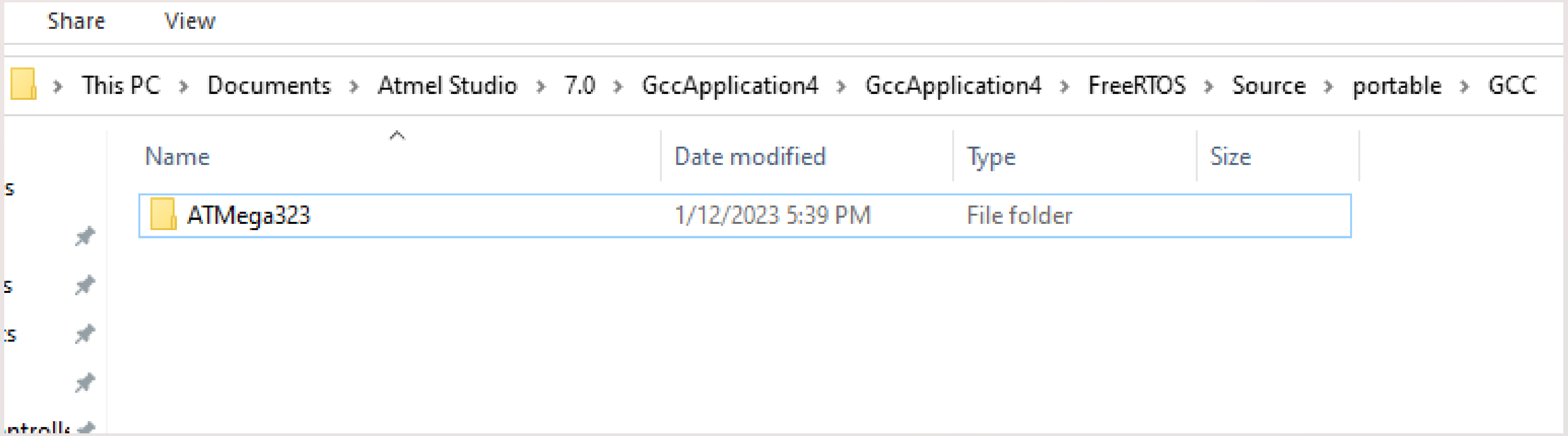
- Inside MemMang we just need one version of the heap it is preferred to keep heap4.c


Home
Share
View
This PC > Documents > Atmel Studio > 7.0 > GccApplication4 > GccApplication4 > FreeRTOS > Source > portable > MemMang
ick access
esktop
ownloads
ocuments
ictures
Microcontrolle
RTOS

Name	Date modified	Type	Size
heap_4.c	12/10/2022 1:21 AM	C File	21 KB

Add Free RTOS to our program

- Inside GCC we just need ATMega323 and will delete all other folder

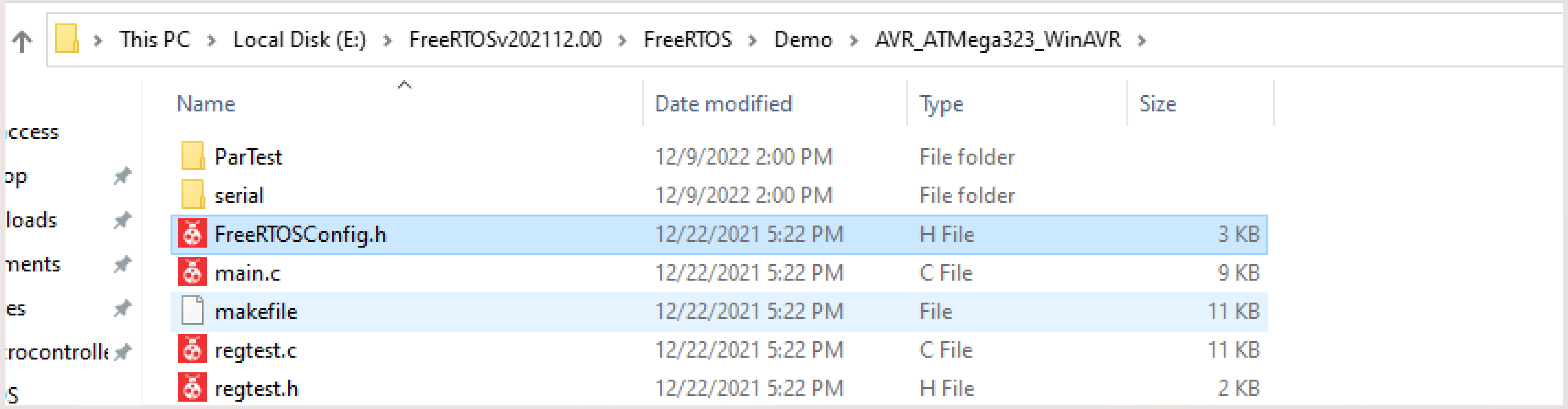


This PC › Documents › Atmel Studio › 7.0 › GccApplication4 › GccApplication4 › FreeRTOS › Source › portable › GCC

Name	Date modified	Type	Size
ATMega323	1/12/2023 5:39 PM	File folder	

Add Free RTOS to our program

- We know just need to add the config .h file in our project we will get it from inside the demo folder in FreeRTOS we go to the avr_atmega323_winAVR folder and copy the config file

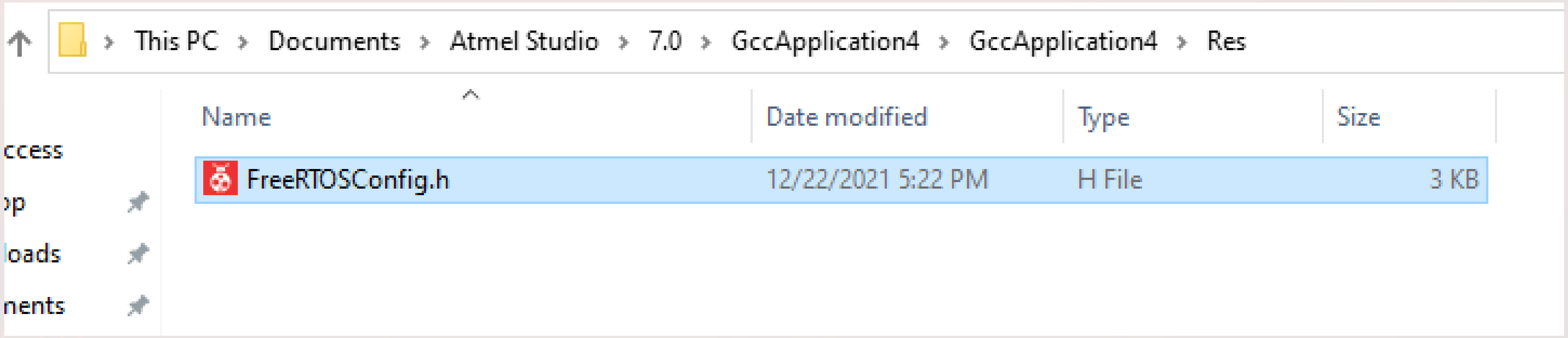


This PC › Local Disk (E:) › FreeRTOSv202112.00 › FreeRTOS › Demo › AVR_ATMega323_WinAVR ›

Name	Date modified	Type	Size
ParTest	12/9/2022 2:00 PM	File folder	
serial	12/9/2022 2:00 PM	File folder	
FreeRTOSConfig.h	12/22/2021 5:22 PM	H File	3 KB
main.c	12/22/2021 5:22 PM	C File	9 KB
makefile	12/22/2021 5:22 PM	File	11 KB
regtest.c	12/22/2021 5:22 PM	C File	11 KB
regtest.h	12/22/2021 5:22 PM	H File	2 KB

Add Free RTOS to our program

- We need to make a new folder beside the FreeRTOS folder in our project and copy the config file to it

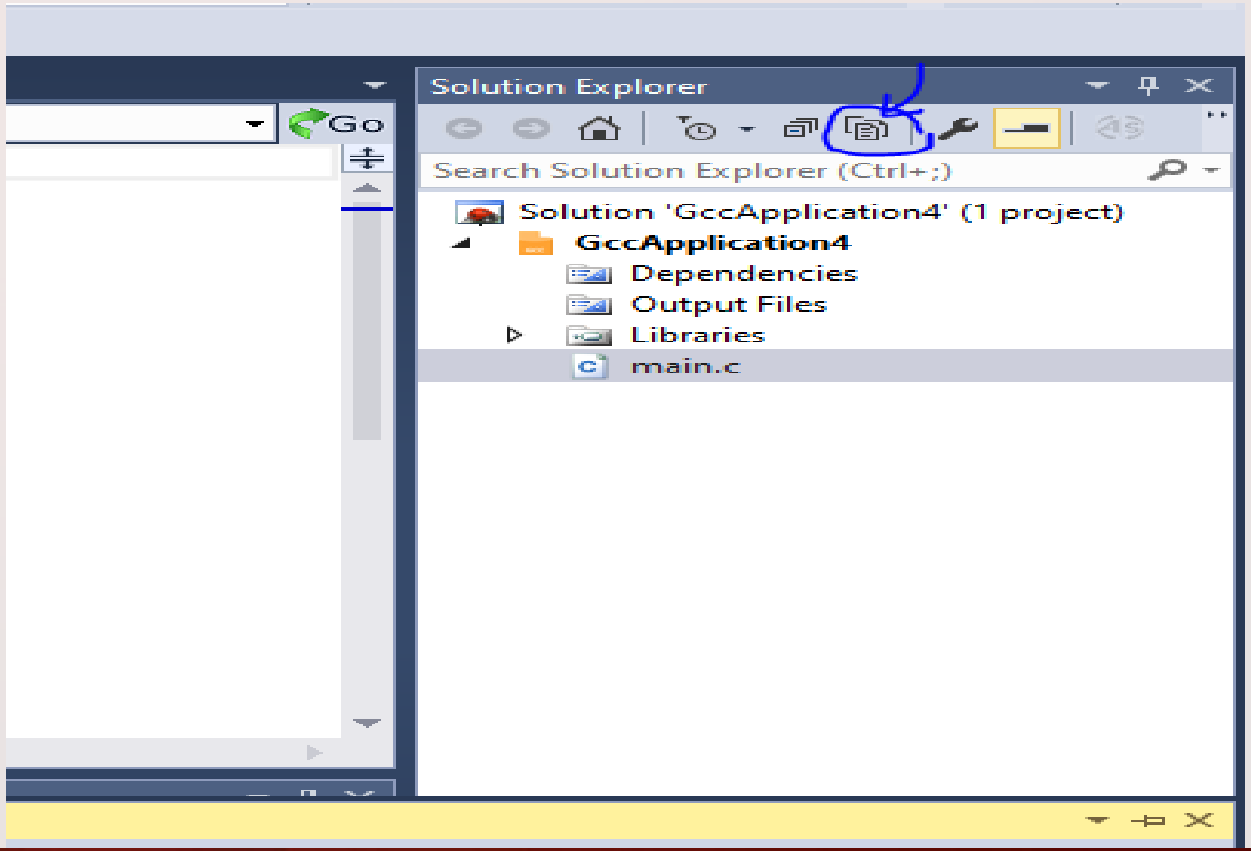


This PC › Documents › Atmel Studio › 7.0 › GccApplication4 › GccApplication4 › Res

Name	Date modified	Type	Size
FreeRTOSConfig.h	12/22/2021 5:22 PM	H File	3 KB

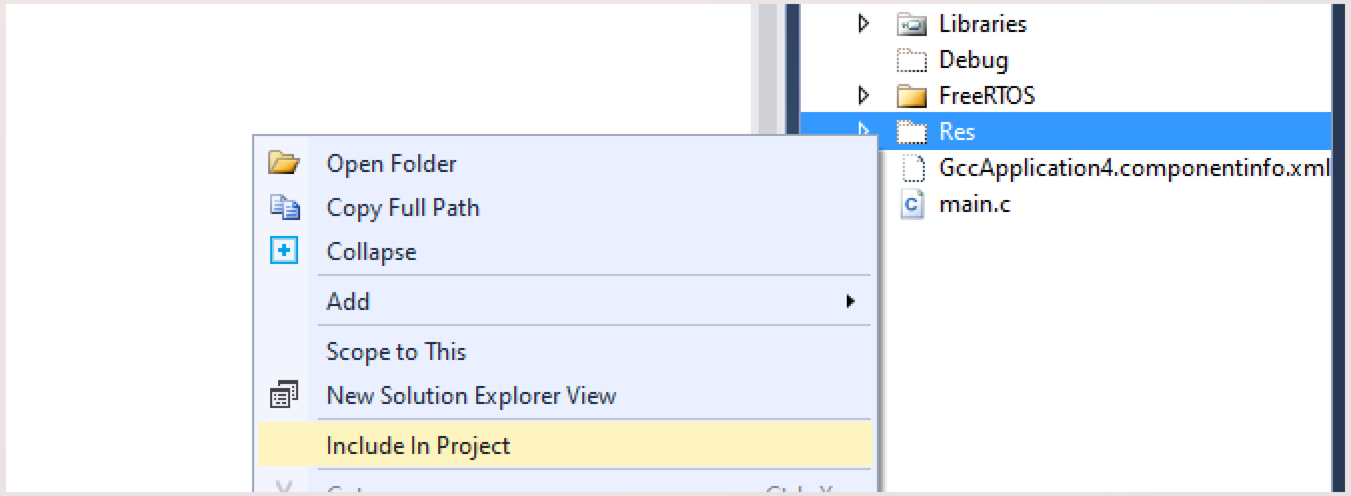
Add Free RTOS to our program

- Let’s go back to our project and open our Atmel studio
- we need to show all files



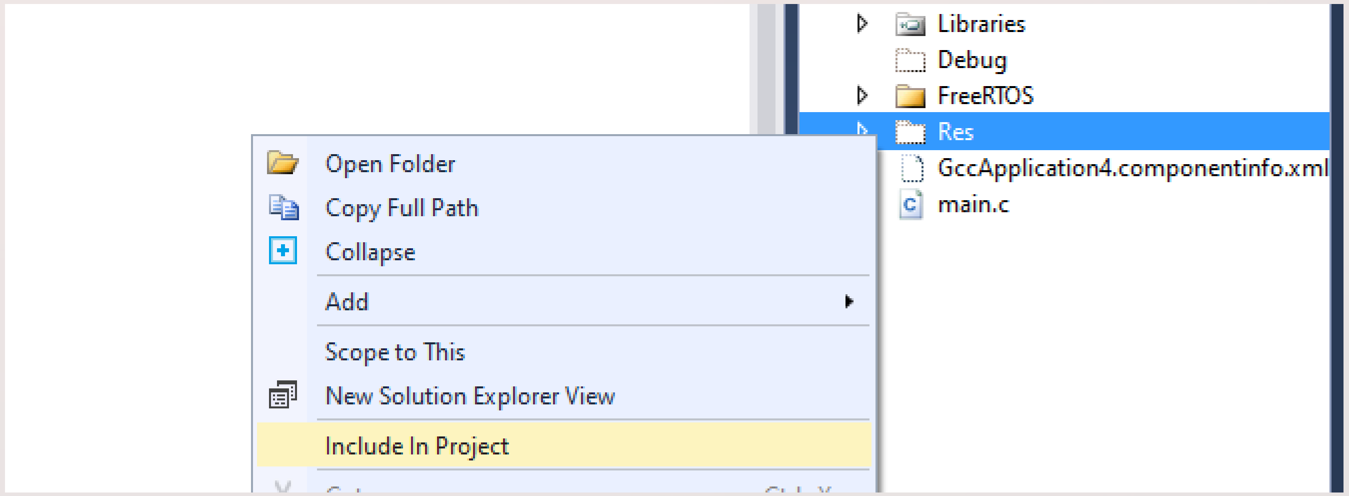
Add Free RTOS to our program

- we need to include the res inside our project that has the config file



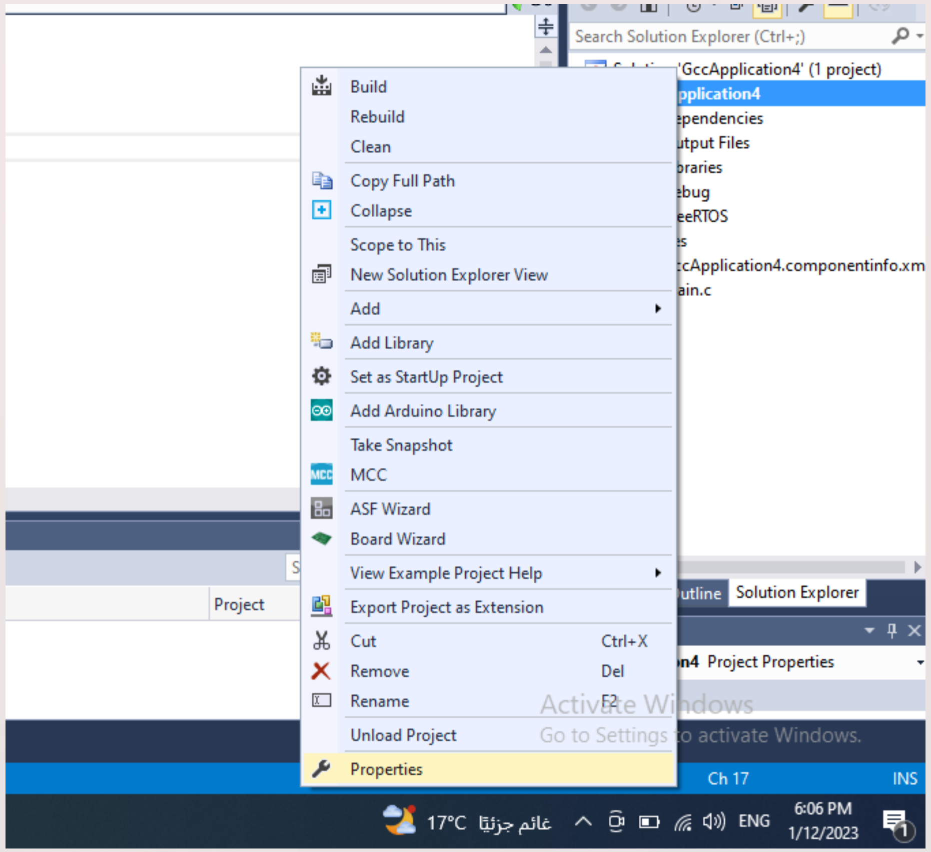
Add Free RTOS to our program

- we need to include the res inside our project that has the config file



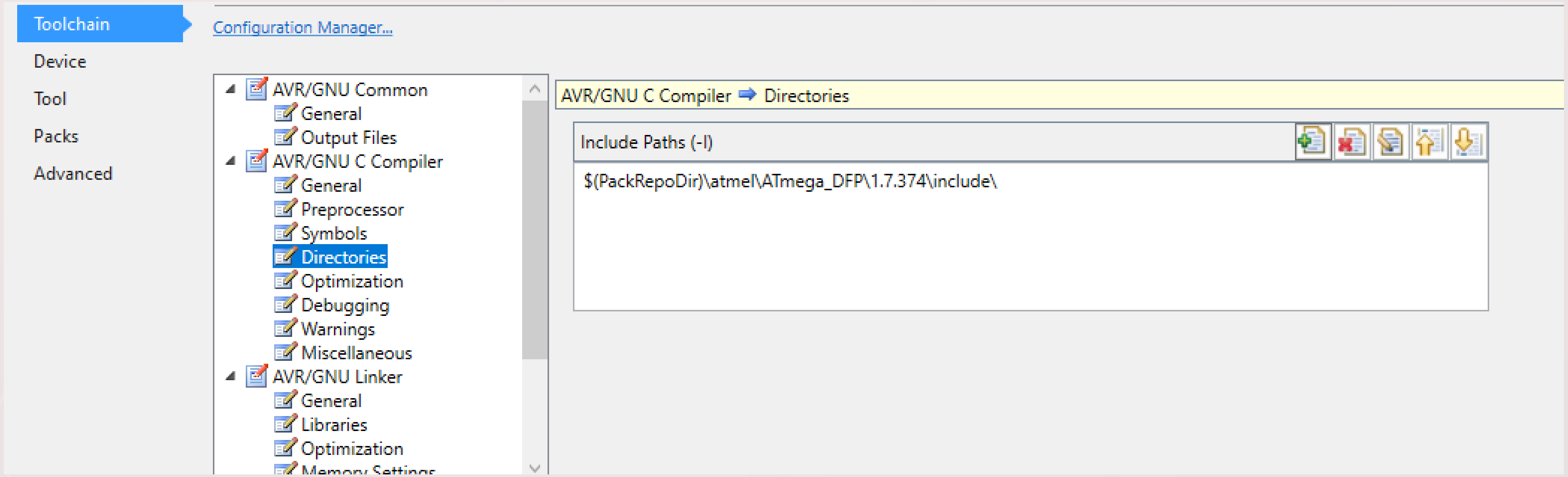
Add Free RTOS to our program

- Now we just need to add the FreeRTOS file inside the toolchain of our program we will go to properties



Add Free RTOS to our program

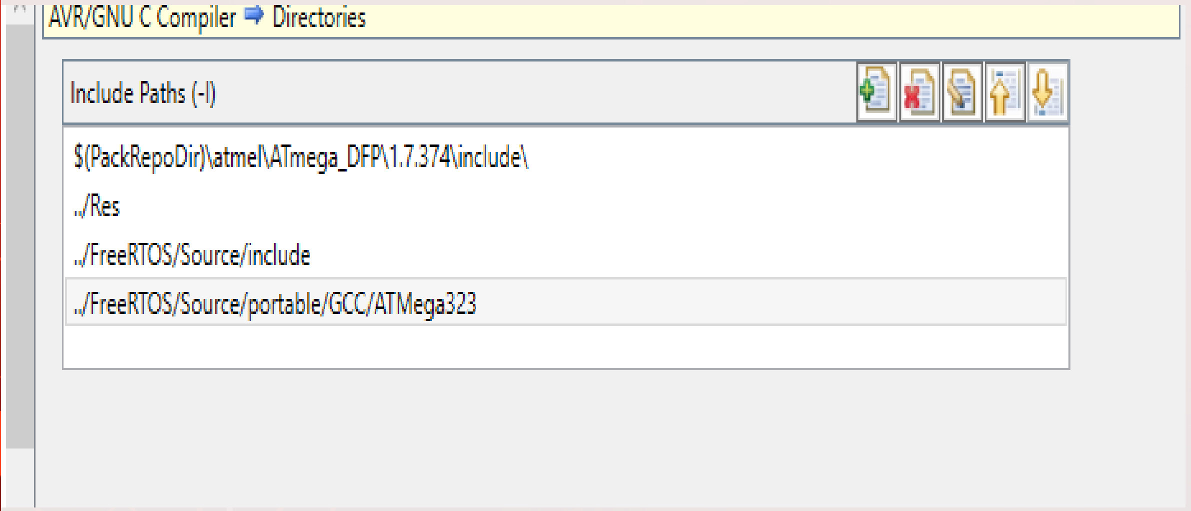
- Inside the toolchain go to the directories



$(PackRepoDir)\atmel\ATmega_DFP\1.7.374\include\

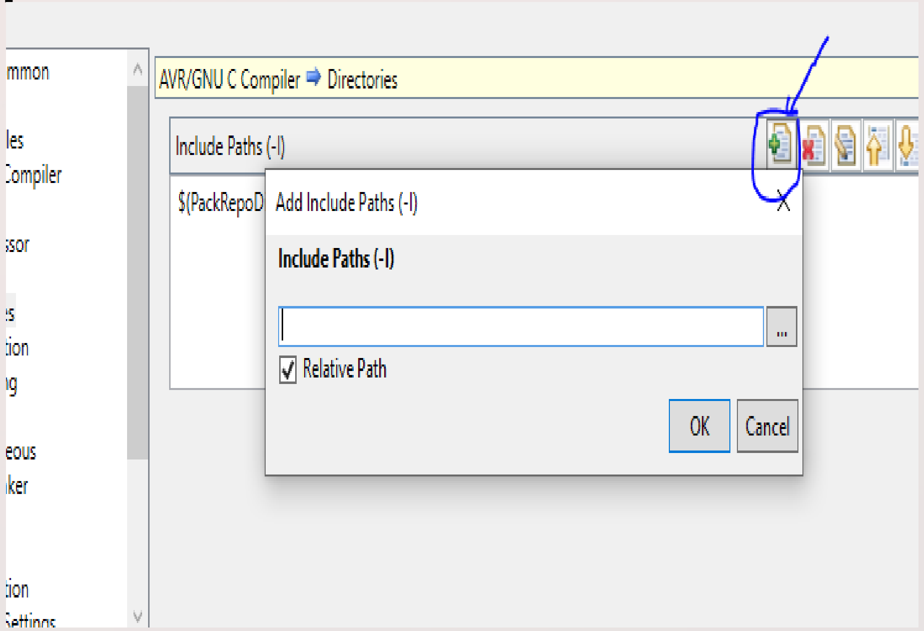
Add Free RTOS to our program

- We need to add three folder(Res that have config file, include, and ATMega323) and then try to build your project it will it run with out error


AVR/GNU C Compiler ➡ Directories

Include Paths (-I)

```
$(PackRepoDir)\atmel\ATmega_DFP\1.7.374\include\
../Res
../FreeRTOS/Source/include
../FreeRTOS/Source/portable/GCC/ATMega323
```


AVR/GNU C Compiler ➡ Directories

Include Paths (-I)

$(PackRepoD

Add Include Paths (-I)

Include Paths (-I)

☑ Relative Path

▪ TaskHandle_t *pxCreatedTask

- pxCreatedTask assigned a handler to the created task.
  At first we create a pointer of type **TaskHandle_t** that handle tasks at FreeRtos ,
  Then passing the address of this pointer to the task created to be used after that by passing it to any task control API.

```
TaskHandle_t  Task1_Handel = NULL ;
```

```
Int main() { xTaskCreate (……,……,……,……,……,&Task1_Handel); }
```

If your application has no use for the task handle, then pxCreatedTask can be set to NULL.

CREATE A TASK IN FreeRTOS

- **The prototype of this API is:**

```
BaseType_t xTaskCreate(  TaskFunction_t       pvTaskCode,
                         const char*const     pcName,
                         uint16_t             usStackDepth,
                         void *               pvParameters,
                         UBaseType_t          uxPriority,
                         TaskHandle_t *       pxCreatedTask );
```

- **There are two possible return values :**
  1. pdTRUE
  2. errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY

CREATE A TASK FreeRTOS

xTaskCreate() Get the following parameters:

- **pvTaskCode : Pointer to the inner C-function of the task**
- **pcName : A descriptive name for the task**
- **usStackDepth : The stack size to allocate for this task**
- **pvParameters : the input parameters of the C_func of the task**
- **uxPriority : The priority at which the created task will execute**
- **pxCreatedTask : Used to pass the handler of the task which is a pointer to the TCB created for this task**

task.h

```
...
typedef struct tskTaskControlBlock* TaskHandle_t;
...
```

SOME NOTES ON TASK PRAMETERS

- **usStackDepth** The size of stack allocated by word size not byte for the task created.
  For example:
  if the stack wide is 32-bits (4byte) and usStackDepth = 100 , then
  400 bytes (100 * 4) of stack will be allocated for this task .

  The stack depth multiplied by the stack width mustn’t exceed the maximum value that can be contained in a variable of type uint16_t.(i.e maximum size = 512 byte).

  - There is no easy way to determine the stack space required by a task. It is possible to calculate, but most users simply assign what they think is a suitable value.

  then they use the features provided by FreeRTOS to ensure that the space allocated is enough , and the Stack of the RAM is not being wasted.

- **void * pvParameters**:

- xTaskCreate() API accept a parameter of type pointer to void (void*) .
- The value passed to pvParameters is the values passed to the c-function of the task .

- **UBaseType_t uxPriority:**

- assigne a priority for the created task that will used by schedular to judge when to Execute
it.
- Priorities can be assigned from 0 to (configMAX_PRIORITIES - 1) ,
which is from the lowest priority to highest priority configured by
configMAX_PRIORITIES
-Passing a uxPriority value above (configMAX_PRIORITIES - 1) will result in
assigning the maximum priority configured .

vTaskDelay API :

- **- The prototype of Delay function**

```
void   vTaskDelay ( TickType_t    xTicksToDelay );
```

- **-** It is used to determine how much time that the task will be in blockage state “periodicity”.
  - vTaskDelay() assign a fixed number of tick for the called task to be placed in the Blocked state, where The task does not use any processing time

- <u>**xTicksToDelay**</u>**:**
  - The number of tick interrupts that the calling task will remain in the Blocked state before being transitioned back into the Ready state.
  For example:
  if a task called vTaskDelay(100).when the tick count was 10,000,then it would immediately enter the Blocked state,and remain in the Blocked state until the tick count reached 10,100.

FREERTOS APIs EXERCISE :

Let’s Use the APIs to design two tasks ,the first to blink led every “150” ticks, the second every “300” ticks, with a tick rate =“500 Hz”

FREERTOS APIs EXERCISE :

```
TaskHandle_t Task1_Handel = NULL ;
TaskHandle_t Task2_Handel = NULL ;

void B_LED1 (void* para)
                        {
           while (1)  {   // each task must contain infinite Loop
                DIO_TOG_PIN(PORTC,PIN2);
                vTaskDelay(150); // the task will be in blockage state about 150 ticks
           }
           }
void B_LED2 (void* para)
                             {
                while (1)  {
                     DIO_TOG_PIN(PORTC,PIN7);
                     vTaskDelay(300); // the task will be in blockage state about 300 ticks
                }
                }
```

FREERTOS APIs EXERCISE :

Main function :

```
void main (void)
{
        /*tasks created inside main*/
        xTaskCreate(B_LED1, “t1”, 250, NULL, 1, &Task1_Handel );
        xTaskCreate(B_LED2, “t2”,  250, NULL, 2, &Task2_Handel );
        ////////////////////////////////////////////////////////////
        vTaskStartScheduler();
        /*this API is used to run the scheduler and start of execution and multitasking*/
}
        /*note that you have to set the ideal hook by 0 to run this program */
```

THANK YOU