DSA PROJ

QUEUES

```
#include <stdlib.h>
#include <string.h>
```

These are preprocessor directives that include the standard library headers `stdlib.h` and `string.h`. They provide definitions and declarations for various functions and types used in the code

```
#define MPU_WRAPPERS_INCLUDED_FROM_API_FILE
```

This line defines a preprocessor macro `MPU_WRAPPERS_INCLUDED_FROM_API_FILE`. It is used to prevent redefinition of API functions in the `task.h` header file when included from an application file.

```
#include "FreeRTOS.h"
#include "os_task.h"
#include "os_queue.h"
```

These lines include the FreeRTOS kernel headers `FreeRTOS.h`, `os_task.h`, and `os_queue.h`. These headers provide the necessary definitions and functions for task management and queue operations in FreeRTOS

```
#if ( configUSE_CO_ROUTINES == 1 )
  #include "os_croutine.h"
#endif
```

co routines

Co-routines are a feature of FreeRTOS that allows cooperative multitasking within a single task. Unlike traditional multitasking, where the scheduler decides when to switch between tasks preemptively, co-routines voluntarily yield execution to each other at defined points within the task.

In the context of OS queues, co-routines can be used to implement task synchronization and communication. Here's how co-routines are related to OS queues:

1. Co-routines as Queue Consumers: Co-routines can be designed to act as consumers of OS queues. They can wait for items to be available in a queue and process them when they become available. Co-routines typically yield execution back to the task scheduler after processing an item, allowing other co-routines or tasks to run.
2. Co-routines as Queue Producers: Similarly, co-routines can act as producers, generating data or events and adding them to an OS queue for consumption by other tasks or co-routines. Once an item is added to the queue, the co-routine may yield, allowing other co-routines or tasks to execute.
3. Synchronization and Communication: Co-routines can synchronize with each other or exchange information through OS queues. They can wait for specific events or conditions signaled by other co-routines by blocking on a queue until the desired event or data is available. This allows for efficient inter-co-routine communication and coordination within a task.

By using co-routines in conjunction with OS queues, you can achieve cooperative multitasking behavior and efficient communication between different parts of your application. This can be particularly useful in systems where the overhead of preemptive multitasking is not necessary or desirable.

This code block includes the `os_croutine.h` header if the configuration macro `configUSE_CO_ROUTINES` is defined as 1. Co-routines are a feature of FreeRTOS that allows cooperative multitasking. This block is conditional based on the configuration setting.

This code block is a preprocessor directive that conditionally includes the header file `"os_croutine.h"` based on the value of the configuration macro `configUSE_CO_ROUTINES`.

Here's how it works:

1. `configUSE_CO_ROUTINES` is a configuration macro defined somewhere in the codebase. Its value determines whether co-routines are enabled or not. In this case, it is checked if `configUSE_CO_ROUTINES` is equal to 1.
2. The `#if` directive checks if the condition `(configUSE_CO_ROUTINES == 1)` evaluates to true. If it does, the following code between `#if` and `#endif` will be included during preprocessing. Otherwise, it will be skipped.
3. If the condition is true, the `#include` directive is used to include the header file `"os_croutine.h"`. This header file likely contains the necessary definitions and functions for co-routines in FreeRTOS.
4. Finally, the `#endif` directive marks the end of the conditional block. Any code following this block will be processed normally by the preprocessor.

In summary, this code block allows the inclusion of the `"os_croutine.h"` header file only when co-routines are enabled (`configUSE_CO_ROUTINES == 1`). It provides a way to conditionally include specific headers or sections of code based on configuration settings, allowing for more flexible and customizable builds of the application.

```
#undef MPU_WRAPPERS_INCLUDED_FROM_API_FILE
```

This line undefines the `MPU_WRAPPERS_INCLUDED_FROM_API_FILE` macro. It resets the macro definition, allowing it to be redefined if necessary in subsequent code.

The code you provided seems to be a typical inclusion of necessary headers for using FreeRTOS in an application. It includes standard library headers, FreeRTOS kernel headers, and optionally co-routine headers based on the configuration setting.

```
/* Constants used with the xRxLock and xTxLock structure members. */
#define queueUNLOCKED                ( ( BaseType_t ) -1 )
#define queueLOCKED_UNMODIFIED       ( ( BaseType_t ) 0 )
```

These lines define two constants. `queueUNLOCKED` is set to `(BaseType_t)-1`, which represents an unlocked state for a queue. `queueLOCKED_UNMODIFIED` is set to `(BaseType_t)0`, indicating a locked state that is unmodified

```
/* When the Queue_t structure is used to represent a base queue its pcHead and
pcTail members are used as pointers into the queue storage area.  When the
Queue_t structure is used to represent a mutex pcHead and pcTail pointers are
not necessary, and the pcHead pointer is set to NULL to indicate that the
pcTail pointer actually points to the mutex holder (if any).  Map alternative
names to the pcHead and pcTail structure members to ensure the readability of
the code is maintained despite this dual use of two structure members.  An
alternative implementation would be to use a union, but use of a union is
against the coding standard (although an exception to the standard has been
permitted where the dual use also significantly changes the type of the
structure member). */
```

```
#define pxMutexHolder                   pcTail
#define uxQueueType                     pcHead
#define queueQUEUE_IS_MUTEX             NULL
```

These lines define the macros `pxMutexHolder`, `uxQueueType`, and `queueQUEUE_IS_MUTEX` as replacements for `pcTail`, `pcHead`, and `NULL`, respectively. These macros provide alternative names to improve code readability when using the `Queue_t` structure.

```
/* Semaphores do not actually store or copy data, so have an item size of zero. */
#define queueSEMAPHORE_QUEUE_ITEM_LENGTH ( ( UBaseType_t ) 0 )
#define queueMUTEX_GIVE_BLOCK_TIME       ( ( TickType_t ) 0U )
```

These lines define constants related to semaphores. Since semaphores do not store or copy data, their item size is set to zero (`queueSEMAPHORE_QUEUE_ITEM_LENGTH`). Additionally, `queueMUTEX_GIVE_BLOCK_TIME` is set to zero, indicating that the mutex give operation should not block.

```
#if( configUSE_PREEMPTION == 0 )
/* If the cooperative scheduler is being used then a yield should not be performed just because a higher priority task has been woken. */
#define queueYIELD_IF_USING_PREEMPTION()
#else
#define queueYIELD_IF_USING_PREEMPTION() portYIELD_WITHIN_API()
#endif
```

These lines define the macro `queueYIELD_IF_USING_PREEMPTION()` based on the value of the configuration macro `configUSE_PREEMPTION`. If `configUSE_PREEMPTION` is equal to 0 (cooperative scheduler is being used), the macro is defined as an empty statement. Otherwise, it is defined as the function `portYIELD_WITHIN_API()`, which performs a yield if a higher priority task has been woken. This macro is used for yielding within the queue API functions depending on the preemption configuration.

SEMAPHORES

Let's break down the code and explain semaphores in detail:

```
/* Semaphores do not actually store or copy data, so have an item size of zero. */
#define queueSEMAPHORE_QUEUE_ITEM_LENGTH ( ( UBaseType_t ) 0 )
```

This line defines a constant macro `queueSEMAPHORE_QUEUE_ITEM_LENGTH` with a value of zero. It represents the item size of a semaphore queue. Semaphores in FreeRTOS do not actually store or copy data like message queues or buffers. They are used for signaling and synchronization purposes between tasks or co-routines. Therefore, the item size for a semaphore is set to zero to indicate that no data is associated with semaphore operations.

```
#define queueMUTEX_GIVE_BLOCK_TIME       ( ( TickType_t ) 0U )
```

This line defines a constant macro `queueMUTEX_GIVE_BLOCK_TIME` with a value of zero. It represents the block time used in the `xQueueGiveMutexRecursive()` function for mutexes. When a task attempts to give a mutex using `xQueueGiveMutexRecursive()`, it may need to block if the mutex is already taken by another task. However, in this case, the block time is set to zero, indicating that the `xQueueGiveMutexRecursive()` function should not block and return immediately if the mutex is unavailable.

Now, let's discuss semaphores:

Semaphores are synchronization mechanisms used to coordinate and control access to shared resources in a concurrent system. They help manage the flow of execution between tasks or co-routines to prevent race conditions or conflicts.

In FreeRTOS, a semaphore is represented by the `SemaphoreHandle_t` data type. Semaphores can have two types:

1. Binary Semaphores: Binary semaphores can take two states - available (unlocked) or unavailable (locked). They are commonly used for mutual exclusion and synchronization between tasks.

2. Counting Semaphores: Counting semaphores can take multiple states, depending on the count value specified during initialization. They allow multiple tasks to access a shared resource concurrently, up to the specified count.

Semaphores provide two main operations:

1. Taking (Waiting) a Semaphore: A task or co-routine can take a semaphore using the `xSemaphoreTake()` function. If the semaphore is available, the task continues execution. Otherwise, it blocks until the semaphore becomes available.
2. Giving (Releasing) a Semaphore: A task or co-routine can give a semaphore back using the `xSemaphoreGive()` function. This operation releases the semaphore, making it available for other tasks or co-routines waiting to take it.

Semaphores are often used for resource sharing, task synchronization, and signaling between tasks or co-routines in FreeRTOS. They provide a simple and effective way to control access to shared resources and coordinate the execution flow in a concurrent system.

MUTEX

A mutex (short for mutual exclusion) is a synchronization mechanism used to protect shared resources in a multi-threaded or multi-tasking environment. It allows only one thread or task to access the protected resource at a time, ensuring exclusive access and preventing concurrent access that may lead to race conditions or data corruption.

In FreeRTOS, a mutex is represented by the `QueueHandle_t` data type, and it is typically implemented as a type of counting semaphore with a maximum count of 1. A mutex can be in one of two states: locked or unlocked.

When a task or co-routine wants to access a shared resource protected by a mutex, it must first acquire (take) the mutex. If the mutex is currently unlocked, the task acquires the mutex and continues its execution. If the mutex is locked by another task, the task attempting to acquire the mutex will block until the mutex becomes available.

While a task holds a mutex, it has exclusive access to the protected resource. Other tasks or co-routines attempting to acquire the same mutex will block until it is released by the current holder.

Once a task finishes using the shared resource, it releases (gives) the mutex, allowing other tasks to acquire it. The mutex then transitions from the locked state to the unlocked state.

Mutexes provide a higher level of protection than semaphores in situations where exclusive access to a resource is required. They help prevent data races, ensure data integrity, and facilitate safe and controlled access to shared resources in a concurrent system.

Mutexes are commonly used to protect critical sections of code, shared data structures, or peripherals where mutual exclusion is essential to maintain system integrity and avoid conflicts between tasks or threads.

Let's break down the code and explain it in detail:

```
#if( configUSE_PREEMPTION == 0 )
    /* If the cooperative scheduler is being used then a yield should not be performed just because a higher priority task has been wo
ken. */
    #define queueYIELD_IF_USING_PREEMPTION()
#else
    #define queueYIELD_IF_USING_PREEMPTION() portYIELD_WITHIN_API()
#endif
```

The code defines a macro `queueYIELD_IF_USING_PREEMPTION()` based on the value of the configuration macro `configUSE_PREEMPTION`. Here's the explanation:

1. `configUSE_PREEMPTION` is a configuration setting that determines whether FreeRTOS uses preemption or not. If `configUSE_PREEMPTION` is set to 0, it means the cooperative scheduler is being used. In the cooperative scheduler, tasks voluntarily yield execution to other tasks, and preemption does not occur.
2. If `configUSE_PREEMPTION` is 0 (cooperative scheduler is being used), the code within the `#if` block is executed:

   ```
   /* If the cooperative scheduler is being used then a yield should not be performed just because a higher priority task has been wo
   ken. */
   #define queueYIELD_IF_USING_PREEMPTION()
   ```

```
```

In this case, the macro `queueYIELD_IF_USING_PREEMPTION()` is defined as an empty statement. It means that if the cooperative scheduler is being used, no yield operation should be performed even if a higher priority task has been woken up. This is because in the cooperative scheduler, tasks yield execution voluntarily, and the decision to yield is not based on task priorities.

3. If `configUSE_PREEMPTION` is not 0 (preemptive scheduler is being used), the code within the `#else` block is executed:

```
#define queueYIELD_IF_USING_PREEMPTION() portYIELD_WITHIN_API()
```

In this case, the macro `queueYIELD_IF_USING_PREEMPTION()` is defined as the `portYIELD_WITHIN_API()` function call. The `portYIELD_WITHIN_API()` function is a FreeRTOS API function that performs a yield operation within an API function. This means that if the preemptive scheduler is being used, a yield operation can be performed if a higher priority task has been woken up.

In summary, the purpose of this code is to define a macro that performs a yield operation based on the configuration setting `configUSE_PREEMPTION`. If the cooperative scheduler is being used, the macro is defined as an empty statement, and if the preemptive scheduler is being used, the macro is defined as the `portYIELD_WITHIN_API()` function call to allow for yielding when a higher priority task has been woken up.

STRUCTURE

Let's break down the code and explain each line of the `typedef struct QueueDefinition`:

```
typedef struct QueueDefinition
{
  int8_t *pcHead;
  int8_t *pcTail;
  int8_t *pcWriteTo;

  union
  {
    int8_t *pcReadFrom;
    UBaseType_t uxRecursiveCallCount;
  } u;

  List_t xTasksWaitingToSend;
  List_t xTasksWaitingToReceive;

  volatile UBaseType_t uxMessagesWaiting;
  UBaseType_t uxLength;
  UBaseType_t uxItemSize;

  volatile BaseType_t xRxLock;
  volatile BaseType_t xTxLock;

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

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

This code defines a structure named `QueueDefinition` using the `typedef` keyword, which allows it to be referred to as `xQUEUE`.

- `int8_t *pcHead;`, `int8_t *pcTail;`, `int8_t *pcWriteTo;`: These pointers indicate the positions within the queue storage area. `pcHead` points to the beginning of the queue, `pcTail` points to the byte at the end (used as a marker), and `pcWriteTo` points to the next free position in the storage area.
- `union`: This union allows the structure to be used for different purposes depending on context. It either holds `pcReadFrom`, pointing to the last read position in a queue, or `uxRecursiveCallCount`, which maintains the count of recursive mutex calls.

- `List_t xTasksWaitingToSend;` and `List_t xTasksWaitingToReceive;`: These are lists of tasks that are blocked waiting to send or receive from the queue, stored in priority order.
- `volatile UBaseType_t uxMessagesWaiting;`: This variable keeps track of the number of items currently in the queue.
- `UBaseType_t uxLength;`: Represents the length of the queue, defined as the number of items it can hold, not the number of bytes.
- `UBaseType_t uxItemSize;`: Represents the size of each item that the queue can hold.
- `volatile BaseType_t xRxLock;` and `volatile BaseType_t xTxLock;`: These variables store the number of items received and transmitted while the queue is locked. They are set to `queueUNLOCKED` when the queue is not locked.
- `uxQueueNumber` and `ucQueueType`: These fields are conditionally included if the trace facility is enabled (`configUSE_TRACE_FACILITY == 1`). `uxQueueNumber` is an identifier for the queue, and `ucQueueType` indicates the type of the queue.
- `struct QueueDefinition *pxQueueSetContainer;`: This field is conditionally included if queue sets are enabled (`configUSE_QUEUE_SETS == 1`). It points to the container queue set that holds the queue.

This structure represents the definition of a queue in FreeRTOS, including various members necessary for managing the queue, tracking waiting tasks, maintaining item counts, and supporting different queue functionalities based on configuration options.

Let's go through the code line by line:

```
typedef xQUEUE Queue_t;
```

This line defines `Queue_t` as a typedef for `xQUEUE`. It is used to provide a more user-friendly name for the `xQUEUE` structure, especially for compatibility with older kernel-aware debuggers.

```
#if ( configQUEUE_REGISTRY_SIZE > 0 )
```

This line checks if the queue registry feature is enabled. The queue registry is an optional component that allows kernel-aware debuggers to locate queue structures.

```
typedef struct QUEUE_REGISTRY_ITEM
{
  const char *pcQueueName;
  QueueHandle_t xHandle;
} xQueueRegistryItem;
```

This code defines the structure `QUEUE_REGISTRY_ITEM`, which represents an entry in the queue registry. It contains two members: `pcQueueName`, which is a pointer to a string representing the name of the queue, and `xHandle`, which is the handle of the queue.

```
typedef xQueueRegistryItem QueueRegistryItem_t;
```

This line typedefs `QueueRegistryItem_t` as a typedef for `xQueueRegistryItem`. It is used to provide compatibility with older kernel-aware debuggers.

```
QueueRegistryItem_t xQueueRegistry[ configQUEUE_REGISTRY_SIZE ];
```

If the queue registry is enabled, this line declares an array `xQueueRegistry` of `QueueRegistryItem_t` structures. It is used to store the queue registry entries, where each entry represents a queue.

```
static void prvUnlockQueue( Queue_t * const pxQueue ) PRIVILEGED_FUNCTION;
```

This line declares a static function `prvUnlockQueue` that is used to unlock a queue previously locked by `prvLockQueue`. Unlocking a queue allows tasks to be unblocked. This function takes a pointer to `Queue_t` as its argument.

```
static BaseType_t prvIsQueueEmpty( const Queue_t *pxQueue ) PRIVILEGED_FUNCTION;
```

This line declares a static function `prvIsQueueEmpty` that is used to determine if a queue is empty. It returns `pdTRUE` if the queue contains no items, and `pdFALSE` otherwise. The function takes a constant pointer to `Queue_t` as its argument.

```
static BaseType_t prvIsQueueFull( const Queue_t *pxQueue ) PRIVILEGED_FUNCTION;
```

This line declares a static function `prvIsQueueFull` that is used to determine if a queue is full. It returns `pdTRUE` if there is no space in the queue, and `pdFALSE` otherwise. The function takes a constant pointer to `Queue_t` as its argument.

```
static BaseType_t prvCopyDataToQueue( Queue_t * const pxQueue, const void *pvItemToQueue, const BaseType_t xPosition ) PRIVILEGED_FUNC
TION;
```

This line declares a static function `prvCopyDataToQueue` that is used to copy an item into the queue. The function takes a pointer to `Queue_t`, a pointer to the item to be queued (`pvItemToQueue`), and a position (`xPosition`) indicating whether the item should be added to the front or back of the queue. It returns `pdPASS` if the item was successfully copied to the queue, and `errQUEUE_FULL` if the queue is full.

```
static void prvCopyDataFromQueue( Queue_t * const pxQueue, void * const pvBuffer ) PRIVILEGED_FUNCTION;
```

This line declares a static function `prvCopyDataFromQueue` that is used to copy an item out of the queue. The function takes a pointer to `Queue_t` and a pointer (`pvBuffer`) to

the destination buffer where the item will be copied.

```
#if ( configUSE_QUEUE_SETS == 1 )
  static BaseType_t prvNotifyQueueSetContainer( const Queue_t * const pxQueue, const BaseType_t xCopyPosition ) PRIVILEGED_FUNCTION;
#endif
```

This block conditionally compiles the `prvNotifyQueueSetContainer` function if queue sets are enabled (`configUSE_QUEUE_SETS == 1`). This function is used to notify the queue set container that a queue contains data.

```
#define prvLockQueue( pxQueue )             \\
  taskENTER_CRITICAL();               \\
  {                           \\
    if( ( pxQueue )->xRxLock == queueUNLOCKED )   \\
    {                         \\
      ( pxQueue )->xRxLock = queueLOCKED_UNMODIFIED;  \\
    }                         \\
    if( ( pxQueue )->xTxLock == queueUNLOCKED )   \\
    {                         \\
      ( pxQueue )->xTxLock = queueLOCKED_UNMODIFIED;  \\
    }                         \\
  }                           \\
  taskEXIT_CRITICAL()
```

This macro is used to lock a queue. It is implemented using a critical section to ensure exclusive access. The macro sets the `xRxLock` and `xTxLock` members of the queue to `queueLOCKED_UNMODIFIED` if they were previously unlocked (`queueUNLOCKED`).

The code you provided consists of function and macro definitions related to queue management, locking/unlocking, and utility functions used in FreeRTOS. These functions and macros are essential for the operation of the queue functionality within the FreeRTOS kernel.