FreeRTOS

A real time operating system for embedded systems

QUEUE MANAGEMENT

Queue Management

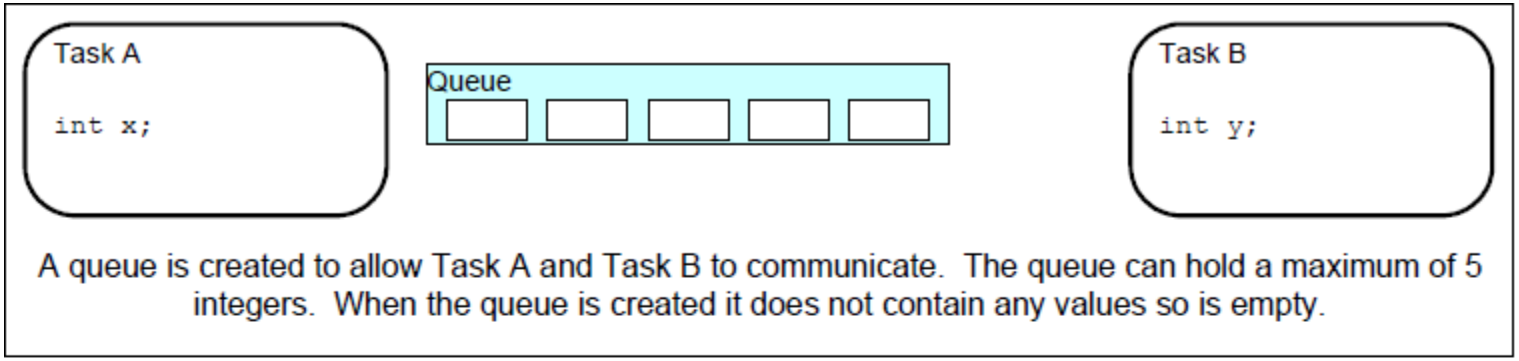
- FreeRTOS applications are structured as a set of independent tasks
  - Each task is effectively a mini program in its own right.
  - It will have to communicate with each other to collectively provide useful system functionality.
- Queue is the underlying primitive
  - Be used for communication and synchronization mechanisms in FreeRTOS.

Queue: Task-to-task communication

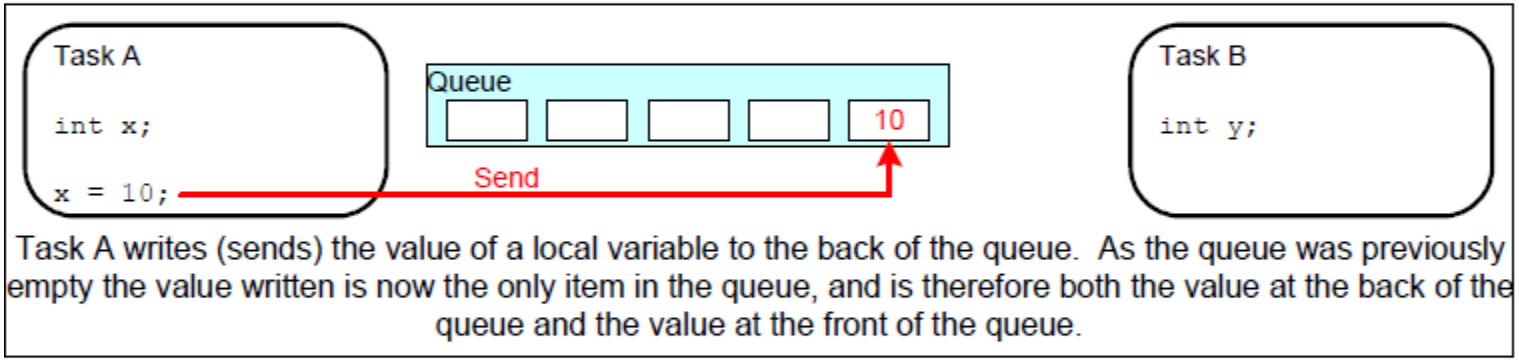
- Scope
  - How to create a queue
  - How a queue manages the data it contains
  - How to send data to a queue
  - How to receive data from a queue
  - What it means to block on a queue
  - How to block on multiple queues.
  - How to overwrite data in a queue.
  - How to clear a queue.
  - The effect of task priorities when writing to and reading from a queue

Queue Characteristics – Data storage

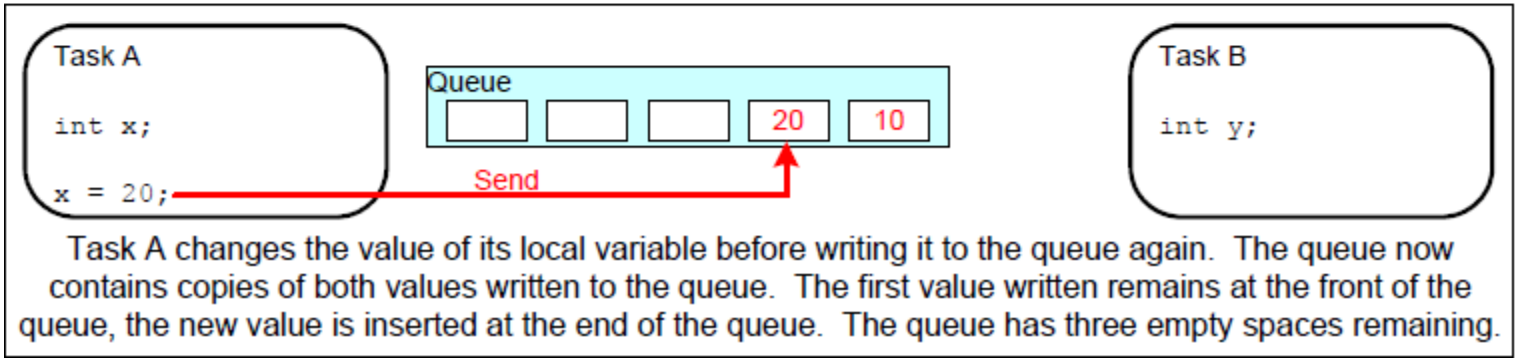
- A queue can hold a finite number of fixed size data items.
  - Normally, used as FIFO buffers where data is written to the end of the queue and removed from the front of the queue.
  - Also possible to write to the front of a queue and to overwrite data that is already at the front of a queue..
  - Writing data to a queue causes a byte-for-byte copy of the data to be stored in the queue itself.
  - Reading data from a queue causes the copy of the data to be removed from the queue.



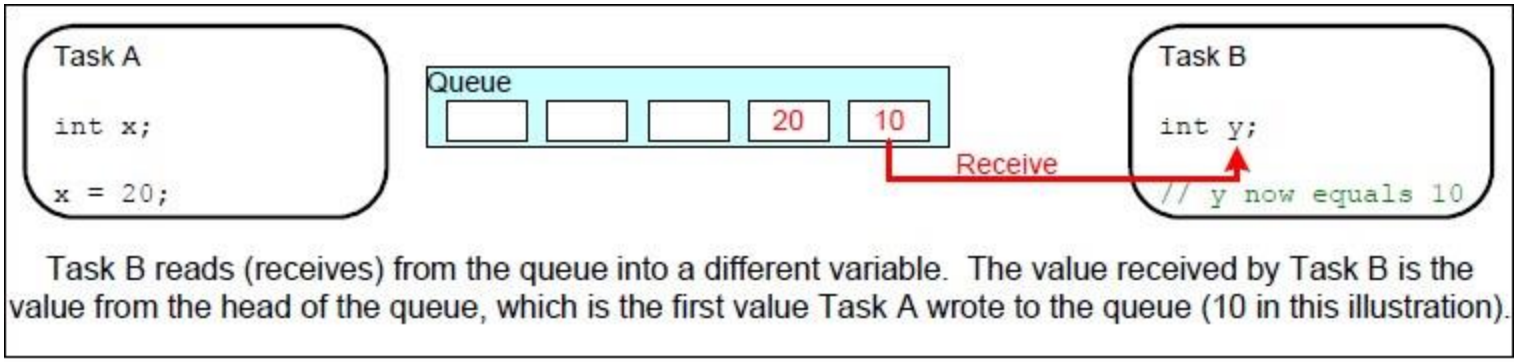
A queue is created to allow Task A and Task B to communicate. The queue can hold a maximum of 5 integers. When the queue is created it does not contain any values so is empty.



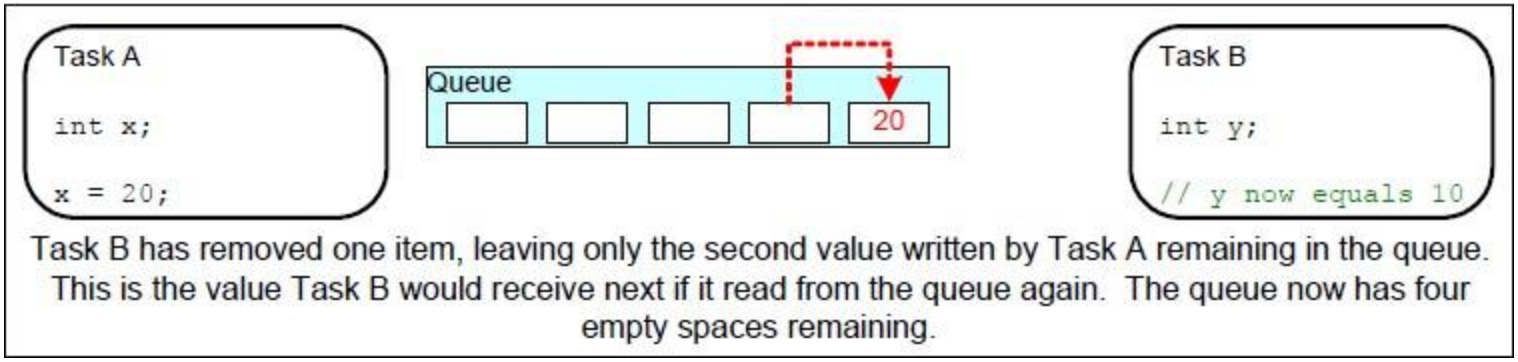
Task A writes (sends) the value of a local variable to the back of the queue. As the queue was previously empty the value written is now the only item in the queue, and is therefore both the value at the back of the queue and the value at the front of the queue.



Task A changes the value of its local variable before writing it to the queue again. The queue now contains copies of both values written to the queue. The first value written remains at the front of the queue, the new value is inserted at the end of the queue. The queue has three empty spaces remaining.



Task B reads (receives) from the queue into a different variable. The value received by Task B is the value from the head of the queue, which is the first value Task A wrote to the queue (10 in this illustration).



Task B has removed one item, leaving only the second value written by Task A remaining in the queue. This is the value Task B would receive next if it read from the queue again. The queue now has four empty spaces remaining.

- Queues are objects in their own right
  - Not owned by or assigned to any particular task
  - Any number of tasks can write to the same queue and any number of tasks can read from the same queue.
  - Very common to have multiple writers, but very rare to have multiple readers.

Blocking on Queue Reads

- When a task attempts to read from a queue it can optionally specify a ‘block’ time
  - The maximum time that the task should be kept in the Blocked state to wait for data to be available from the queue, should the queue already be empty.
  - It is automatically moved to the Ready state when another task or interrupt places data into the queue.
  - It will also be moved automatically from the Blocked state to the Ready state if the specified block time expires before data becomes available.
  - Queues can have multiple readers, so it is possible for a single queue to have more than one task blocked on it waiting for data. When this is the case:

Blocking on Queue Reads

- Only one task will be unblocked when data becomes available.
  - Queue can have multiple readers.
    - So, it is possible for a single queue to have more than one task blocked on it waiting for data.
  - The task that is unblocked will always be the highest priority task that is waiting for data.
  - If the blocked tasks have equal priority, the task that has been waiting for data the longest will be unblocked.

Blocking on Queue Writes

- Just as it can when reading from a queue, A task can optionally specify a ‘block’ time when writing to a queue.
  - The maximum time that task should be held in the Blocked state to wait for space to be available on the queue, should the queue already be full.

Blocking on Queue Writes

- Queue can have multiple writers.
  - It is possible for a full queue to have more than one task blocked on it waiting to complete a send operation.
- Only one task will be unblocked when **space** on the queue becomes available.
  - The task that is unblocked will always be the highest priority task that is waiting for **space**.
  - If the blocked tasks have equal priority, the task that has been waiting for **space** the longest will be unblocked.

Using a Queue

- A queue must be explicitly created before it can be used.
  - FreeRTOS allocates RAM from the heap when a queue is created.
  - RAM holds both the queue data structures and the items that are contained in the queue.

- xQueueCreate() API Function
  - Be used to create a queue and returns an xQueueHandle to reference the queue it creates.

- Function Prototype

```
QueueHandle_t  xQueueCreate(
        UBaseType_t uxQueueLength,
        UBaseType_t uxItemSize );
```

uxQueueLength	The maximum number of items the queue can hold at any one time.
uxItemSize	The size, in bytes, required to hold each item in the queue. Items are queued by copy, not by reference, so this is the number of bytes that will be copied for each queued item. Each item in the queue must be the same size.

Returns:

If the queue is created successfully then a handle to the created queue is returned. If the memory required to create the queue could not be allocated then NULL is returned.

xQueueSendToBack() and xQueueSendToFront() API Functions

- xQueueSendToBack()
  - Be equivalent to xQueueSend()
  - Be used to send data to the back (tail) of a queue
- xQueueSendToFront()
  - Be used to send data to the front (head) of a queue
- Please note, never call these two API functions from an interrupt service routine (ISR).
  - Interrupt-safe versions will be used in their place and described in next chapter.

- Function prototypes

```
BaseType_t  xQueueSendToBack (
                    QueueHandle_t       xQueue,
                    const void * pvItemToQueue,
                    TickType_t  xTicksToWait);

BaseType_t  xQueueSendToFront(
                    QueueHandle_t  xQueue,
                    const void * pvItemToQueue,
                    TickType_t  xTicksToWait);
```

  - xQueue: The handle of the queue to which the data is being send (written). It will have been returned from the call to xQueueCreate() used to create the queue.

- pvItemToQueue: a pointer to the data to be copied into the queue.
  - The size of each item that the queue can hold is set when the queue is created, so the data will be copied from *pvItemToQueue* into the queue storage area.
- xTicksToWait: the maximum amount of time the task should remain in the Blocked state to wait for the space to become available on the queue, should the queue already be full.
  - if xTicksToWait is zero, both APIs will return immediately in case the queue is already full.
  - The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The constant portTICK_PERIOD_MS can be used to convert a time specified in MS into ticks.

- Returned value: two possible return values.
  - pdTRUE will be returned if data was successfully sent to the queue.
    - If a block time was specified, it is possible that the calling task was placed in the Blocked state to wait for another task or interrupt to make room in the queue, before the function returned,
    - Data was successfully written to the queue before the block time expired.
  - errQUEUE_FULL will be returned if data could not be written to the queue as the queue was already full.
    - In a similar scenario that a block time was specified, but it expired before space becomes available in the queue.

xQueueReceive() and xQueuePeek() API Function

- xQueueReceive()
  - Be used to receive (consume) an item from a queue. The item received is removed from the queue.
- xQueuePeek()
  - Be used receive an item from the queue without the item being removed from the queue.
  - Receives the item from the head of the queue.
- Please note, never call these two API functions from an interrupt service routine (ISR).

•Function prototypes

```
BaseType_t   xQueueReceive (
                    QueueHandle_t  xQueue,
                    const void  *pvBuffer,
                    TickType_t xTicksToWait);

BaseType_t   xQueuePeek(
                    QueueHandle_t  xQueue,
                    const void * pvBuffer,
                    TickType_t  xTicksToWait);
```

– xQueue: The handle of the queue from which the data is being received (read). It will have been returned from the call to xQueueCreate().

- pvBuffer: a pointer to the memory into which the received data will be copied.
  - The memory pointed to by pvBuffer must be at least large enough to hold the data item held by the queue.
- xTicksToWait: the maximum amount of time the task should remain in the Blocked state to wait for the data to become available on the queue, should the queue already be empty.
  - if xTicksToWait is zero, both APIs will return immediately in case the queue is already empty.
  - The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The constant portTICK_RATE_MS can be used to convert a time specified in MS into ticks.

- Returned value: two possible return values.
  - pdTRUE will be returned if data was successfully read from the queue.
    - If a block time was not zero, it is possible that the calling task was placed in the Blocked state to wait for another task or interrupt to send the data to the queue before the function is returned,
    - data was successfully read from the queue before the block time expired.
  - errQUEUE_EMPTY will be returned if data could not be read from the queue as the queue was already empty.
    - In a similar scenario that a block time was not zero, but it expired before data was sent.

uxQueueMessageWaiting() API Function

- Be used to query the number of items that are currently in a queue.
- Prototype

```
UBaseType_t uxQueueMessagesWaiting (
                                QueueHandle_t  xQueue);
```

  - Returned value: the number of items that the queue being queried is currently holding. If zero is returned, the queue is empty.

Example 10. Blocking when receiving from a queue

- To demonstrate
  - a queue being created,
    - Hold data items of type *long*
  - data being sent to the queue from multiple tasks,
    - Sending tasks do not specify a block time, lower priority than receiving task.
- And data being received from the queue
    - Receiving task specifies a block time 100ms
- So, queue never contains more than one item
  - Once data is sent to the queue, the receiving task will unblock, pre-empt the sending tasks, and remove the data – leaving the queue empty once again.

```
static void vSenderTask( void *pvParameters )
{
long lValueToSend;
BaseType_t  xStatus;

lValueToSend = ( long ) pvParameters;
  Serial.print("lValueToSend = ");
  Serial.println(lValueToSend);

  /* As per most tasks, this task is implemented within an infinite loop. */
  for( ;; )
  {
    xStatus = xQueueSendToBack( xQueue, &lValueToSend, 0 );

    if( xStatus != pdPASS )
    {
       Serial.print( "Could not send to the queue.\r\n" );
    }

    /* Allow the other sender task to execute. */
    //taskYIELD();
  }
}
```

```
static void vReceiverTask( void *pvParameters )
{
long lReceivedValue;
BaseType_t  xStatus;
const TickType_t xTicksToWait = 100 / portTICK_PERIOD_MS;
for( ;; )
  {
    if( uxQueueMessagesWaiting( xQueue ) != 0 )
    {
      Serial.print( "Queue should have been empty!\r\n" );
    }
    xStatus = xQueueReceive( xQueue, &lReceivedValue, xTicksToWait );

    if( xStatus == pdPASS )
    {
      Serial.print("Received = ");
      Serial.println(lReceivedValue);
    }
    else
    {
       Serial.print( "Could not receive from the queue.\r\n" );
    }
  }
}
```

- vSenderTask() does not specify a block time.
  - continuously writing to the queue

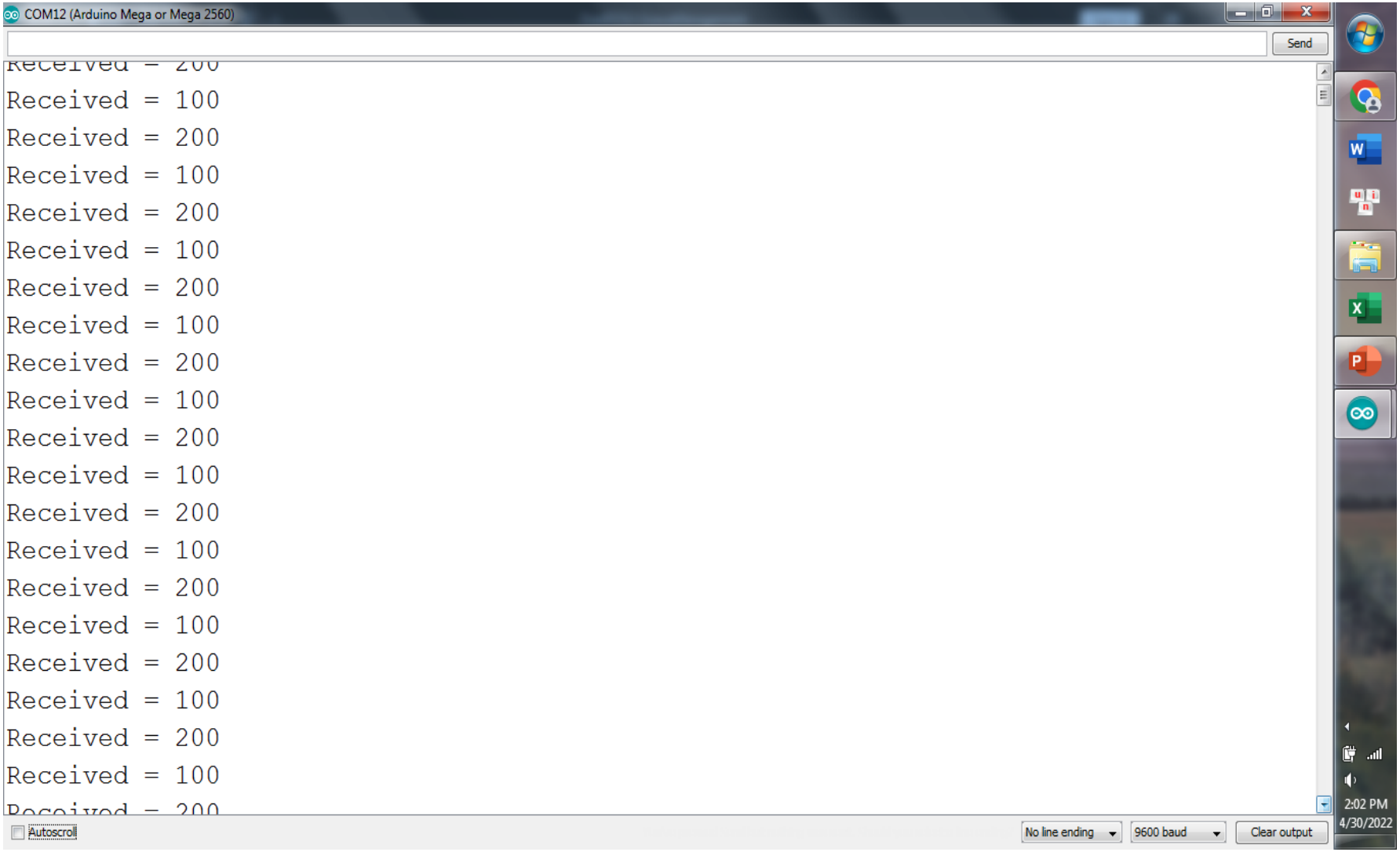
```
xStatus = xQueueSendToBack(xQueue, pvParameters, 0);
If (xStatus != pdPASS) {
        Serial.print("Could not send to the queue.\n"); }
//taskYIELD();
```

- vReceiverTask() specifies a block time 100ms.
  - Enter the Blocked state to wait for data to be available, leaves it when either data is available on the queue, or 100ms expires, which should never occur.

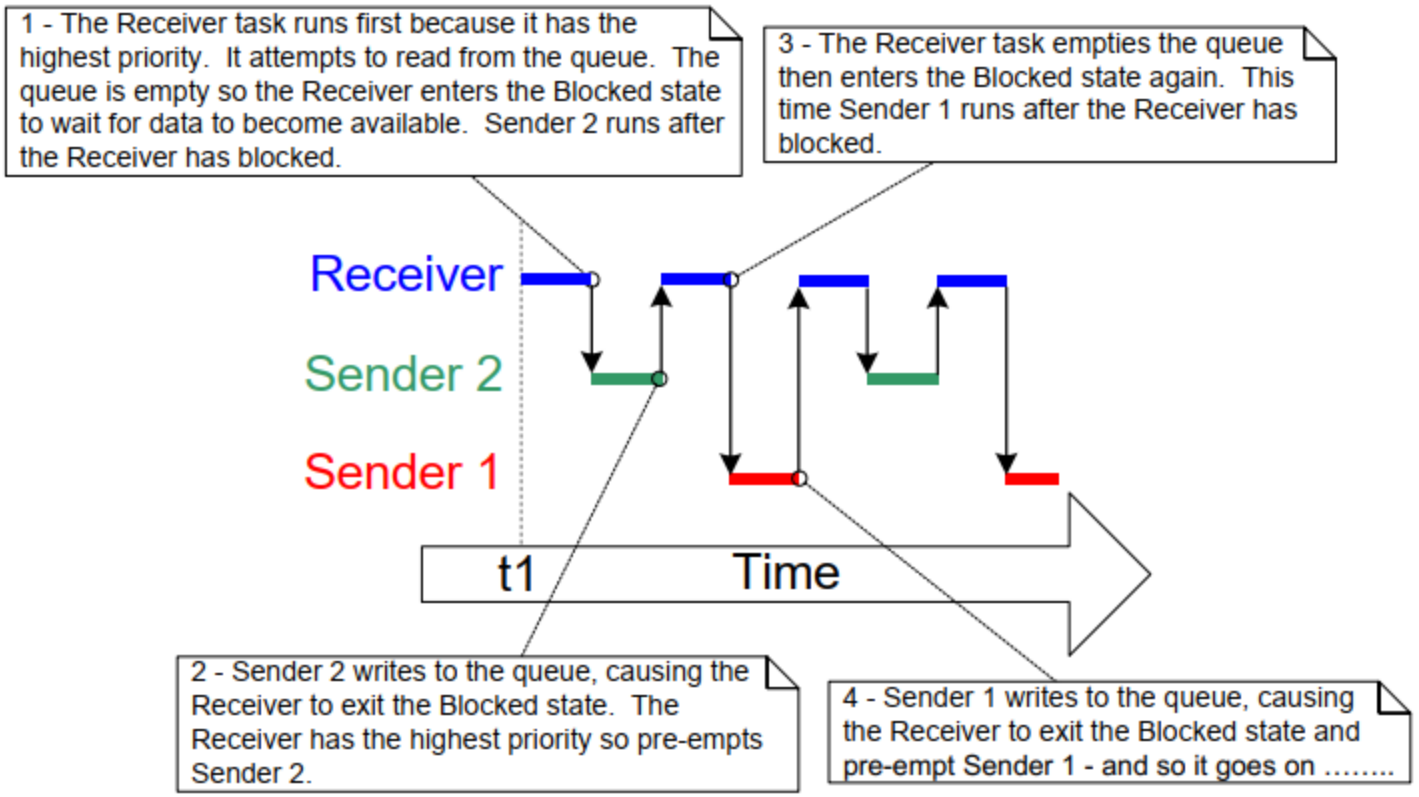
```
xStatus = xQueueReceive(xQueue,
        &xReceivedValue, 100/portTICK_RATE_MS);
        if (xStatus == pdPASS) {  Serial.print("Received = ");
        Serial.println(lReceivedValue );
        }
```

```
/* Declare a variable of type QueueHandle_t.  This is used to store the queue
that is accessed by all three tasks. */
QueueHandle_t xQueue;
void setup( void )
{
  Serial.begin(9600);

    /* The queue is created to hold a maximum of 5 long values. */
    xQueue = xQueueCreate( 5 , sizeof( long ) );

  if( xQueue != NULL )
```

```
{

   xTaskCreate( vSenderTask, "Sender1", 1000, ( void * ) 100, 1, NULL );
   xTaskCreate( vSenderTask, "Sender2", 1000, ( void * ) 200, 1, NULL );

   /* Create the task that will read from the queue.  The task is created with
   priority 2, so above the priority of the sender tasks. */
   xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 2, NULL );

   /* Start the scheduler so the created tasks start executing. */
   vTaskStartScheduler();
  }
  else
  {
   /* The queue could not be created. */
   Serial.println("The queue could not be created.");
  }

for( ;; );
//  return 0;
}
```


COM12 (Arduino Mega or Mega 2560)
Send
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
Autoscroll
No line ending
9600 baud
Clear output
2:02 PM
4/30/2022

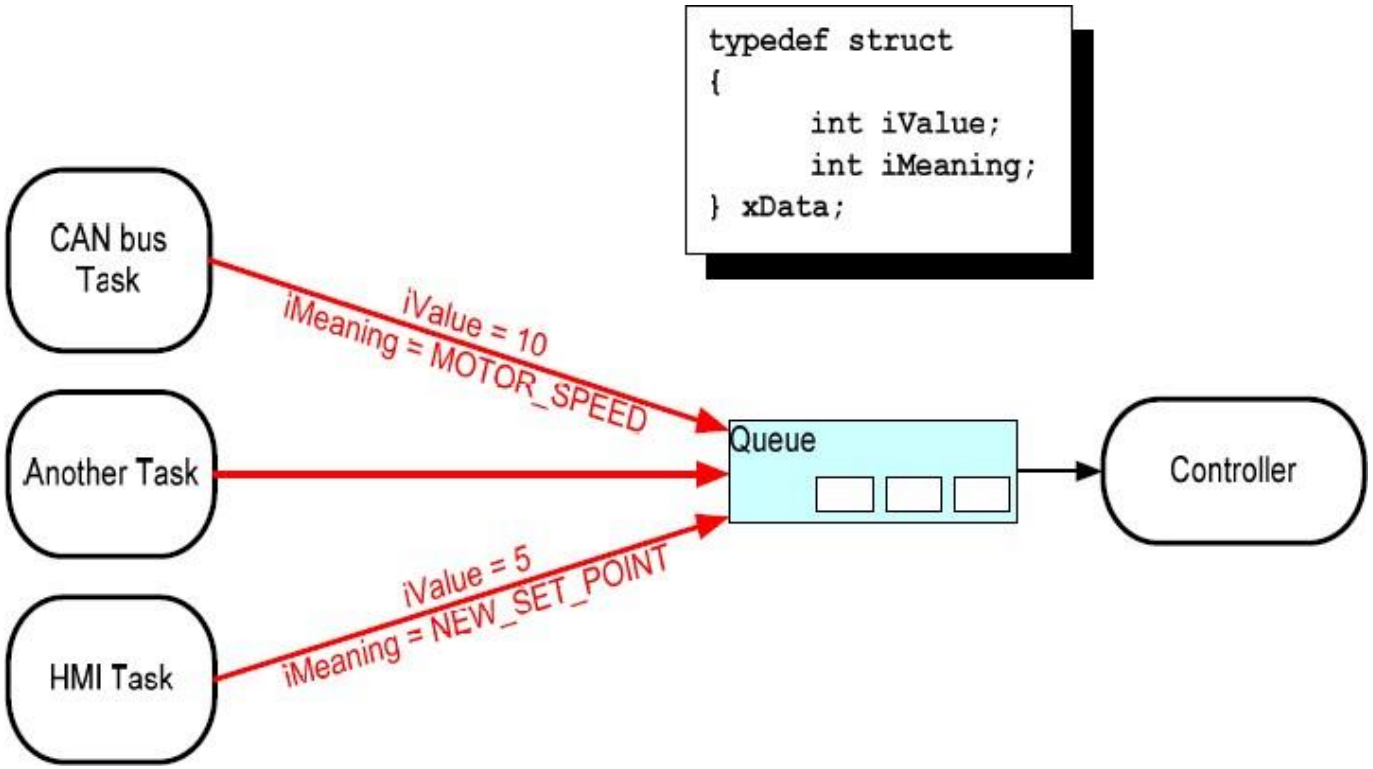
Execution sequence



Receiving Data From Multiple Sources

- It is common for a task to receive data from multiple sources on a single queue.
  - Receiver needs to know the data source to allow it to determine how to process the data.
  - Use the queue to transfer structures which contain both data value and data source, like

```
typedef struct {
  int iValue; // a data value
  int iMeaning; // a code indicating data source
} xData;
```



- Controller task performs the primary system function.
  - React to inputs and changes to the system state communicated to it on the queue.
  - A CAN bus task encapsulates the CAN bus interfacing functionality, like the actual motor speed value.
  - A HMI task encapsulates all the HMI functionality, like the actual new set point value.

Example 11

- Two differences from Example 10
  - Receiving task has a **lower** priority than the sending tasks.
  - The queue is used to pass structures, rather than simple long integers between the tasks.
- The Queue will normally be full because
  - Once the receiving task removes an item from the queue, it is pre-empted by one of the sending tasks which then immediately refills the queue.
  - Then sending tasks re-enters the Blocked state to wait for space to become available on the queue again.

Structure Type

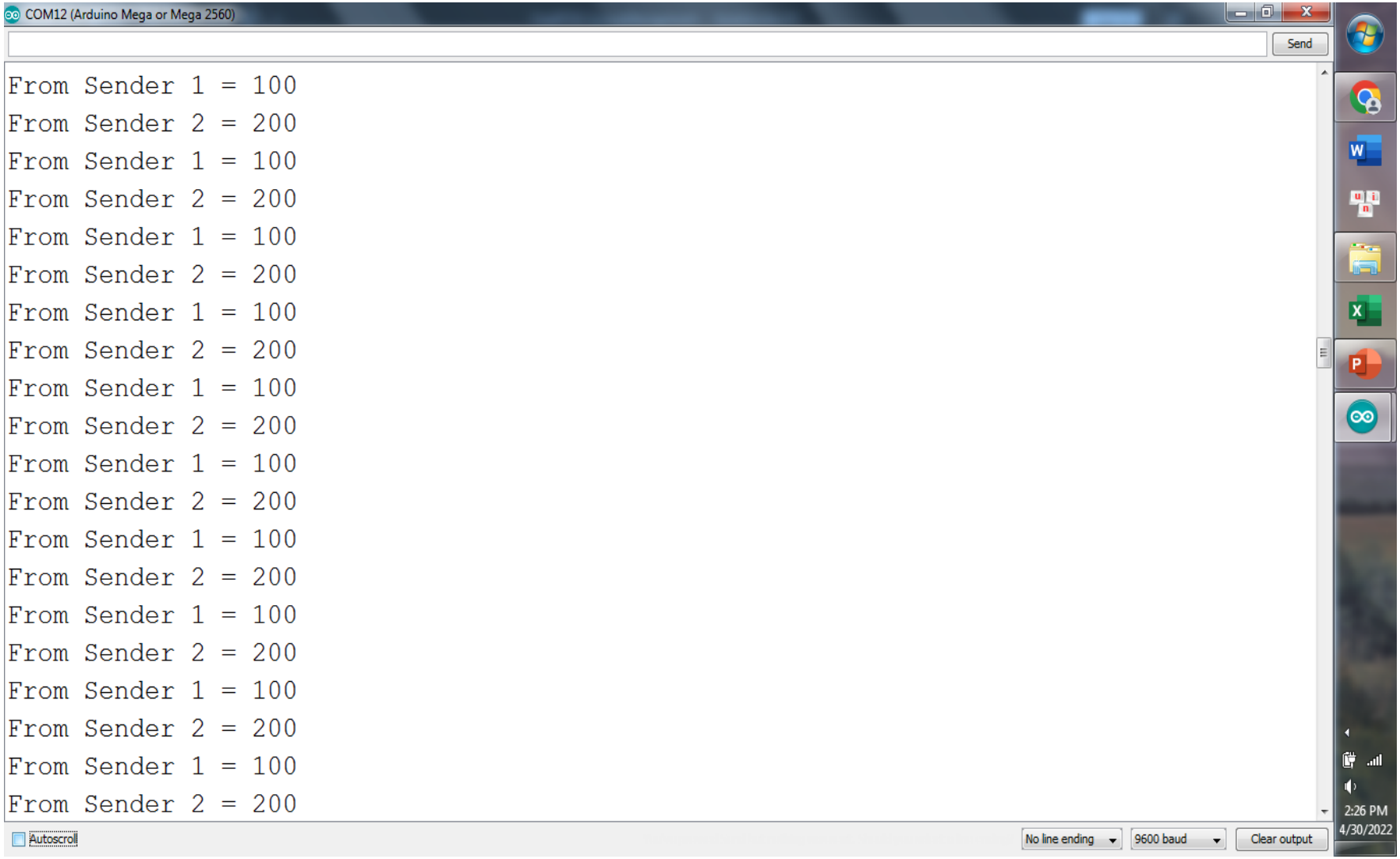
```
/* Define the structure type that will be passed on the queue. */
typedef struct
{
  unsigned char ucValue;
  unsigned char ucSource;
} xData;

/* Declare two variables of type xData that will be passed on the queue. */
static const xData xStructsToSend[ 2 ] =
{
  { 100, mainSENDER_1 }, /* Used by Sender1. */
  { 200, mainSENDER_2 }  /* Used by Sender2. */
};
```

```
static void vSenderTask( void *pvParameters )
{
BaseType_t  xStatus;
const TickType_t xTicksToWait = 100 / portTICK_PERIOD_MS;

  /* As per most tasks, this task is implemented within an infinite loop. */
  for( ;; )
  {
    xStatus = xQueueSendToBack( xQueue, pvParameters, xTicksToWait );

    if( xStatus != pdPASS )
    {
      /* We could not write to the queue because it was full - this must
      be an error as the receiving task should make space in the queue
      as soon as both sending tasks are in the Blocked state. */
      Serial.print( "Could not send to the queue.\r\n" );
    }

    /* Allow the other sender task to execute. */
    taskYIELD();
  }
}
```

```
static void vReceiverTask( void *pvParameters )
{
/* Declare the structure that will hold the values received from the queue. */
xData xReceivedStructure;
BaseType_t  xStatus;

  /* This task is also defined within an infinite loop. */
  for( ;; )
  {
    /* As this task only runs when the sending tasks are in the Blocked state,
    and the sending tasks only block when the queue is full, this task should
    always find the queue to be full.  3 is the queue length. */
    if( uxQueueMessagesWaiting( xQueue ) != 3 )
    {
      Serial.print( "Queue should have been full!\r\n" );
    }
```

```
xStatus = xQueueReceive( xQueue, &xReceivedStructure, 0 );

   if( xStatus == pdPASS )
   {
     /* Data was successfully received from the queue, print out the received
     value and the source of the value. */
     if( xReceivedStructure.ucSource == mainSENDER_1 )
     {
       Serial.print( "From Sender 1 = ");
       Serial.println(xReceivedStructure.ucValue );
     }
     else
     {
       Serial.print( "From Sender 2 = ");
       Serial.println(xReceivedStructure.ucValue );
     }
   }
   else
   {
Serial.print( "Could not receive from the queue.\r\n" );
   }
  }
}
```

```
void setup( void )
{
  Serial.begin(9600);
    /* The queue is created to hold a maximum of 3 structures of type xData. */
    xQueue = xQueueCreate( 3, sizeof( xData ) );

  if( xQueue != NULL )
  {
    xTaskCreate( vSenderTask, "Sender1", 200, ( void * ) &( xStructsToSend[ 0 ] ), 2, NULL );
    xTaskCreate( vSenderTask, "Sender2", 200, ( void * ) &( xStructsToSend[ 1 ] ), 2, NULL );

    xTaskCreate( vReceiverTask, "Receiver", 200, NULL, 1, NULL );

    /* Start the scheduler so the created tasks start executing. */
    vTaskStartScheduler();
  }
  else
  {
    /* The queue could not be created. */
  }

for( ;; );
}
```


COM12 (Arduino Mega or Mega 2560)
Send
From Sender 1 = 100
From Sender 2 = 200
From Sender 1 = 100
From Sender 2 = 200
From Sender 1 = 100
From Sender 2 = 200
From Sender 1 = 100
From Sender 2 = 200
From Sender 1 = 100
From Sender 2 = 200
From Sender 1 = 100
From Sender 2 = 200
From Sender 1 = 100
From Sender 2 = 200
From Sender 1 = 100
From Sender 2 = 200
From Sender 1 = 100
From Sender 2 = 200
From Sender 1 = 100
From Sender 2 = 200
Autoscroll
No line ending
9600 baud
Clear output
2:26 PM
4/30/2022

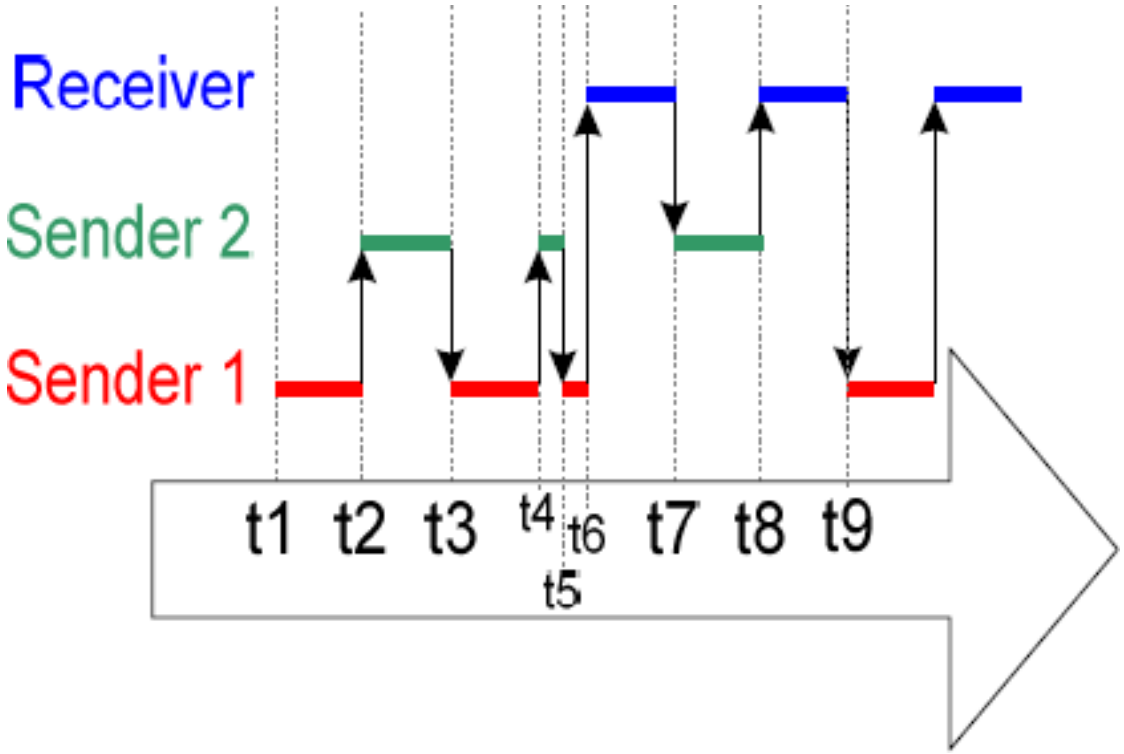
- In vSenderTask(), the sending task specifies a block time of 100ms.
  - So, it enters the Blocked state to wait for space to become available each time the queue becomes full.
  - It leaves the Blocked state when either the space is available on the queue or 100ms expires without space be available (should never expire as receiving task is continuously removing items from the queue).

```
xStatus = xQueueSendToBack(xQueue,
          pvParameters,
          100/portTICK_PERIOD_MS);
If (xStatus != pdPASS) {
      Serial.print("Could not send to the queue.\n");
}
taskYIELD();
```

- vReceiverTask() will run only when both sending tasks are in the Blocked state.
  - Sending tasks will enter the Blocked state only when the queue is full as they have higher priorities.
  - The receiving task will execute only when the queue is already full. -> it always expects to receive data even without a ‘block’ time.

```
xStatus = xQueueReceive(xQueue,
                        &xReceivedStructure, 0);
if (xStatus == pdPASS) {
        // print the data received
}
```

Execution sequence – Sender 1 and 2 have higher priorities than Receiver



Working with large data

- It is not efficient to copy the data itself into and out of the queue byte by byte, when the size of the data being stored in the queue is large.
- It is preferable to use the queue to transfer pointers to the data.
  - More efficient in both processing time and the amount of RAM required to create the queue.
- But, when queuing pointers, extreme care must be taken to ensure that:

The owner of the RAM being pointed to is clearly defined.

- When multiple tasks share memory via a pointer, they do not modify its contents simultaneously, or take any other action that cause the memory contents invalid or inconsistent.
  - Ideally, only the sending task is permitted to access the memory before the pointer is sent to the queue , and only the receiving task is permitted to access the memory after the pointer has been received from the queue.

The RAM being pointed to remains valid.

- If the memory being pointed to was allocated dynamically, exactly one task be responsible for freeing the memory.
- No task should attempt to access the memory after it has been freed.
- A pointer should never be used to access data that has been allocated on a task stack. The data will not be valid after the stack frame has changed.

Create a queue that can hold up to 5 pointers.

```
/* Declare a variable of type QueueHandle_t to hold
the handle of the queue being created. */
QueueHandle_t xPointerQueue;

/* Create a queue that can hold a maximum of 5
pointers, in this case character pointers. */
xPointerQueue = xQueueCreate( 5, sizeof( char * ) );
```

Allocate a buffer, writes a string to the buffer, then sends a pointer to the buffer to the queue.

```
void vStringSendingTask( void *pvParameters )
{
    char *pcStringToSend;
    const size_t xMaxStringLength = 50;
    BaseType_t xStringNumber = 0;

for( ;; )
{
pcStringToSend = ( char * ) prvGetBuffer(xMaxStringLength);
snprintf( pcStringToSend, xMaxStringLength, "String number %d\r\n",
xStringNumber );
xStringNumber++;
xQueueSend( xPointerQueue, &pcStringToSend, portMAX_DELAY );
}
}
```

Receive a pointer to a buffer from the queue, then prints the string contained in the buffer.

```
void vStringReceivingTask( void *pvParameters )
{
    char *pcReceivedString;

    for( ;; )
    {
    xQueueReceive( xPointerQueue,
&pcReceivedString, portMAX_DELAY );
vPrintString( pcReceivedString );

prvReleaseBuffer( pcReceivedString );
    }
}
```

Using a Queue to Send Different Types and Lengths of Data

- Previous sections of this book demonstrated two powerful design patterns; sending structures to a queue, and sending pointers to a queue. Combining those techniques allows a task to use a single queue to receive any data type from any data source. The implementation of the FreeRTOS+TCP TCP/IP stack provides a practical example of how this is achieved.
- The TCP/IP stack, which runs in its own task, must process events from many different sources. Different event types are associated with different types and lengths of data. IPStackEvent_t structures describe all events that occur outside of the TCP/IP task, and are sent to the TCP/IP task on a queue.

Receiving From Multiple Queues – Queue Sets

- Often application designs require a single task to receive data of different sizes, data with different meanings, and data from different sources.
- The previous section demonstrated how to do this in a neat and efficient way using a single queue that receives structures.
- However, sometimes an application's designer is working with constraints that limit their design choices, necessitating the use of a separate queue for some data sources. In such cases a 'queue set' can be used.
- Queue sets allow a task to receive data from more than one queue without the task polling each queue in turn to determine which, if any, contains data.
- A design that uses a queue set to receive data from multiple sources is less neat, and less efficient, than a design that achieves the same functionality using a single queue that receives structures. For that reason, it is recommended to only use queue sets if design constraints make their use absolutely necessary.

Creating a queue set

- Queues sets are referenced by handles, which are variables of type QueueSetHandle_t.
- The xQueueCreateSet() API function creates a queue set and returns a QueueSetHandle_t that references the created queue set.
  QueueSetHandle_t xQueueCreateSet(const UBaseType_t uxEventQueueLength);
- When a queue that is a member of a queue set receives data, the handle of the receiving queue is sent to the queue set. uxEventQueueLength defines the maximum number of queue handles the queue set being created can hold at any one time.

Creating a queue set

- Queue handles are only sent to a queue set when a queue within the set receives data. A queue cannot receive data if it is full, so no queue handles can be sent to the queue set if all the queues in the set are full. Therefore, the maximum number of items the queue set will ever have to hold at one time is the sum of the lengths of every queue in the set.
- Semaphores can also be added to a queue set. For calculating uxEventQueueLength, the length of a binary semaphore is one, the length of a mutex is one, and the length of a counting semaphore is given by the semaphore's maximum count value.
- If a non-NULL value is returned then the queue set was created successfully and the returned value is the handle to the created queue set.

The xQueueAddToSet() API Function

- Add a queue or semaphore to a queue set.

```
BaseType_t xQueueAddToSet(
QueueSetMemberHandle_t xQueueOrSemaphore,
                        QueueSetHandle_t xQueueSet );
```

- xQueueOrSemaphore: The handle of the queue or semaphore that is being added to the queue set.
- xQueueSet: The handle of the queue set to which the queue or semaphore is being added.
- Return value: pdPASS or pdFAIL

The xQueueSelectFromSet() API Function

- Read a queue handle from the queue set.

```
QueueSetMemberHandle_t xQueueSelectFromSet(
QueueSetHandle_t xQueueSet,
 const TickType_t xTicksToWait );
```

- xQueueSet: The handle of the queue set from which a queue handle or semaphore handle is being received (read).
- xTicksToWait: The maximum amount of time the calling task should remain in the Blocked state to wait to receive a queue or semaphore handle from the queue set, if all the queues and semaphore in the set are empty.
- A return value that is not NULL will be the handle of a queue or semaphore that is known to contain data. (If a block time was specified (xTicksToWait was not zero), then it is possible the calling task was placed into the Blocked state to wait for data to become available from a queue or semaphore in the set, but a handle was successfully read from the queue set before the block time expired.)

Example111- Using a Queue Set

- This example creates two sending tasks and one receiving task. The sending tasks send data to the receiving task on two separate queues, one queue for each task.
- The two queues are added to a queue set, and the receiving task reads from the queue set to determine which of the two queues contain data.

```
/* Declare two variables of type QueueHandle_t. Both
queues are added to the same queue set. */
static QueueHandle_t xQueue1 = NULL, xQueue2 = NULL;

/* Declare a variable of type QueueSetHandle_t. This is the
queue set to which the two queues are added. */
static QueueSetHandle_t xQueueSet = NULL;

void setup() {
    Serial.begin(9600);
/* Create the two queues, both of which send character
pointers. The priority of the receiving task is above the
priority of the sending tasks, so the queues will never have
more than one item in them at any one time*/
    xQueue1 = xQueueCreate( 1, sizeof( char * ) );
    xQueue2 = xQueueCreate( 1, sizeof( char * ) );
```

```
/* Create the queue set. Two queues will be added to
the set, each of which can contain 1 item, so the
maximum number of queue handles the  queue set
will ever have to hold at one time is 2 (2 queues
multiplied  by 1 item per queue). */
    xQueueSet = xQueueCreateSet( 1 * 2 );

    /* Add the two queues to the set. */
    xQueueAddToSet( xQueue1, xQueueSet );
    xQueueAddToSet( xQueue2, xQueueSet );
```

```
/* Create the tasks that send to the queues. */
    xTaskCreate( vSenderTask1, "Sender1", 1000,
NULL, 1, NULL );
    xTaskCreate( vSenderTask2, "Sender2", 1000,
NULL, 1, NULL );

    /* Create the task that reads from the queue set to
determine which of the two queues contain data. */
    xTaskCreate( vReceiverTask, "Receiver", 1000,
NULL, 2, NULL );

    /* Start the scheduler so the created tasks start
executing. */
    // vTaskStartScheduler();
}
```

```
void vSenderTask1( void *pvParameters )
{
    const TickType_t xBlockTime = pdMS_TO_TICKS( 100 );
    const char * const pcMessage = "Message from vSenderTask1\r\n";

    /* As per most tasks, this task is implemented within an infinite loop. */

    for( ;; )
    {

        /* Block for 100ms. */
        vTaskDelay( xBlockTime );
```

```
 /* Send this task's string to xQueue1. It is not
necessary to use a block time, even though the
queue can only hold one item. This is  because the
priority of the task that reads from the queue is
higher than the priority of this task; as soon as this
task writes to the queue it will be pre-empted by the
task that reads from the queue, so the queue will
already be empty again by the time the call to
xQueueSend() returns. The block time is set to 0. */
        xQueueSend( xQueue1, &pcMessage, 0 );
    }
}
```

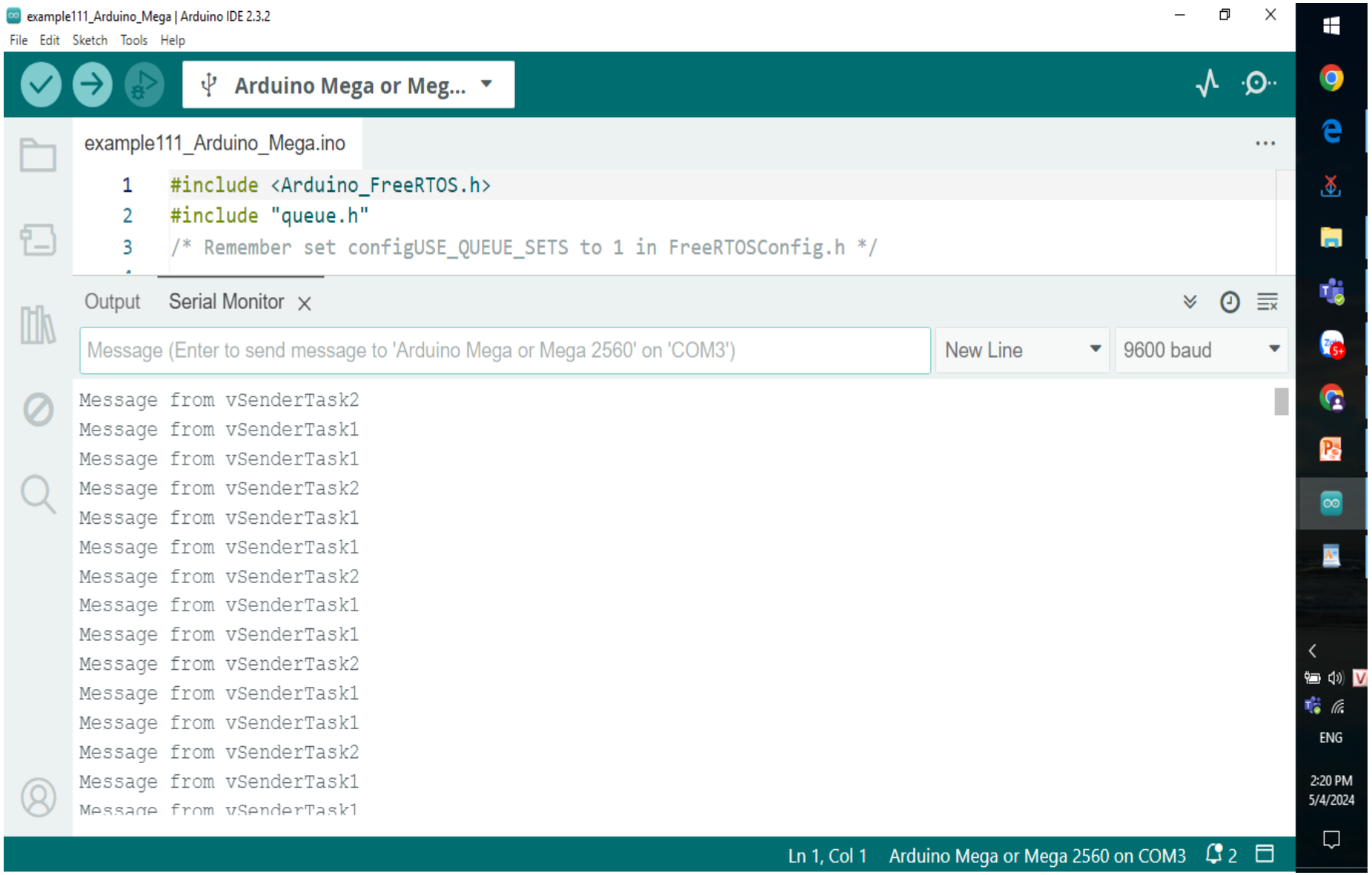
```
void vSenderTask2( void *pvParameters )
{
    const TickType_t xBlockTime = pdMS_TO_TICKS( 200 );
    const char * const pcMessage = "Message from vSenderTask2\r\n";

    /* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {
        /* Block for 200ms. */
        vTaskDelay( xBlockTime );
```

```
/* Send this task's string to xQueue2. It is not
necessary to use a block time, even though the
queue can only hold one item. This is because the
priority of the task that reads from the queue is higher
than the priority of this task; as soon as this task
writes to the queue it will be pre-empted by the task
that reads from the queue, so the queue will already
be empty again by the time the call to xQueueSend()
returns. The block time is set to 0. */
        xQueueSend( xQueue2, &pcMessage, 0 );
    }
}
```

```
void vReceiverTask( void *pvParameters )
{
    QueueHandle_t xQueueThatContainsData;
    char *pcReceivedString;

    /* As per most tasks, this task is implemented
within an infinite loop. */
    for( ;; )
    {
        /* Block on the queue set to wait for one of the
queues in the set to contain data. Cast the
QueueSetMemberHandle_t value returned from
xQueueSelectFromSet() to a QueueHandle_t, as it is
known all the members of the set are queues (the
queue set does not contain any  semaphores). */
```

```
xQueueThatContainsData = ( QueueHandle_t )
xQueueSelectFromSet(xQueueSet, portMAX_DELAY );

        /* An indefinite block time was used when reading
from the queue set, so xQueueSelectFromSet() will not
have returned unless one of the  queues in the set
contained data, and xQueueThatContainsData cannot
be NULL. Read from the queue. It is not necessary to
specify a block time because it is known the queue
contains data. The block time is set to 0. */
        xQueueReceive( xQueueThatContainsData,
&pcReceivedString, 0 );
        /* Print the string received from the queue. */
        Serial.print( pcReceivedString );
    }
}
```

Example111 – Serial monitor



Using a Queue to Create a Mailbox

- There is no consensus on terminology within the embedded community, and 'mailbox' will mean different things in different RTOSes. In this book, the term mailbox is used to refer to a queue that has a length of one. A queue may be described as a mailbox because of the way it is used in the application, rather than because it has a functional difference to a queue:
- A queue is used to send data from one task to another task, or from an interrupt service routine to a task. The sender places an item in the queue, and the receiver removes the item from the queue. The data passes through the queue from the sender to the receiver.
- A mailbox is used to hold data that can be read by any task, or any interrupt service routine. The data does not pass through the mailbox, but instead remains in the mailbox until it is overwritten. The sender overwrites the value in the mailbox. The receiver reads the value from the mailbox, but does not remove the value from the mailbox.

The xQueueOverwrite() API Function

BaseType_t xQueueOverwrite(QueueHandle_t xQueue, const void * pvItemToQueue);

- Like the xQueueSendToBack() API function, the xQueueOverwrite() API function sends data to a queue. Unlike xQueueSendToBack(), if the queue is already full, then xQueueOverwrite() overwrites data that is already in the queue.
- xQueueOverwrite() must only be used with queues that have a length of one. The overwrite mode will always write to the front of the queue and update the front of queue pointer, but it will not update the messages waiting.

The xQueueOverwrite() API Function

BaseType_t xQueueOverwrite(QueueHandle_t xQueue, const void * pvItemToQueue);

- xQueue:The handle of the queue to which the data is being sent (written). The queue handle will have been returned from the call to xQueueCreate() or xQueueCreateStatic() used to create the queue.
- pvItemToQueue: A pointer to the data to be copied into the queue.
- Return value: xQueueOverwrite() writes to the queue even when the queue is full, so pdPASS is the only possible return value.

The xQueuePeek() API Function

```
BaseType_t xQueuePeek( QueueHandle_t xQueue,
                       void * const pvBuffer,
                       TickType_t xTicksToWait );
```

- xQueuePeek() receives (reads) an item from a queue *without* removing the item from the queue. xQueuePeek() receives data from the head of the queue without modifying the data stored in the queue, or the order in which data is stored in the queue.
- *xQueuePeek() has the same function parameters and return value as xQueueReceive().*