CODE-DECODE

MAIN.C

```
#include <Arduino.h>
#include "FreeRTOS.h"
#include "os_task.h"

// Define the LED pin numbers
#define LED1_PIN 2
#define LED2_PIN 3

// Define the task delay in milliseconds
#define TASK_DELAY_MS 500

// Task function to blink LED1
void vTask1(void *pvParameters)
{
    pinMode(LED1_PIN, OUTPUT);
    for (;;)
    {
        digitalWrite(LED1_PIN, HIGH);
        vTaskDelay(TASK_DELAY_MS / portTICK_PERIOD_MS);
        digitalWrite(LED1_PIN, LOW);
        vTaskDelay(TASK_DELAY_MS / portTICK_PERIOD_MS);
    }
}

// Task function to blink LED2
void vTask2(void *pvParameters)
{
    pinMode(LED2_PIN, OUTPUT);
    for (;;)
    {
        digitalWrite(LED2_PIN, HIGH);
        vTaskDelay(TASK_DELAY_MS / portTICK_PERIOD_MS);
        digitalWrite(LED2_PIN, LOW);
        vTaskDelay(TASK_DELAY_MS / portTICK_PERIOD_MS);
    }
}

void setup()
{
    // Start the serial communication
    Serial.begin(9600);

    // Create task handles
    TaskHandle_t task1_handle;
    TaskHandle_t task2_handle;

    // Create task1 to blink LED1
    xTaskCreate(vTask1, "Task1", configMINIMAL_STACK_SIZE, NULL, 1, &task1_handle);
```

```
    // Create task2 to blink LED2
    xTaskCreate(vTask2, "Task2", configMINIMAL_STACK_SIZE, NULL, 1, &task2_handle);

    // Start the scheduler
    vTaskStartScheduler();
}

void loop()
{
    // Empty loop
}
```

This code is written for the Arduino platform and uses the FreeRTOS library to create two tasks that blink two different LEDs at a specified delay. The `setup()` function initializes the serial communication, creates the task handles, and then creates two tasks, `vTask1` and `vTask2`, using the `xTaskCreate()` function. Each task is responsible for blinking a different LED connected to the corresponding pin number. Once the tasks are created, the `vTaskStartScheduler()` function is called to start the FreeRTOS scheduler and begin multitasking.

On execution, both tasks will be executed in parallel and alternate in blinking their corresponding LED. The tasks will continue running indefinitely until the system is reset or powered off.

> 💡 In the FreeRTOS library, a task handle is a pointer to a data structure that represents a particular task. It is returned by the `xTaskCreate()` function when a new task is created and can be used to reference and manipulate the task later in the program.
>
> Task handles are useful when working with FreeRTOS because they provide a way to manage and control the tasks running in the system. For example, a task handle can be used to pause, resume, or delete a task. Additionally, task handles can be used to retrieve information about the state of a task, such as its priority, stack usage, or runtime statistics.

FUNCTIONS VTASK1 AND VTASK2

`vTask1` and `vTask2` are two task functions defined in the code provided. Both tasks are responsible for blinking an LED connected to a different pin number, as specified by the `LED1_PIN` and `LED2_PIN` macros.

The `vTask1` function blinks the LED connected to `LED1_PIN`. It starts by setting the pin mode of `LED1_PIN` to `OUTPUT`. Then, the function enters an infinite loop that toggles the LED on and off at a specified delay using the `digitalWrite()` function and `vTaskDelay()` function respectively. The `vTaskDelay()` function is used to pause the task execution for a certain amount of time specified in milliseconds, using the `TASK_DELAY_MS` macro. By dividing the `TASK_DELAY_MS` value with the `portTICK_PERIOD_MS` constant, which is defined in the FreeRTOS library, the delay time is converted into the tick count that FreeRTOS uses for scheduling.

The `vTask2` function is similar to `vTask1`, but it blinks the LED connected to `LED2_PIN`. It also starts by setting the pin mode of `LED2_PIN` to `OUTPUT`, enters an infinite loop that toggles the LED on and off at a specified delay, and uses the `digitalWrite()` and `vTaskDelay()` functions in a similar manner.

Overall, both tasks are simple examples of how FreeRTOS can be used to perform multitasking on an Arduino board. The tasks are designed to run independently of each other, each blinking a different LED at a specified interval, and executing concurrently with other tasks or operations in the system.

MACROS IN C

In C programming, a macro is a fragment of code that is given a name and can be substituted for other code in the program. Macros are defined using the `#define` preprocessor directive, which associates a name with a replacement text.

Macros are used in C programming to make code more readable and maintainable. They can be used to define constants, perform simple computations, or create reusable code snippets. Macros can also be used to encapsulate complex code or conditional statements, making it easier to read and understand the program's logic.

Macros are executed by the C preprocessor, which is a program that runs before the C compiler. The preprocessor scans the source code for macro definitions and replaces all occurrences of the macro name in the code with its corresponding replacement text. This process is called macro expansion.

For example, consider the following macro definition:

```
#define PI 3.14159
```

In this example, `PI` is defined as a macro that expands to the value `3.14159`. When the preprocessor encounters the `PI` macro in the code, it replaces it with its value. So, the following line:

```
double circumference = 2 * PI * radius;
```

Loop Function

In the code provided earlier, the `loop()` function is empty, which means it does not contain any code. The `loop()` function is typically used in Arduino programming to execute repetitive tasks or to continuously monitor sensor inputs or other events.

However, in this particular code, the `loop()` function is not used because the main program logic is implemented using FreeRTOS tasks, which run concurrently with the `loop()` function. Once the FreeRTOS scheduler is started using `vTaskStartScheduler()`, it takes over the control of the program and manages the execution of the tasks.

Since the FreeRTOS scheduler runs in a loop internally, there is no need to implement a loop in the main program. Instead, the tasks are executed repeatedly in a continuous loop until the program is stopped or reset.

Therefore, in this code, the `loop()` function is not used and can be left empty.

xTaskCreate Function

```
#define xTaskCreate( pvTaskCode, pcName, usStackDepth, pvParameters, uxPriority, pxCre
atedTask ) xTaskGenericCreate( ( pvTaskCode ), ( pcName ), ( usStackDepth ), ( pvParam
eters ), ( uxPriority ), ( pxCreatedTask ), ( NULL ), ( NULL ) )

line 443 file ostask.h
```

The code you provided is a `#define` statement, which is a C preprocessor directive used to define a macro.

In this case, the macro being defined is `xTaskCreate`, which is a function-like macro that takes six arguments. The purpose of this macro is to create a new task in the FreeRTOS scheduler.

The implementation of the `xTaskCreate` macro is a call to another function called `xTaskGenericCreate`, passing the six arguments to it. The `xTaskGenericCreate` function is a generic task creation function in FreeRTOS, which takes more arguments than the `xTaskCreate` macro.

The arguments of the `xTaskCreate` macro are defined as follows:

- `pvTaskCode`: A pointer to the task function that implements the task's code.
- `pcName`: A descriptive name for the task.
- `usStackDepth`: The stack size in words (not bytes) of the task's stack space.
- `pvParameters`: A pointer to a structure or data that is passed to the task function as an argument.
- `uxPriority`: The priority level of the task, with the highest priority being 0.
- `pxCreatedTask`: A pointer to a task handle that is used to reference the created task.

The `#define` statement is used to define the `xTaskCreate` macro, which can be used throughout the program as if it were a regular function. When the macro is used in the code, the preprocessor replaces it with the corresponding call to `xTaskGenericCreate`, passing the arguments specified in the macro.

This macro allows for a more concise and readable code by simplifying the task creation process in FreeRTOS, especially when creating tasks with default parameters.

Function as Parameter in C

In C programming, functions can be passed as parameters to other functions, just like any other data type. This is known as function pointer, which is a variable that stores the memory address of a function.

To pass a function as a parameter to another function, the function pointer type must be declared as a parameter in the function signature. For example, the following function declaration takes a function pointer as a parameter:

```
void myFunction(int (*callback)(int, int));
```

`vTask1` is a function, but in C, functions are treated as first-class objects, which means they can be passed as arguments to other functions. In this case, `vTask1` is passed as the first argument to `xTaskCreate`, which expects a function pointer as its first argument.

The `vTask1` function is passed as a function pointer, which is essentially the memory address of the function. When `xTaskCreate` calls `vTask1`, it dereferences the function pointer to call the actual function.

This is a common technique used in C programming, where functions are used as arguments to other functions to provide flexibility and modularity in code design.

usStackDepth in Details

> 💡 In FreeRTOS, the `usStackDepth` parameter in the `xTaskCreate` function determines the size of the task's stack space in words, where a word is typically the size of the processor's data bus (e.g. 32 bits for a 32-bit processor). The stack space is used by the task to store its local variables, function calls, and other data during its execution.
>
> For example, if the `usStackDepth` is set to 100, then the task's stack space will be 100 words, which would typically be 400 bytes on a 32-bit processor. However, the actual size of the stack space may depend on the specific hardware and compiler used.
>
> It's important to allocate enough stack space for each task to avoid stack overflow errors, which occur when a task's stack space is insufficient to accommodate its runtime needs. A stack overflow error can cause unpredictable behavior, crashes, or even system failures.
>
> The `usStackDepth` parameter can be set according to the task's specific requirements, based on its stack usage, the complexity of its code, and the number of nested function calls it makes. A rule of thumb is to allocate at least 4-8 times the amount of stack space that the task is expected to use during its execution.
>
> For example, if a task is expected to use up to 100 bytes of stack space during its execution, then the `usStackDepth` parameter should be set to at least 32x100 = 3200. However, it's usually better to allocate more stack space than necessary, as it's easier to reduce it later than to increase it once the code is deployed.

xTaskGenericCreate Function → os_wpu_wrappers.h

```
#ifndef MPU_WRAPPERS_H
#define MPU_WRAPPERS_H

/* This file redefines API functions to be called through a wrapper macro, but
only for ports that are using the MPU. */
#ifdef portUSING_MPU_WRAPPERS

  /* MPU_WRAPPERS_INCLUDED_FROM_API_FILE will be defined when this file is
  included from queue.c or task.c to prevent it from having an effect within
  those files. */
  #ifndef MPU_WRAPPERS_INCLUDED_FROM_API_FILE

    #define xTaskGenericCreate        MPU_xTaskGenericCreate
    #define vTaskAllocateMPURegions     MPU_vTaskAllocateMPURegions
    #define vTaskDelete           MPU_vTaskDelete
    #define vTaskDelayUntil         MPU_vTaskDelayUntil
    #define vTaskDelay            MPU_vTaskDelay
    #define uxTaskPriorityGet       MPU_uxTaskPriorityGet
    #define vTaskPrioritySet        MPU_vTaskPrioritySet
    #define eTaskGetState         MPU_eTaskGetState
    #define vTaskSuspend          MPU_vTaskSuspend
    #define vTaskResume           MPU_vTaskResume
    #define vTaskSuspendAll         MPU_vTaskSuspendAll
    #define xTaskResumeAll          MPU_xTaskResumeAll
    #define xTaskGetTickCount       MPU_xTaskGetTickCount
    #define uxTaskGetNumberOfTasks      MPU_uxTaskGetNumberOfTasks
    #define vTaskList           MPU_vTaskList
    #define vTaskGetRunTimeStats      MPU_vTaskGetRunTimeStats
    #define vTaskSetApplicationTaskTag    MPU_vTaskSetApplicationTaskTag
    #define xTaskGetApplicationTaskTag    MPU_xTaskGetApplicationTaskTag
    #define xTaskCallApplicationTaskHook  MPU_xTaskCallApplicationTaskHook
    #define uxTaskGetStackHighWaterMark   MPU_uxTaskGetStackHighWaterMark
    #define xTaskGetCurrentTaskHandle   MPU_xTaskGetCurrentTaskHandle
    #define xTaskGetSchedulerState      MPU_xTaskGetSchedulerState
    #define xTaskGetIdleTaskHandle      MPU_xTaskGetIdleTaskHandle
    #define uxTaskGetSystemState      MPU_uxTaskGetSystemState

    #define xQueueGenericCreate       MPU_xQueueGenericCreate
    #define xQueueCreateMutex       MPU_xQueueCreateMutex
    #define xQueueGiveMutexRecursive    MPU_xQueueGiveMutexRecursive
    #define xQueueTakeMutexRecursive    MPU_xQueueTakeMutexRecursive
    #define xQueueCreateCountingSemaphore MPU_xQueueCreateCountingSemaphore
    #define xQueueGenericSend       MPU_xQueueGenericSend
    #define xQueueAltGenericSend      MPU_xQueueAltGenericSend
    #define xQueueAltGenericReceive     MPU_xQueueAltGenericReceive
    #define xQueueGenericReceive      MPU_xQueueGenericReceive
    #define uxQueueMessagesWaiting      MPU_uxQueueMessagesWaiting
    #define vQueueDelete          MPU_vQueueDelete
    #define xQueueGenericReset        MPU_xQueueGenericReset
    #define xQueueCreateSet         MPU_xQueueCreateSet
    #define xQueueSelectFromSet       MPU_xQueueSelectFromSet
    #define xQueueAddToSet          MPU_xQueueAddToSet
    #define xQueueRemoveFromSet       MPU_xQueueRemoveFromSet
    #define xQueuePeekFromISR       MPU_xQueuePeekFromISR
    #define xQueueGetMutexHolder      MPU_xQueueGetMutexHolder
```

```
    #define pvPortMalloc          MPU_pvPortMalloc
    #define vPortFree            MPU_vPortFree
    #define xPortGetFreeHeapSize      MPU_xPortGetFreeHeapSize
    #define vPortInitialiseBlocks     MPU_vPortInitialiseBlocks

    #if configQUEUE_REGISTRY_SIZE > 0
      #define vQueueAddToRegistry       MPU_vQueueAddToRegistry
      #define vQueueUnregisterQueue     MPU_vQueueUnregisterQueue
    #endif

    /* Remove the privileged function macro. */
    #define PRIVILEGED_FUNCTION

  #else /* MPU_WRAPPERS_INCLUDED_FROM_API_FILE */

        /* Ensure API functions go in the privileged execution section. */
        #define PRIVILEGED_FUNCTION  __attribute__ ((section(".kernelTEXT")))
        #define PRIVILEGED_DATA __attribute__ ((section(".kernelBSS")))

  #endif /* MPU_WRAPPERS_INCLUDED_FROM_API_FILE */

#else /* portUSING_MPU_WRAPPERS */

  #define PRIVILEGED_FUNCTION
  #define PRIVILEGED_DATA
  #define portUSING_MPU_WRAPPERS 0

#endif /* portUSING_MPU_WRAPPERS */


#endif /* MPU_WRAPPERS_H */

line num 83 os_wpu_wrappers.h
```

The `xTaskGenericCreate` function is a FreeRTOS API function that is **used to create a new task**. The function is defined in the FreeRTOS source code, and its implementation can vary depending on the specific port and architecture being used.

The code snippet you provided is from the `mpu_wrappers.h` header file, which is used to redefine API functions for ports that are using the MPU (Memory Protection Unit) feature. The MPU is a hardware feature that provides memory protection and access control for embedded systems, and FreeRTOS provides support for it through the `portUSING_MPU_WRAPPERS` configuration setting.

When the `portUSING_MPU_WRAPPERS` setting is enabled, the `xTaskGenericCreate` function is redefined as `MPU_xTaskGenericCreate` in the `mpu_wrappers.h` file using a preprocessor macro. This allows the function to be called through a wrapper function that sets up the MPU configuration before calling the actual implementation of the function.

The purpose of this wrapper function is to provide additional security and safety features for tasks that are created using the `xTaskGenericCreate` function. By using the

MPU, the wrapper function can ensure that each task is executed within its designated memory space and has access only to the resources that it needs, which helps to prevent memory corruption and other types of security vulnerabilities.

Overall, the `xTaskGenericCreate` function is an essential API function in FreeRTOS that is used to create new tasks, and its implementation can be customized for different architectures and configurations. The `MPU_xTaskGenericCreate` wrapper function provides additional security features for tasks that are created using this function on systems that support the MPU.

> 💡 The source code for the `xTaskGenericCreate` function is available in the FreeRTOS source code, which is open source and freely available. The exact implementation of the function can vary depending on the specific port and architecture being used, but the core functionality is the same across all implementations.
>
> To view the source code for `xTaskGenericCreate`, you can download the FreeRTOS source code from the official FreeRTOS website (**https://www.freertos.org/**) or from the FreeRTOS GitHub repository (**https://github.com/FreeRTOS/FreeRTOS-Kernel**). The source code is well-documented and organized, making it relatively easy to navigate and understand.

💡 API stands for Application Programming Interface. In the context of software development, an API is a set of functions, protocols, and tools for building software applications.

API functions are functions that are exposed by a software library or framework for use by developers. They provide a standardized interface for interacting with the underlying system or software, abstracting away the complexity of the system and making it easier for developers to write software that interacts with it.

In the context of FreeRTOS, the API functions are the set of functions that are provided by the FreeRTOS kernel for creating and managing tasks, semaphores, mutexes, queues, and other operating system constructs. These functions are defined in the FreeRTOS API documentation, and developers can use them to create applications that run on FreeRTOS.

So from now onward if we come across these types of function its enough to know how it works no need to look at them in detail like its source code.

vTaskStartScheduler Function

```
void vTaskStartScheduler( void )
{
    BaseType_t xReturn;

    /* Add the idle task at the lowest priority. */
#if ( INCLUDE_xTaskGetIdleTaskHandle == 1 )
    {
        /* Create the idle task, storing its handle in xIdleTaskHandle so it can
    be returned by the xTaskGetIdleTaskHandle() function. */
        xReturn = xTaskCreate( prvIdleTask, "IDLE", tskIDLE_STACK_SIZE, ( void * ) NUL
L, ( tskIDLE_PRIORITY | portPRIVILEGE_BIT ), &xIdleTaskHandle ); /*lint !e961 MISRA ex
ception, justified as it is not a redundant explicit cast to all supported compilers.
 */
    }
#else
    {
        /* Create the idle task without storing its handle. */
        xReturn = xTaskCreate( prvIdleTask, "IDLE", tskIDLE_STACK_SIZE, ( void * ) NUL
L, ( tskIDLE_PRIORITY | portPRIVILEGE_BIT ), NULL );  /*lint !e961 MISRA exception, ju
stified as it is not a redundant explicit cast to all supported compilers. */
    }
```

```
#endif /* INCLUDE_xTaskGetIdleTaskHandle */

#if ( configUSE_TIMERS == 1 )
    {
        if( xReturn == pdPASS )
        {
            xReturn = xTimerCreateTimerTask();
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
#endif /* configUSE_TIMERS */

    if( xReturn == pdPASS )
    {
        /* Interrupts are turned off here, to ensure a tick does not occur
    before or during the call to xPortStartScheduler().  The stacks of
    the created tasks contain a status word with interrupts switched on
    so interrupts will automatically get re-enabled when the first task
    starts to run. */
        portDISABLE_INTERRUPTS();

#if ( configUSE_NEWLIB_REENTRANT == 1 )
        {
            /* Switch Newlib's _impure_ptr variable to point to the _reent
      structure specific to the task that will run first. */
            _impure_ptr = &( pxCurrentTCB->xNewLib_reent );
        }
#endif /* configUSE_NEWLIB_REENTRANT */

        xSchedulerRunning = pdTRUE;
        xTickCount = ( TickType_t ) 0U;

        /* If configGENERATE_RUN_TIME_STATS is defined then the following
    macro must be defined to configure the timer/counter used to generate
    the run time counter time base. */
        portCONFIGURE_TIMER_FOR_RUN_TIME_STATS();

        /* Setting up the timer tick is hardware specific and thus in the
    portable interface. */
        if( xPortStartScheduler() != pdFALSE )
        {
            /* Should not reach here as if the scheduler is running the
      function will not return. */
        }
        else
        {
            /* Should only reach here if a task calls xTaskEndScheduler(). */
        }
    }
    else
    {
        /* This line will only be reached if the kernel could not be started,
    because there was not enough FreeRTOS heap to create the idle task
    or the timer task. */
        configASSERT( xReturn );
```

```
    }
}
/*-----------------------------------------------------------*/
```

This code is the implementation of the FreeRTOS scheduler, which is responsible for managing the execution of tasks on an embedded system.

The function `vTaskStartScheduler` initializes the scheduler and starts the execution of tasks. It performs the following steps:

1. Create the idle task: The idle task is a special task that runs when there are no other tasks to execute. It is created using the `xTaskCreate` API function, with the task function `prvIdleTask` and the task name "IDLE". If `INCLUDE_xTaskGetIdleTaskHandle` is defined, the task handle is stored in `xIdleTaskHandle` so it can be retrieved later using the `xTaskGetIdleTaskHandle` function.
2. Create the timer task: If `configUSE_TIMERS` is defined, a timer task is created using the `xTimerCreateTimerTask` function.
3. Disable interrupts: Interrupts are disabled to prevent a tick from occurring before or during the call to `xPortStartScheduler`.
4. Switch the Newlib reentrant structure: If `configUSE_NEWLIB_REENTRANT` is defined, the `_impure_ptr` variable is switched to point to the `_reent` structure specific to the task that will run first.
5. Set up the timer tick: If `configGENERATE_RUN_TIME_STATS` is defined, the timer/counter used to generate the run time counter time base is configured using the `portCONFIGURE_TIMER_FOR_RUN_TIME_STATS` macro.
6. Start the scheduler: The function `xPortStartScheduler` is called to start the scheduler. If the function returns, it means that the scheduler has stopped due to a task calling `xTaskEndScheduler`.
7. Handle errors: If there is not enough FreeRTOS heap to create the idle task or the timer task, `xReturn` will not be `pdPASS`, and the function will assert using the `configASSERT` macro.

The `vTaskStartScheduler` function is typically called at the end of the system initialization code and should not return.

EXPLANATION

1. **Create the idle task:** The idle task is a special task that runs when there are no other tasks to execute. It is created using the xTaskCreate API function, with the task function prvIdleTask and the task name "IDLE". If INCLUDE_xTaskGetIdleTaskHandle is defined, the task handle is stored in xIdleTaskHandle so it can be retrieved later using the xTaskGetIdleTaskHandle function.
2. **Create the timer task:** If configUSE_TIMERS is defined, a timer task is created using the xTimerCreateTimerTask function. The timer task is responsible for managing FreeRTOS software timers.
3. **Disable interrupts:** Interrupts are disabled to prevent a tick from occurring before or during the call to xPortStartScheduler. This ensures that the scheduler starts with a clean slate and avoids any timing issues during startup.
4. **Switch the Newlib reentrant structure:**

💡 The Newlib C library provides a set of standard C library functions that are widely used in embedded systems. However, the Newlib C library is not designed to be thread-safe or reentrant, which means that it cannot be safely used in a multi-tasking environment where multiple tasks are executing concurrently.

To address this issue, FreeRTOS provides a feature called "Newlib reentrancy support", which allows multiple tasks to use the Newlib C library functions safely and concurrently. This feature is enabled by defining the configUSE_NEWLIB_REENTRANT configuration macro.

When this feature is enabled, FreeRTOS provides a set of C library functions that are reentrant and thread-safe, and that can be used in place of the standard Newlib C library functions. These functions use a thread-specific instance of the _reent structure to store their internal state, which ensures that each task has its own copy of the C library state.

The vTaskStartScheduler function calls xPortStartScheduler, which is the FreeRTOS scheduler function that starts the scheduler. Before calling xPortStartScheduler, vTaskStartScheduler switches the _impure_ptr variable to point to the _reent structure specific to the task that will run first. This ensures that the first task that runs after the scheduler starts has its own copy of the C library state.

By switching the _impure_ptr variable in this way, FreeRTOS ensures that each task that uses the Newlib C library functions has its own copy of the C library state, which eliminates conflicts between tasks that use the same C library functions. This is an important feature when developing multi-tasking applications that use the Newlib C library, and it helps to ensure that the application is robust and reliable.

5. **Set up the timer tick:**

💡 **When `configGENERATE_RUN_TIME_STATS` is defined, FreeRTOS collects run-time statistics for tasks in the system.** These statistics can be used to monitor the execution time of tasks and to identify performance bottlenecks in the system.

The statistics are generated using a timer/counter that is configured using the `portCONFIGURE_TIMER_FOR_RUN_TIME_STATS` macro. The timer/counter is typically a hardware timer that generates an interrupt at a fixed frequency. **When the interrupt occurs, the interrupt service routine (ISR) updates the run-time counters for each task.**

The `portCONFIGURE_TIMER_FOR_RUN_TIME_STATS` macro is defined in the port layer for each supported platform. It configures the timer/counter for the appropriate frequency and interrupt priority. The macro is typically customized for each platform to ensure that the timer/counter is configured correctly for that platform.

Once the timer/counter is configured, the FreeRTOS scheduler automatically collects the run-time statistics for each task. The statistics can be accessed using the `ulTaskGetRunTimeStats` function or by using a performance analysis tool that supports the FreeRTOS run-time statistics format.

> The run-time statistics are useful for analyzing the performance of the system and identifying potential performance bottlenecks. By measuring the run-time of each task and interrupt service routine, you can identify which parts of the system are consuming the most CPU time and optimize accordingly.
>
> For example, if a particular task is taking up a lot of CPU time, you can investigate its code and try to optimize it by reducing unnecessary computations, optimizing algorithms, or re-structuring the code. Alternatively, you can increase the priority of other tasks that are not getting enough CPU time.
>
> Run-time statistics can also help identify unexpected behavior in the system. For example, if the run-time of a particular task

> suddenly increases, it may indicate a bug or a system overload that needs to be addressed.
>
> Overall, run-time statistics can be a powerful tool for optimizing the performance and reliability of a real-time system.

6. **Start the scheduler:** The function xPortStartScheduler is called to start the scheduler. This function is responsible for setting up the system timer tick, enabling interrupts, and switching to the first task.

   Will learn in detail about this under function `xPortStartScheduler()` in **os_port.c line num 381**

7. **Handle errors:** If there is not enough FreeRTOS heap to create the idle task or the timer task, xReturn will not be pdPASS, and the function will assert using the configASSERT macro. This ensures that any issues during startup are caught and handled appropriately.

💡 In FreeRTOS, `xReturn` is a variable that holds the return value of various functions. The `pdPASS` macro is defined as 1, and is often used to indicate a successful completion of a function call. So when a function returns `pdPASS`, it means the function has completed successfully.

For example, in the context of the `vTaskStartScheduler` function we were discussing earlier, `xReturn` is used to store the return value of the `xPortStartScheduler` function, which in turn returns `pdPASS` if the scheduler was started successfully.

xPortStartScheduler() in os_port.c (381)

The function xPortStartScheduler is a part of the FreeRTOS kernel and is responsible for starting the scheduler. Here's a detailed explanation of the steps involved in this process:

1. Configure the system timer tick: The scheduler requires a timer tick interrupt to schedule tasks. xPortStartScheduler configures the timer tick to generate

interrupts at a regular interval by calling the portSETUP_TICK_INTERRUPT() macro. The interval is set to the value of configTICK_RATE_HZ, which is typically defined as 1000 Hz.

2. Enable interrupts: The scheduler requires interrupts to be enabled to respond to external events and switch between tasks. xPortStartScheduler enables interrupts by calling the portENABLE_INTERRUPTS() macro.
3. Initialize the task lists: The scheduler initializes its internal data structures to manage tasks. It sets up the ready lists, blocked lists, and delayed lists to hold tasks in different states.
4. Switch to the first task: The scheduler switches to the highest-priority task that is ready to run. This is done by calling the portRESTORE_CONTEXT() macro to restore the context of the first task. This will switch the program counter to the first instruction of the first task.
5. Run the scheduler: The scheduler runs indefinitely, switching between tasks as necessary. When a task becomes blocked or delayed, it is moved to the appropriate list, and the scheduler selects the next task to run. When a task is ready to run again, it is moved back to the ready list, and the scheduler selects it when it is the highest-priority task.

Overall, xPortStartScheduler is a crucial function that initializes and runs the FreeRTOS kernel. It configures the timer tick, enables interrupts, and switches to the first task to start executing the user code.

```
BaseType_t xPortStartScheduler(void)
{
  /* Configure the regions in the MPU that are common to all tasks. */
  prvSetupDefaultMPU();

  /* Start the timer that generates the tick ISR. */
  prvSetupTimerInterrupt();

  /* Reset the critical section nesting count read to execute the first task. */
  ulCriticalNesting = 0;

  /* Start the first task.  This is done from portASM.asm as ARM mode must be
  used. */
  vPortStartFirstTask();

  /* Should not get here! */
  return pdFAIL;
}
```

The xPortStartScheduler function is responsible for starting the FreeRTOS scheduler. Here is a more detailed explanation of what this function does:

1. prvSetupDefaultMPU(): This function is called to configure the regions in the MPU that are common to all tasks. MPU stands for Memory Protection Unit, and it is a hardware feature available on some processors that allows the operating system to restrict access to memory regions. In FreeRTOS, the MPU is used to protect the kernel's memory and prevent tasks from accessing each other's memory.
2. prvSetupTimerInterrupt(): This function is called to start the timer that generates the tick ISR. The tick ISR is a periodic interrupt that is used by the FreeRTOS scheduler to keep track of time and switch tasks.(its like for each task there is a specific time after that it will produce an interrupt so that current tasks gets back to ready state and other comes to running state).
3. ulCriticalNesting = 0: This line resets the critical section nesting count. Critical sections are used to protect shared resources and prevent race conditions. When a critical section is entered, interrupts are disabled and the critical section nesting count is incremented. When the critical section is exited, the nesting count is decremented and interrupts are re-enabled if the nesting count reaches zero.
4. vPortStartFirstTask(): This function is responsible for starting the first task. This is done from portASM.asm as ARM mode must be used. The first task that runs after the scheduler is started is the task with the highest priority that is in the "Ready" state.
5. return pdFAIL: This line is executed only if the function vPortStartFirstTask() fails to start a task. It returns pdFAIL, which is a macro defined as (BaseType_t)0. If the scheduler starts successfully, this line should not be reached.

In summary, the xPortStartScheduler function is responsible for starting the FreeRTOS scheduler by configuring the MPU, starting the timer tick, resetting the critical section nesting count, and starting the first task.

💡 A critical section is a piece of code that accesses shared resources (such as variables, data structures, or peripherals) that cannot be accessed simultaneously by multiple tasks or interrupts. During a critical section, it is important to prevent other tasks or interrupts from accessing the shared resource to avoid race conditions, data corruption, or other issues.

To prevent other tasks or interrupts from accessing the shared resource during a critical section, FreeRTOS uses a mechanism called a "critical section nesting count". When a task or interrupt enters a critical section, it increments the nesting count, and when it exits the critical section, it decrements the nesting count. If the nesting count is zero, then the critical section is considered to be "unlocked" and other tasks or interrupts are free to access the shared resource. However, if the nesting count is non-zero, then the critical section is considered to be "locked" and other tasks or interrupts are blocked from accessing the shared resource until the nesting count returns to zero.

In FreeRTOS, critical sections are typically managed using the taskENTER_CRITICAL() and taskEXIT_CRITICAL() macros, which automatically manage the nesting count and interrupts

prvSetupDefaultMPU Function

```
static void prvSetupDefaultMPU( void )
{
  /* make sure MPU is disabled */
  prvMpuDisable();

  /* First setup the entire flash for unprivileged read only access. */
  prvMpuSetRegion(portUNPRIVILEGED_FLASH_REGION,  0x00000000, portMPU_SIZE_4MB | portMPU_REGION_ENABLE, portMPU_PRIV_RO_USER_RO_EXEC | portMPU_NORMAL_OIWTNOWA_SHARED);

  /* Setup the first 32K for privileged only access.  This is where the kernel code is
  placed. */
  prvMpuSetRegion(portPRIVILEGED_FLASH_REGION,  0x00000000, portMPU_SIZE_32KB | portMPU_REGION_ENABLE, portMPU_PRIV_RO_USER_NA_EXEC | portMPU_NORMAL_OIWTNOWA_SHARED);

  /* Setup the the entire RAM region for privileged read-write and unprivileged read only access */
  prvMpuSetRegion(portPRIVILEGED_RAM_REGION,  0x08000000, portMPU_SIZE_512KB | portMPU_REGION_ENABLE, portMPU_PRIV_RW_USER_RO_EXEC | portMPU_NORMAL_OIWTNOWA_SHARED);

  /* Default peripherals setup */
```

```
  prvMpuSetRegion(portGENERAL_PERIPHERALS_REGION,  0xF0000000,
          portMPU_SIZE_256MB | portMPU_REGION_ENABLE | portMPU_SUBREGION_1_DISABLE | p
ortMPU_SUBREGION_2_DISABLE | portMPU_SUBREGION_3_DISABLE | portMPU_SUBREGION_4_DISABL
E,
          portMPU_PRIV_RW_USER_RW_NOEXEC | portMPU_DEVICE_NONSHAREABLE);

  /* Privilege System Region setup */
  prvMpuSetRegion(portPRIVILEGED_SYSTEM_REGION,  0xFFF80000, portMPU_SIZE_512KB | port
MPU_REGION_ENABLE, portMPU_PRIV_RW_USER_RO_NOEXEC | portMPU_DEVICE_NONSHAREABLE);

  /* Enable MPU */
  prvMpuEnable();
}
```

The function `prvSetupDefaultMPU()` sets up the Memory Protection Unit (MPU) to provide memory protection to the system. The MPU is a hardware component in the microcontroller that provides a means of defining memory regions with different access permissions. By configuring the MPU, the system can provide different levels of access to different memory regions, which can help to prevent accidental or malicious corruption of memory and improve system stability.

The function first disables the MPU using the `prvMpuDisable()` function.

`prvMpuSetRegion`: it sets up the MPU regions for flash, RAM, peripherals, and the system. The flash is set up for unprivileged read-only access, except for the first 32 KB where the kernel code is placed, which is set up for privileged read-only access. The RAM is set up for privileged read-write and unprivileged read-only access. The peripherals are set up for privileged read-write and non-shareable access, and the system region is set up for privileged read-write and non-executable, non-shareable access.

After configuring the MPU regions, the `prvMpuEnable()` function is called to enable the MPU.

vPortStartFirstTask() Function

```
/* vPortStartFirstSTask() is defined in portASM.asm */
extern void vPortStartFirstTask( void );
```

It is defined in portASM.asm

The `vPortStartFirstTask()` function is declared to return void, which means it does not return any value. Therefore, there is no need to check its return value in the code that calls it. Once `vPortStartFirstTask()` completes its execution, it starts the scheduler, which never returns control back to the caller. The scheduler runs continuously, and task switching occurs based on the scheduling algorithm chosen at compile-time.

TASK CONTROL BOX (In Detail have been discussed after list_t structure)

```
typedef struct tskTaskControlBlock
{
    volatile StackType_t  *pxTopOfStack;  /*< Points to the location of the last item
 placed on the tasks stack.  THIS MUST BE THE FIRST MEMBER OF THE TCB STRUCT. */

#if ( portUSING_MPU_WRAPPERS == 1 )
    xMPU_SETTINGS xMPUSettings;   /*< The MPU settings are defined as part of the port
layer.  THIS MUST BE THE SECOND MEMBER OF THE TCB STRUCT. */
    BaseType_t    xUsingStaticallyAllocatedStack; /* Set to pdTRUE if the stack is a s
tatically allocated array, and pdFALSE if the stack is dynamically allocated. */
#endif

    ListItem_t      xGenericListItem; /*< The list that the state list item of a task
 is reference from denotes the state of that task (Ready, Blocked, Suspended ). */
    ListItem_t      xEventListItem;   /*< Used to reference a task from an event list.
*/
    UBaseType_t     uxPriority;     /*< The priority of the task.  0 is the lowest pri
ority. */
    StackType_t     *pxStack;     /*< Points to the start of the stack. */
    char        pcTaskName[ configMAX_TASK_NAME_LEN ];/*< Descriptive name given to th
e task when created.  Facilitates debugging only. */ /*lint !e971 Unqualified char typ
es are allowed for strings and single characters only. */

#if ( portSTACK_GROWTH > 0 )
    StackType_t   *pxEndOfStack;    /*< Points to the end of the stack on architecture
s where the stack grows up from low memory. */
#endif

#if ( portCRITICAL_NESTING_IN_TCB == 1 )
    UBaseType_t   uxCriticalNesting;  /*< Holds the critical section nesting depth for
ports that do not maintain their own count in the port layer. */
#endif

#if ( configUSE_TRACE_FACILITY == 1 )
    UBaseType_t   uxTCBNumber;    /*< Stores a number that increments each time a TCB
 is created.  It allows debuggers to determine when a task has been deleted and then r
ecreated. */
    UBaseType_t   uxTaskNumber;   /*< Stores a number specifically for use by third pa
rty trace code. */
#endif

#if ( configUSE_MUTEXES == 1 )
    UBaseType_t   uxBasePriority;   /*< The priority last assigned to the task - used
 by the priority inheritance mechanism. */
    UBaseType_t   uxMutexesHeld;
#endif
```

```
#if ( configUSE_APPLICATION_TASK_TAG == 1 )
    TaskHookFunction_t pxTaskTag;
#endif

#if ( configGENERATE_RUN_TIME_STATS == 1 )
    uint32_t    ulRunTimeCounter; /*< Stores the amount of time the task has spent in
 the Running state. */
#endif

#if ( configUSE_NEWLIB_REENTRANT == 1 )
    /* Allocate a Newlib reent structure that is specific to this task.
    Note Newlib support has been included by popular demand, but is not
    used by the FreeRTOS maintainers themselves.  FreeRTOS is not
    responsible for resulting newlib operation.  User must be familiar with
    newlib and must provide system-wide implementations of the necessary
    stubs. Be warned that (at the time of writing) the current newlib design
    implements a system-wide malloc() that must be provided with locks. */
    struct  _reent xNewLib_reent;
#endif

#if ( configUSE_TASK_NOTIFICATIONS == 1 )
    volatile uint32_t ulNotifiedValue;
    volatile eNotifyValue eNotifyState;
#endif
    //#if ( configUSE_EDFVD_SCHEDULER == 1 )
    /*UBaseType_t task_NormalBudget;
    UBaseType_t task_SafeBudget;
    UBaseType_t task_DegradedBudget1;
    UBaseType_t task_DegradedBudget2;
    UBaseType_t task_Behaviour1;
    UBaseType_t task_Behaviour2;
    UBaseType_t task_Criticality;
    UBaseType_t task_current_executionLimit;
    UBaseType_t task_current_Behaviour;
    UBaseType_t uxOriginalPriority;
    UBaseType_t uxPseudoPriority;*/

    //uint32_t taskProperty[3][4] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

    UBaseType_t task_Criticality;
    UBaseType_t task_current_executionLimit;
    UBaseType_t task_current_Behaviour;
    UBaseType_t taskID;
    UBaseType_t task_SafeBudget;


    //   #endif
} tskTCB;
```

The `tskTaskControlBlock` is a struct that defines the control block for a task in FreeRTOS. It contains several members that provide information about the task, such as its priority, stack, name, and various other properties.

The first member of the struct is a pointer to the top of the task's stack, followed by several other members, including `xGenericListItem` and `xEventListItem`, which are used to reference the task from various lists.

Other members of the struct include `uxPriority`, which specifies the task's priority, `pcTaskName`, which is a character array that stores the task's name, and `pxStack`, which points to the start of the task's stack.

There are also members that are used for various features that can be enabled or disabled at compile-time, such as `uxTCBNumber` and `uxTaskNumber`, which are used for debugging and tracing, and `uxMutexesHeld`, which is used to keep track of the number of mutexes held by the task.

Finally, there are several members that are specific to certain features or architectures, such as `xMPUSettings` and `xUsingStaticallyAllocatedStack`, which are used for memory protection on certain architectures, and `xNewLib_reent`, which is a structure used for reentrancy when using the Newlib C library.

The `tskTCB` struct is defined in FreeRTOS and is used internally by the kernel to manage tasks. It should not be modified by user code.

IMPORTANT

```
/* Lists for ready and blocked tasks. --------------------*/
PRIVILEGED_DATA static List_t pxReadyTasksLists[ configMAX_PRIORITIES ];/*< Prioritise
d ready tasks. */
PRIVILEGED_DATA static List_t xDelayedTaskList1;            /*< Delayed tasks. */
PRIVILEGED_DATA static List_t xDelayedTaskList2;            /*< Delayed tasks (two lis
ts are used - one for delays that have overflowed the current tick count. */
PRIVILEGED_DATA static List_t * volatile pxDelayedTaskList;       /*< Points to the de
layed task list currently being used. */
PRIVILEGED_DATA static List_t * volatile pxOverflowDelayedTaskList;   /*< Points to th
e delayed task list currently being used to hold tasks that have overflowed the curren
t tick count. */
PRIVILEGED_DATA static List_t xPendingReadyList;            /*< Tasks that have been r
eadied while the scheduler was suspended.  They will be moved to the ready list when t
he scheduler is resumed. */
line 223 os_tasks.c
```

```
/*
 * Definition of the type of queue used by the scheduler.
 */
typedef struct xLIST
{
  listFIRST_LIST_INTEGRITY_CHECK_VALUE        /*< Set to a known value if configUSE_LI
```

```
ST_DATA_INTEGRITY_CHECK_BYTES is set to 1. */
  configLIST_VOLATILE UBaseType_t uxNumberOfItems;
  ListItem_t * configLIST_VOLATILE pxIndex;   /*< Used to walk through the list.  Poin
ts to the last item returned by a call to listGET_OWNER_OF_NEXT_ENTRY (). */
  MiniListItem_t xListEnd;            /*< List item that contains the maximum possible
item value meaning it is always at the end of the list and is therefore used as a mark
er. */
  listSECOND_LIST_INTEGRITY_CHECK_VALUE       /*< Set to a known value if configUSE_LI
ST_DATA_INTEGRITY_CHECK_BYTES is set to 1. */
} List_t;
line 205 os_list.c
```

The `List_t` structure is used to define linked lists that hold tasks in the FreeRTOS kernel

The structure contains the following members:

- `uxNumberOfItems`: The number of items currently in the list.
- `pxIndex`: A pointer to the last item returned by a call to `listGET_OWNER_OF_NEXT_ENTRY()`. This member is used to walk through the list.
- `xListEnd`: A `MiniListItem_t` structure that is always at the end of the list and is used as a marker.
- `listFIRST_LIST_INTEGRITY_CHECK_VALUE` and `listSECOND_LIST_INTEGRITY_CHECK_VALUE`: These are optional members that are used to check the integrity of the list data if `configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES` is set to 1.

⚡ The data structure used above is called a linked list, which is a data structure consisting of a sequence of nodes, where each node contains a value and a pointer to the next node in the sequence.

In this specific implementation, the linked list is implemented as a doubly linked list, meaning each node also contains a pointer to the previous node in the sequence. This allows for easy traversal of the list in both forward and backward directions.

The linked list used in this implementation is defined by the `List_t` structure. Each instance of the `List_t` structure represents a single linked list. The structure contains a `uxNumberOfItems` field which is used to keep track of the number of items in the list. The `pxIndex` field is a pointer that is used to iterate through the list. It points to the last item returned by a call to `listGET_OWNER_OF_NEXT_ENTRY()`, which is a macro that returns the owner of the next item in the list. The `xListEnd` field is a `MiniListItem_t` structure which is used as a marker to indicate the end of the list.

The `MiniListItem_t` structure contains an `xItemValue` field which is used to store the value of the item in the list. In this implementation, `portMAX_DELAY` is used as the value for the `xItemValue` field of the `xListEnd` item to ensure that it always appears at the end of the list. The `pxNext` and `pxPrevious` fields are pointers to the next and previous items in the list, respectively.

Finally, the `vListInitialise()` function is used to initialize a `List_t` instance. It initializes the `pxIndex` pointer to point to the `xListEnd` item, initializes the `xListEnd` item to have a value of `portMAX_DELAY` and to point to itself, and sets the `uxNumberOfItems` field to 0. If data integrity checking is enabled, the function also sets known values into the list integrity check fields.

Functions related to List_t structure in os_list.c

vlistInitialise Function

```
void vListInitialise( List_t * const pxList )
{
  /* The list structure contains a list item which is used to mark the
  end of the list.  To initialise the list the list end is inserted
  as the only list entry. */
```

```
  pxList->pxIndex = ( ListItem_t * ) &( pxList->xListEnd );     /*lint !e826 !e740 The
mini list structure is used as the list end to save RAM.  This is checked and valid.
 */

  /* The list end value is the highest possible value in the list to
  ensure it remains at the end of the list. */
  pxList->xListEnd.xItemValue = portMAX_DELAY;

  /* The list end next and previous pointers point to itself so we know
  when the list is empty. */
  pxList->xListEnd.pxNext = ( ListItem_t * ) &( pxList->xListEnd ); /*lint !e826 !e740
The mini list structure is used as the list end to save RAM.  This is checked and vali
d. */
  pxList->xListEnd.pxPrevious = ( ListItem_t * ) &( pxList->xListEnd );/*lint !e826 !e
740 The mini list structure is used as the list end to save RAM.  This is checked and
 valid. */

  pxList->uxNumberOfItems = ( UBaseType_t ) 0U;

  /* Write known values into the list if
  configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES is set to 1. */
  listSET_LIST_INTEGRITY_CHECK_1_VALUE( pxList );
  listSET_LIST_INTEGRITY_CHECK_2_VALUE( pxList );
}
```

Line by Line Explanation

This is the definition of the `vListInitialise` function in FreeRTOS, which initializes a doubly linked list. Let's go through the function line by line:

```
pxList->pxIndex = (ListItem_t*)&(pxList->xListEnd);
```

This line sets the initial value of `pxIndex` to point to the `xListEnd` member of the list structure. This member is used as a sentinel value to mark the end of the list.

```
pxList->xListEnd.xItemValue = portMAX_DELAY;
```

This line sets the value of `xItemValue` in the `xListEnd` member to the maximum possible value, which ensures that it always remains at the end of the list.

```
pxList->xListEnd.pxNext = (ListItem_t*)&(pxList->xListEnd);
pxList->xListEnd.pxPrevious = (ListItem_t*)&(pxList->xListEnd);
```

These two lines set the `pxNext` and `pxPrevious` members of the `xListEnd` member to point to itself. This means that if there are no other items in the list, the `pxIndex` member of the list structure points to `xListEnd`, which points back to itself, indicating that the list is empty.

```
pxList->uxNumberOfItems = (UBaseType_t)0U;
```

This line sets the initial value of `uxNumberOfItems` to zero, indicating that the list is empty.

```
listSET_LIST_INTEGRITY_CHECK_1_VALUE(pxList);
listSET_LIST_INTEGRITY_CHECK_2_VALUE(pxList);
```

These two lines set the initial values of two integrity check members in the `List_t` structure. These members are used for data integrity checking to ensure that the list is not corrupted during operation.

Overall, `vListInitialise` initializes a doubly linked list by setting the sentinel value `xListEnd` as the only item in the list and initializing the other members of the list structure accordingly.

💡 Data integrity checking is the process of verifying the accuracy and consistency of data stored or transmitted over a system or network. It involves detecting and preventing errors or data corruption that may occur due to various reasons, such as hardware or software failures, human errors, or malicious attacks.

In embedded systems or real-time applications, data integrity is crucial to ensure the reliability and safety of the system. For example, in a medical device, a corrupted data stream may cause incorrect readings or incorrect treatment, which can be life-threatening.

To ensure data integrity, various techniques are used, such as redundancy, error detection, and error correction codes. In addition, data integrity checking can be done by adding checksums or hashes to the data, which can be used to verify that the data has not been tampered with or corrupted during transmission or storage.

In the context of FreeRTOS, data integrity checking is an optional feature that can be enabled by setting the `configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES` configuration macro to 1. When this feature is enabled, two additional fields are added to the `List_t` structure, which are used to store known values that are checked to ensure the data integrity of the list.

🤔 Based on the provided code snippet, we can confirm that the list being initialized is a circular double linked list. Here's why:

1. Initialization of List End:
    - The code initializes the `xListEnd` member of the `pxList` structure, which acts as the end marker of the list.
    - The `xListEnd` node is inserted as the only entry in the list by assigning its address to `pxList->pxIndex`.
    - The `xListEnd` node is used to mark the end of the list.
2. Circular Connection:
    - The `xListEnd` node has both `pxNext` and `pxPrevious` pointers that point to itself.
    - `pxList->xListEnd.pxNext` is assigned the address of `pxList->xListEnd`, making it point to itself.
    - `pxList->xListEnd.pxPrevious` is assigned the address of `pxList->xListEnd`, making it point to itself.
    - This circular connection ensures that the end of the list is linked back to itself.
3. Other Initializations:
    - The `xItemValue` of `xListEnd` is set to `portMAX_DELAY`, which is likely used as a special value to indicate the end of the list.
    - The `uxNumberOfItems` variable is set to 0, indicating that the list is initially empty.

Therefore, based on the code snippet provided, it can be confirmed that a circular double linked list is being initialized.

vlistInitialiseItem Function

```
void vListInitialiseItem( ListItem_t * const pxItem )
{
```

```
    /* Make sure the list item is not recorded as being on a list. */
    pxItem->pvContainer = NULL;

    /* Write known values into the list item if
    configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES is set to 1. */
    listSET_FIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE( pxItem );
    listSET_SECOND_LIST_ITEM_INTEGRITY_CHECK_VALUE( pxItem );
}
/*-----------------------------------------------------------*/
```

Line by Line Explanation

This function initializes a given list item, represented by the pointer `pxItem`. Here's a detailed explanation of each line:

```
pxItem->pvContainer = NULL;
```

This line sets the `pvContainer` member of the list item to `NULL`. This member is used to keep track of which list the item belongs to. Setting it to `NULL` ensures that the item is not recorded as being on any list at the time of initialization.

```
listSET_FIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE( pxItem );
listSET_SECOND_LIST_ITEM_INTEGRITY_CHECK_VALUE( pxItem );
```

These two lines are macro calls that set the integrity check values for the list item. These checks are used to detect memory corruption or other errors that could cause the linked list to become corrupted. The macros are only defined if `configUSE_LIST_DATA_INTEGRITY_CHECK_BYTES` is set to `1`. If it is not, these lines will not have any effect.

DIFFERENCE BTW VLISTINITIALISE AND VLISTINITIALISEITEM

`vListInitialise` is a function that initializes a whole list. It sets up the initial values of the `List_t` structure that represents the list and prepares it for use.

`vListInitialiseItem`, on the other hand, is a function that initializes a single `ListItem_t` structure, which is the basic building block of a linked list. It sets up the initial values of the `ListItem_t` structure, such as the `pvContainer` pointer that points to the list that this item belongs to.

When a new item is added to a list, it should be initialized using `vListInitialiseItem` before being added to the list using a list insertion function. The `pvContainer` pointer is set to NULL initially to indicate that the item is not yet on a list. Once the item is added to a list, the `pvContainer` pointer is updated to point to the list it was added to.

> ⚡ In the FreeRTOS kernel, each task has a priority, and the tasks are organized into ready lists based on their priorities. Each ready list is represented as a linked list of `ListItem_t` structures, where each `ListItem_t` structure represents a task that has the same priority. Each `ListItem_t` structure has a pointer to the `TCB_t` structure of the task that it represents. The `TCB_t` structure contains information about the task, such as its stack pointer, program counter, and state.
>
> When a task becomes ready, it is added to the appropriate ready list by creating a new `ListItem_t` structure for the task and inserting it into the ready list. The `pvOwner` member of the `ListItem_t` structure is set to point to the `TCB_t` structure of the task, and the `pxNext` and `pxPrevious` members of the `ListItem_t` structure are set to point to the next and previous `ListItem_t` structures in the ready list.
>
> When a task is selected to run by the scheduler, it is removed from the ready list by removing its `ListItem_t` structure from the list. The `pxNext` and `pxPrevious` pointers of the `ListItem_t` structures before and after the removed `ListItem_t` structure are updated to maintain the integrity of the ready list. The `pvOwner` member of the removed `ListItem_t` structure is set to NULL to indicate that it is no longer on a list.

vlistInsertend Function

Note: `pvContainer` pointer that points to the list that this item belongs to.

Here is a detailed explanation of each line of code in `vListInsertEnd` function:

```
ListItem_t * const pxIndex = pxList->pxIndex;
```

This line of code creates a local pointer `pxIndex` and initializes it to the list's index. `pxIndex` is used to make it easier to read the rest of the code.

```
listTEST_LIST_INTEGRITY( pxList );
listTEST_LIST_ITEM_INTEGRITY( pxNewListItem );
```

These two lines of code check the integrity of the list and the list item respectively. These checks are only performed when `configASSERT()` is defined. `configASSERT()` is a macro that is defined in the FreeRTOSConfig.h header file, and is used to provide run-time checks in the code. In this case, the checks ensure that the list data structures are not being overwritten in memory.

```
pxNewListItem->pxNext = pxIndex;
pxNewListItem->pxPrevious = pxIndex->pxPrevious;
pxIndex->pxPrevious->pxNext = pxNewListItem;
pxIndex->pxPrevious = pxNewListItem;
```

These four lines of code insert a new list item at the end of the list pointed to by `pxList`. The new list item is pointed to by `pxNewListItem`. The first line sets the `pxNext` member of `pxNewListItem` to `pxIndex`, which is the list index. The second line sets the `pxPrevious` member of `pxNewListItem` to `pxIndex->pxPrevious`, which is the item that was previously at the end of the list. The third line sets the `pxNext` member of the previous last item to `pxNewListItem`, and the fourth line sets the `pxPrevious` member of `pxIndex` to `pxNewListItem`.

```
pxNewListItem->pvContainer = ( void * ) pxList;
```

This line of code sets the `pvContainer` member of the new list item to `pxList`, which is a pointer to the list that the item belongs to.

```
( pxList->uxNumberOfItems )++;
```

This line of code increments the number of items in the list pointed to by `pxList`.

In summary, the `vListInsertEnd` function inserts a new list item at the end of the list pointed to by `pxList`. It first checks the integrity of the list and the list item, and then inserts the new item by updating the `pxNext` and `pxPrevious` pointers of the appropriate list items. Finally, it sets the `pvContainer` member of the new list item to the list that it belongs to, and increments the number of items in the list.

vlistInsert Function

⚡ **Sorting concept used here**

`ListItem_t` is a structure that represents a node in a linked list. Each node contains a pointer to the next node in the list (`pxNext`), a pointer to the previous node (`pxPrevious`), and a numerical value (`xItemValue`). The `xItemValue` is used to maintain the order of the nodes in the list.

`pxNewListItem` is a pointer to a `ListItem_t` structure that represents the node to be inserted into the linked list.

When a new node is inserted into the linked list, the `xItemValue` of the new node is compared to the `xItemValue` of the nodes already in the list. The new node is inserted into the list in such a way that the list remains sorted in increasing `xItemValue` order.

Example:

```
Node A: xItemValue = 1
Node B: xItemValue = 3
Node C: xItemValue = 5
```

Now suppose a new node `D` is to be inserted with `xItemValue` = 4. The loop in the `vListInsert` function will start at the beginning of the list, compare `xItemValue` of Node A to `xItemValue` of Node D, which is smaller, then move to Node B and compare its `xItemValue` with `xItemValue` of Node D again, which is smaller too. At Node C, `xItemValue` of Node D is greater, thus Node D will be inserted just before Node C, i.e., after Node B:

```
Node A: xItemValue = 1
Node B: xItemValue = 3
Node D: xItemValue = 4
Node C: xItemValue = 5
```

So the `xItemValue` of each node determines its position in the linked list.

EXPLANATION OF CODE

here is a detailed explanation of each line of code in the `vListInsert()` function:

```
void vListInsert( List_t * const pxList, ListItem_t * const pxNewListItem ){
```

This is the function definition for `vListInsert()`, which takes in a pointer to a list (`pxList`) and a pointer to a new list item (`pxNewListItem`).

```
ListItem_t *pxIterator;
const TickType_t xValueOfInsertion = pxNewListItem->xItemValue;
```

These two lines define a pointer to a list item (`pxIterator`) and a constant tick value (`xValueOfInsertion`) that is obtained from the `xItemValue` field of the new list item.

```
listTEST_LIST_INTEGRITY( pxList );
listTEST_LIST_ITEM_INTEGRITY( pxNewListItem );
```

These two lines perform integrity checks on the list and list item to ensure that they have not been corrupted in memory. These checks are only performed if `configASSERT()` is defined.

```
if( xValueOfInsertion == portMAX_DELAY )
{
pxIterator = pxList->xListEnd.pxPrevious;
}
else
{
for( pxIterator = ( ListItem_t * ) &( pxList->xListEnd ); pxIterator->pxNext->xItemVal
ue <= xValueOfInsertion; pxIterator = pxIterator->pxNext )
{
/* There is nothing to do here, just iterating to the wanted
insertion position. */
}
}
```

🤔 These lines check if the `xValueOfInsertion` is equal to `portMAX_DELAY`. If it is, then the `pxIterator` is set to the previous item in the list. Otherwise, a loop is used to iterate through the list until the correct position is found for the new item. The loop starts at the end of the list and continues until the next item has an `xItemValue` greater than the `xValueOfInsertion`. This ensures that the list remains sorted in increasing `xItemValue` order.

```
pxNewListItem->pxNext = pxIterator->pxNext;
pxNewListItem->pxNext->pxPrevious = pxNewListItem;
pxNewListItem->pxPrevious = pxIterator;
pxIterator->pxNext = pxNewListItem;
```

These lines insert the new list item into the list. The `pxNext` and `pxPrevious` pointers of the new list item and the adjacent list items are updated accordingly

```
pxNewListItem->pvContainer = ( void * ) pxList;
```

This line sets the `pvContainer` field of the new list item to point to the list it has been inserted into.

```
( pxList->uxNumberOfItems )++;
```

Finally, this line increments the `uxNumberOfItems` field of the list to indicate that a new item has been added to the list.

DIFFERENCE BETWEEN THE ABOVE TWO FUNCTIONS

The `vListInsertEnd()` function always inserts a new list item at the end of the list, regardless of its `xItemValue`. On the other hand, `vListInsert()` function inserts a new list item in a sorted manner, based on its `xItemValue`.

In `vListInsert()`, the function iterates through the list, starting from the beginning, until it finds the appropriate position to insert the new list item based on its `xItemValue`. This ensures that the list remains sorted by `xItemValue`. The iterator (`pxIterator`) is then used to insert the new list item at the correct position.

If the new list item has the same `xItemValue` as the last item in the list, `vListInsert()` will insert the new item after it, while `vListInsertEnd()` will insert it at the end of the list, making `vListInsert()` more suitable for maintaining a sorted list.

Both functions also perform some basic error checking and ensure the integrity of the list data structure by calling `listTEST_LIST_INTEGRITY()` and `listTEST_LIST_ITEM_INTEGRITY()` respectively, provided that `configASSERT()` is defined.

uxListRemove Function

```
UBaseType_t uxListRemove( ListItem_t * const pxItemToRemove )
{
/* The list item knows which list it is in.  Obtain the list from the list
item. */
List_t * const pxList = ( List_t * ) pxItemToRemove->pvContainer;

  pxItemToRemove->pxNext->pxPrevious = pxItemToRemove->pxPrevious;
  pxItemToRemove->pxPrevious->pxNext = pxItemToRemove->pxNext;

  /* Make sure the index is left pointing to a valid item. */
  if( pxList->pxIndex == pxItemToRemove )
  {
    pxList->pxIndex = pxItemToRemove->pxPrevious;
  }
  else
  {
    mtCOVERAGE_TEST_MARKER();
  }

  pxItemToRemove->pvContainer = NULL;
  ( pxList->uxNumberOfItems )--;

  return pxList->uxNumberOfItems;
}
/*-----------------------------------------------------------*/
```

This function removes a given item from a list.

```
UBaseType_t uxListRemove( ListItem_t * const pxItemToRemove )
```

- `UBaseType_t` is an unsigned integer type, typically defined as `uint32_t` or `uint16_t`, depending on the architecture. ***This function returns the updated number of items in the list***.
- `ListItem_t` is a structure that represents a list item. It contains a pointer to the next and previous items in the list, as well as an item value, which is used to sort the list.
- `pxItemToRemove` is a pointer to the item to be removed from the list.

```
List_t * const pxList = ( List_t * ) pxItemToRemove->pvContainer;
```

- `List_t` is a structure that represents a list. It contains a pointer to the first and last items in the list, as well as the number of items in the list.

- `pxItemToRemove->pvContainer` is a pointer to the list that contains the item to be removed. The `pvContainer` field is set when an item is inserted into a list.

```
pxItemToRemove->pxNext->pxPrevious = pxItemToRemove->pxPrevious;
pxItemToRemove->pxPrevious->pxNext = pxItemToRemove->pxNext;
```

- These two lines remove the item from the list by updating the `pxNext` and `pxPrevious` pointers of the adjacent items to bypass the item to be removed.

```
if( pxList->pxIndex == pxItemToRemove )
{
pxList->pxIndex = pxItemToRemove->pxPrevious;
}
else
{
mtCOVERAGE_TEST_MARKER();
}
```

- This line updates the list's `pxIndex` field, which is used for fast searching of items in the list. If the item being removed is the same as the `pxIndex`, then the `pxIndex` is updated to point to the previous item. Otherwise, the `pxIndex` remains unchanged.
- `mtCOVERAGE_TEST_MARKER()` is a macro used for code coverage testing and is not relevant to the function's behavior.

```
pxItemToRemove->pvContainer = NULL;
( pxList->uxNumberOfItems )--;
return pxList->uxNumberOfItems;
```

These lines update the item's `pvContainer` field to indicate that it is no longer in a list and decrement the number of items in the list. The updated number of items is then returned.

Concept of pvContainer in Remove

`pvContainer` is a pointer to the list that contains the item to be removed. When an item is added to the list, the list pointer is saved in the `pvContainer` field of the item's structure, and this is how the item knows which list it belongs to.

For example, let's say we have a list of integers:

```
List_t xIntegerList;
```

We can initialize this list using the macro provided by FreeRTOS, like this:

```
LIST_INITIALISE( &xIntegerList );
```

Now, let's say we want to add some items to the list:

```
int a = 10, b = 20, c = 30;
ListItem_t xItem1, xItem2, xItem3;
xItem1.xItemValue = a;
xItem2.xItemValue = b;
xItem3.xItemValue = c;
vListInsertEnd( &xIntegerList, &xItem1 );
vListInsertEnd( &xIntegerList, &xItem2 );
vListInsertEnd( &xIntegerList, &xItem3 );
```

> In the `vListInsertEnd()` function, the `pvContainer` field of each item is set to the address of the list it was inserted into, in this case `&xIntegerList`.

Now, if we want to remove an item from the list, we can use the `uxListRemove()` function:

```
UBaseType_t uxNumberOfItems = uxListRemove( &xItem2 );
```

In this example, we are removing the second item in the list (`&xItem2`). When we call `uxListRemove()`, it retrieves the list pointer from the `pvContainer` field of `xItem2`, which in this case is `&xIntegerList`.

The `uxListRemove()` function then removes the item from the list, updates the `pvContainer` field of the item to `NULL`, decrements the list item count and returns the new item count.

This way, the `pvContainer` field is used to maintain a link between the list and its items, allowing us to perform operations on the list items, such as removal, without needing to pass in the list pointer each time.

vListIterate Function

```
void vListIterate( List_t * const pxList, ListItem_t * const pxNewListItem )
{
  ListItem_t *pxIterator;
  const TickType_t xValueOfInsertion = pxNewListItem->xItemValue;
  for( pxIterator = ( ListItem_t * ) &( pxList->xListEnd ); pxIterator->pxNext->xItemV
alue <= xValueOfInsertion; pxIterator = pxIterator->pxNext )
  {

  }
}
```

The code you provided is an example of a loop called `vListIterate` that iterates through a list based on a specific condition. Let's break down the code and explain it step by step:

1. Initialization:
    - The function `vListIterate` takes two parameters: a pointer to a list (`pxList`) and a pointer to a new list item (`pxNewListItem`).
    - A local variable `pxIterator` is declared, which will be used as the iterator through the list.
    - The value of `xValueOfInsertion` is assigned as the `xItemValue` of `pxNewListItem`.
2. Loop:
    - The loop iterates as long as the `xItemValue` of the next element (`pxIterator->pxNext->xItemValue`) is less than or equal to `xValueOfInsertion`.
    - The loop condition `pxIterator->pxNext->xItemValue <= xValueOfInsertion` is evaluated before each iteration.
    - If the condition is true, the loop continues. Otherwise, the loop terminates.
3. Iteration:
    - During each iteration of the loop, the `pxIterator` is updated using the statement `pxIterator = pxIterator->pxNext`.
    - This moves the iterator to the next element in the list.
    - The loop body is empty, indicated by the comment "There is nothing to do here, just iterating to the wanted insertion position."

The purpose of this loop is to find the desired insertion position in the list based on the `xItemValue` of `pxNewListItem`. By iterating through the list until `xValueOfInsertion` is no longer greater than the `xItemValue` of the next element, the loop effectively finds the correct position for insertion.

It's important to note that the code you provided only performs the iteration and does not perform any actual insertion or modification of the list. The purpose of this loop is to determine the insertion position for a new list item in preparation for subsequent insertion operations.

Remember, the provided code snippet alone does not provide the complete picture of the list implementation, and additional code is required to perform the actual insertion or modification of the list based on the determined insertion position.

OS_QUEUE.C

```
typedef struct QueueDefinition
{
  int8_t *pcHead;         /*< Points to the beginning of the queue storage area. */
  int8_t *pcTail;         /*< Points to the byte at the end of the queue storage area.
Once more byte is allocated than necessary to store the queue items, this is used as a
marker. */
  int8_t *pcWriteTo;        /*< Points to the free next place in the storage area. */

  union             /* Use of a union is an exception to the coding standard to ensure
two mutually exclusive structure members don't appear simultaneously (wasting RAM). */
  {
    int8_t *pcReadFrom;     /*< Points to the last place that a queued item was read f
rom when the structure is used as a queue. */
    UBaseType_t uxRecursiveCallCount;/*< Maintains a count of the number of times a re
cursive mutex has been recursively 'taken' when the structure is used as a mutex. */
  } u;

  List_t xTasksWaitingToSend;   /*< List of tasks that are blocked waiting to post ont
o this queue.  Stored in priority order. */
  List_t xTasksWaitingToReceive;  /*< List of tasks that are blocked waiting to read f
rom this queue.  Stored in priority order. */

  volatile UBaseType_t uxMessagesWaiting;/*< The number of items currently in the queu
e. */
  UBaseType_t uxLength;     /*< The length of the queue defined as the number of items
it will hold, not the number of bytes. */
  UBaseType_t uxItemSize;     /*< The size of each items that the queue will hold. */

  volatile BaseType_t xRxLock;  /*< Stores the number of items received from the queue
(removed from the queue) while the queue was locked.  Set to queueUNLOCKED when the qu
eue is not locked. */
  volatile BaseType_t xTxLock;  /*< Stores the number of items transmitted to the queu
```

```
e (added to the queue) while the queue was locked.  Set to queueUNLOCKED when the queu
e is not locked. */

  #if ( configUSE_TRACE_FACILITY == 1 )
    UBaseType_t uxQueueNumber;
    uint8_t ucQueueType;
  #endif

  #if ( configUSE_QUEUE_SETS == 1 )
    struct QueueDefinition *pxQueueSetContainer;
  #endif

} xQUEUE;
```

The code snippet you provided defines a structure called `QueueDefinition`, which represents a queue data structure. Here is an explanation of the members of the structure and their use:

1. `pcHead`: Points to the beginning of the queue storage area. It indicates the first item in the queue.
2. `pcTail`: Points to the byte at the end of the queue storage area. It acts as a marker to indicate the end of the queue.
3. `pcWriteTo`: Points to the next free place in the storage area. It is used when adding items to the queue.
4. `u`: A union that allows two mutually exclusive structure members to occupy the same memory. It is used to provide different functionality depending on how the queue is used.
   - `pcReadFrom`: Points to the last place from where an item was read when the structure is used as a queue.
   - `uxRecursiveCallCount`: Maintains a count of the number of times a recursive mutex has been recursively 'taken' when the structure is used as a mutex.
5. `xTasksWaitingToSend`: A list of tasks that are blocked and waiting to post (send) data onto this queue. The list is stored in priority order.
6. `xTasksWaitingToReceive`: A list of tasks that are blocked and waiting to read from this queue. The list is stored in priority order.
7. `uxMessagesWaiting`: The number of items currently in the queue.
8. `uxLength`: The length of the queue, defined as the maximum number of items it can hold.
9. `uxItemSize`: The size of each item that the queue can hold.

10. `xRxLock`: Stores the number of items received from the queue (removed from the queue) while the queue was locked. It is set to `queueUNLOCKED` when the queue is not locked.
11. `xTxLock`: Stores the number of items transmitted to the queue (added to the queue) while the queue was locked. It is set to `queueUNLOCKED` when the queue is not locked.
12. `uxQueueNumber` (conditional compilation): A number assigned to the queue for tracing purposes.
13. `ucQueueType` (conditional compilation): A type assigned to the queue for tracing purposes.
14. `pxQueueSetContainer` (conditional compilation): A pointer to the queue set container if the queue is part of a queue set.

This `xQUEUE` structure is typically used as the underlying data structure for implementing a queue in an operating system or real-time operating system (RTOS). It provides the necessary members to manage the storage, tracking of tasks waiting to send/receive, and other properties of the queue. The specific use and behavior of the queue would depend on the implementation of the operating system or RTOS that utilizes this structure.