

Use of MailBox in FreeRTOS

Targeted competences: Simple use of the mailbox in FreeRTOS

Hardware: STM32F7 Nucleo board

Framework: STM32CubeIDE v1.6.1 from STMicroelectronics

The aim of this document is to show how to use the mailbox concept in FreeRTOS in an example for a STM32 board.

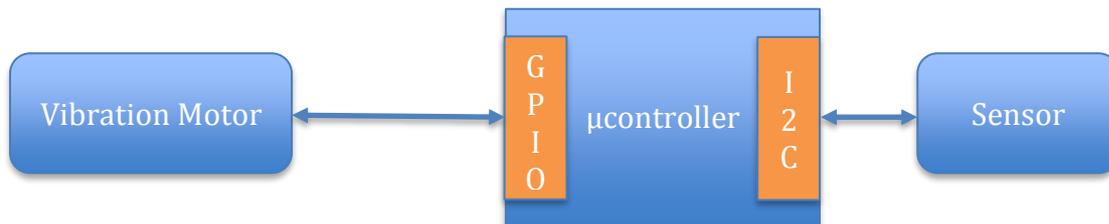


Figure 1: Global view of the system

We develop here a small code able to generate a control signal for a vibration motor inside a task and in function of values sent by another task by using a mailbox.

1. Microcontroller configuration

The first step is to configure the microcontroller. In our case, we use the NUCLEO-F767ZI platform based on a STM32F7 architecture. In order to configure this board we will use the CubeMx software. The first step after choosing the board is to name the project and then start to configure the microcontroller. First, we must use the freeRTOS by choosing him in the middleware tab as in figure 2.

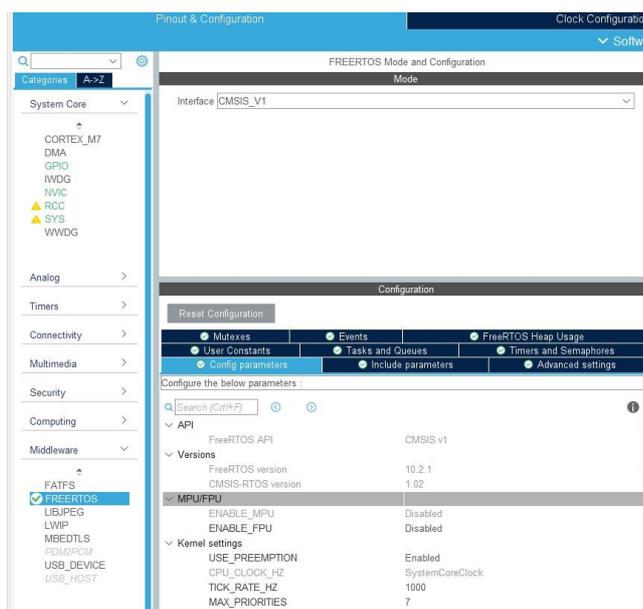


Figure 2: CubeIDE interface

In the FreeRTOS configuration tab, you have nothing to change; use the default configuration. In order to use tasks, we must create them into the Tasks and Queues tab.

Here, I add 2 tasks: one for the production of data (read the sensor data and send them via the mailbox) and one for the consumer of the sensor data. I use the default configuration for each task as you can see in the figure below. I called the tasks respectively ConsoTask that will use the StartConsoTask function and ProducTask that will use the StartProducTask. Each task has a normal priority level.

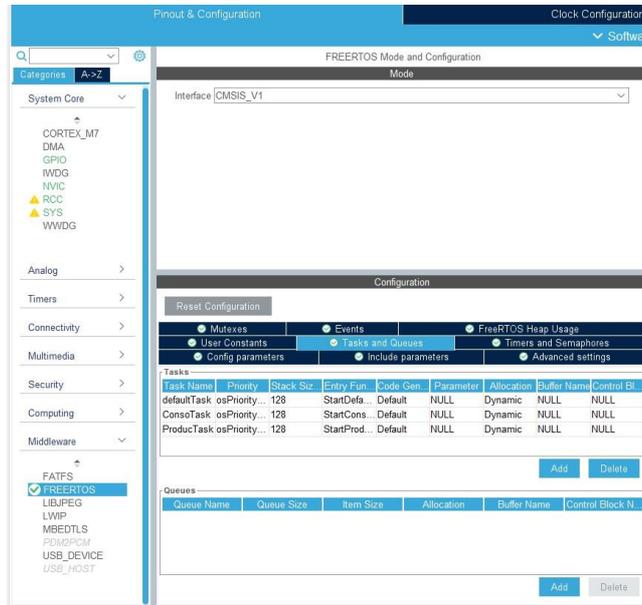


Figure 3: Task configuration tab

The next step is to configure the I²C port in order to use it to connect the sensor to the microcontroller. As you can see in the figure below, I kept the classical (by default) configuration.

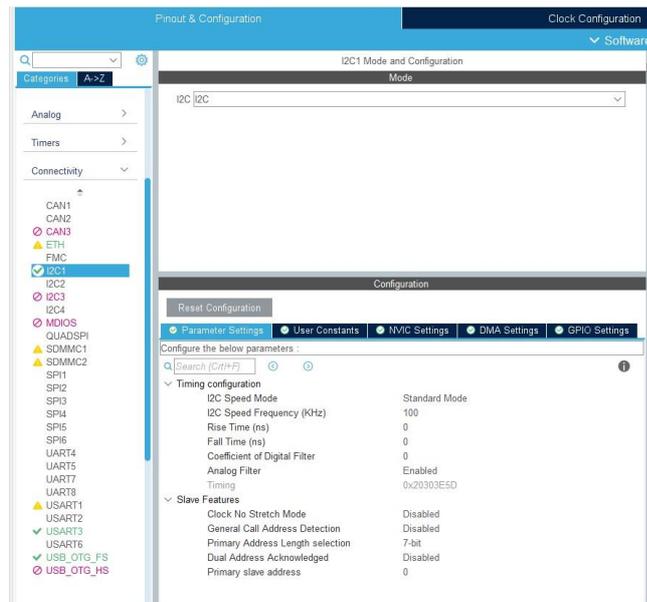


Figure 4: I²C bus configuration

The last step is to configure the GPIO in output in order to pilot the vibration motor. Indeed, it is very easy to pilot him since we have just to set or reset an output pin to switch on or off the motor.

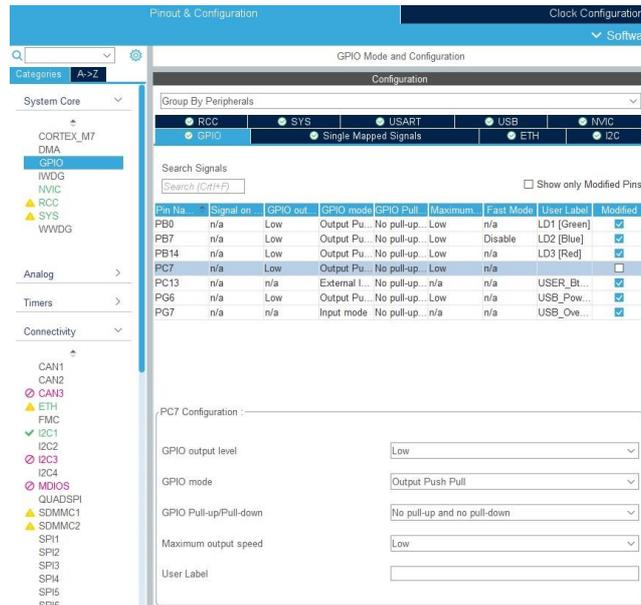


Figure 5: GPIO Configuration

Here, I also kept the default configuration then I connected the pin output to the motor control input and the VDD and GND pins of the moto to the VDD and GND pins of the board.

When this last step has been realized you have to generate the configuration code and the project by clicking on the Device configuration tool code generation.

Before using the sensor, we test the behavior of the mailbox with a simpler example. This example will read the state of the user button of the board and if this one is pushed then we switch on both the led 1 of the board and the vibration motor. The button state will be acquire by a first task and send via the mailbox to the other task. Therefore, the first task will read the user button state and will send it via a mailbox to the second task. This second task will compare the user button state with the push state and if these states match, then both the led 1 and the vibration motor will be switch on.

To do this, we must define some variables and the mailbox in order to send the user button state from one task to the other. You can see in the figure below these definitions.

```

26
27 /* USER CODE END Includes */
28
29 /* Private typedef .....*/
30 /* USER CODE BEGIN PTD */
31 typedef struct
32 { /* Mail object structure */
33   uint32_t var1; /* var1 is a uint32_t */
34   uint32_t var2; /* var2 is a uint32_t */
35   uint8_t var3; /* var3 is a uint8_t */
36 } Mail_TypeDef;
37 /* USER CODE END PTD */
38
39 /* Private define .....*/
40 /* USER CODE BEGIN PD */
41 #define MAIL_SIZE ((uint32_t) 1) /*size of the mail
42 /* USER CODE END PD */
43
44 /* Private macro .....*/
45 /* USER CODE BEGIN PM */
46
47 /* USER CODE END PM */
48
49 /* Private variables .....*/
50
51 #M_HandleTypeDef heth;
52
53 #C_HandleTypeDef h3c1;
54
55 #M_HandleTypeDef huart1;
56
57 #C_HandleTypeDef hpd_usb_dto_fs;
58
59 #M_HandleTypeDef defaulttaskhandle;
60 #M_HandleTypeDef consolehandle;
61 #M_HandleTypeDef producttaskhandle;
62 /* USER CODE BEGIN PV */
63 uint32_t ProducerValue1 = 0; /*producer values: 2 uint32_t & 1 uint8_t
64 uint32_t ProducerValue2 = 0;
65
66 uint32_t ConsumerValue1 = 0; /*consumer values: 2 uint32_t & 1 uint8_t
67 uint8_t ConsumerValue2 = 0;
68
69 osMailQDef mailId; /*mail ID
70
71 uint8_t ButtonState0; /*gives the state of the User Button 0 not pushed 1 pushed
72 /* USER CODE END PV */
73
74 /* Private function prototypes .....*/

```

```

-terminated: Mailbox Debug [STM32 Cortex-M C/C++ Application] (ST-LINK (ST-LINK GDB server)) (Terminated Jun 17, 2021, 9:48:29 AM)
Target is not responding, retrying...
Error! Failed to read target status
Debugger connection lost.
Shutting down...

```

Figure 6: variable and mailbox definition

The first definition is for a data structure that store all the values of the mailbox. Here, 2 variables of 32 bit unsigned char and one 8 bit unsigned char are defined. Firstly, we will use the 8-bit variable to store the user button state, and then we will use the two other variables (32-bits) to store the values coming from the sensor. I define also the size of the mail; this parameter will be used later when we will create the mail queue. I define 4 more 32-bit unsigned char variables and 2 8-bit unsigned char variables to handle the different data. Finally, I define a mail ID variable whose the type is osMailQId (defined in the cmsis_os.h file).

The next step is to define and to create the mail queue and it is realized respectively by using the osMailQDef definition (line 147) and by using the osMailCreate function (line 149).

Next, we have to write the code of the production task in order to provide the data to the consumer task. The data will be store and send between the two tasks by using the mailbox. The figure below shows the first code that I wrote; this code is a simple example in order to test the functionality.

```

528 /* USER CODE BEGIN Header_StartProductTask */
529 /**
530  * @brief Function implementing the ProductTask thread.
531  * @param argument: Not used
532  * @retval None
533  */
534 /* USER CODE END Header_StartProductTask */
535 void StartProductTask(void const * argument)
536 {
537     /* USER CODE BEGIN StartProductTask */
538     #define MailboxTypeDef "pMail"; //pointer to the Mail structure
539     /* Infinite loop */
540     for(;;)
541     {ButtonState= HAL_GPIO_ReadPin(USER_Btn_GPIO_Port, USER_Btn_Pin); //read the state of the User button and store it into the ButtonState variable
542     ProducerValue3=ButtonState; //store the button stat into the producer value 3
543
544     pMail = osMailAlloc(mailId, osMailForever); /* Allocate memory */
545     pMail->var1 = ProducerValue1; /* Set the mail content */
546     pMail->var2 = ProducerValue2;
547     pMail->var3 = ProducerValue3;
548
549     if(osMailPut(mailId, pMail) != osOK) // verify that the mail is ok led 2 on or toggles led2 otherwise
550     {
551         HAL_GPIO_TogglePin(GPIOB,LD2_Pin);
552     }
553     else
554     {
555         HAL_GPIO_WritePin(GPIOB,LD2_Pin,GPIO_PIN_SET);
556     }
557
558     osDelay(1);
559 }
560 /* USER CODE END StartProductTask */
561 }
562
563 /**
564  * @brief This function is executed in case of error occurrence.
565  * @retval None
566  */
567 void Error_Handler(void)
568 {
569     /* USER CODE BEGIN Error_Handler_Debug */
570     /* User can add his own implementation to report the HAL error return state */
571     __disable_irq();
572     while (1)
573     {}
574 }

```

Figure 7: The producer task code

In this code, I read the state of the User button of the board and I store it into the ProducerValue3 variable. This variable is then store into the var3 field of the mailbox. If we cannot store the value into the mailbox then the led2 toggles, otherwise the led2 is switched on.

The second task to write is the consumer one. The figure below shows the code that I wrote in order to use the data comes from the mailbox.

```

528 /* USER CODE BEGIN Header_StartProductTask */
529 /**
530  * @brief Function implementing the ProductTask thread.
531  * @param argument: Not used
532  * @retval None
533  */
534 /* USER CODE END Header_StartProductTask */
535 void StartProductTask(void const * argument)
536 {
537     /* USER CODE BEGIN StartProductTask */
538     #define MailboxTypeDef "pMail"; //pointer to the Mail structure
539     /* Infinite loop */
540     for(;;)
541     {ButtonState= HAL_GPIO_ReadPin(USER_Btn_GPIO_Port, USER_Btn_Pin); //read the state of the User button and store it into the ButtonState variable
542     ProducerValue3=ButtonState; //store the button stat into the producer value 3
543
544     pMail = osMailAlloc(mailId, osMailForever); /* Allocate memory */
545     pMail->var1 = ProducerValue1; /* Set the mail content */
546     pMail->var2 = ProducerValue2;
547     pMail->var3 = ProducerValue3;
548
549     if(osMailPut(mailId, pMail) != osOK) // verify that the mail is ok led 2 on or toggles led2 otherwise
550     {
551         HAL_GPIO_TogglePin(GPIOB,LD2_Pin);
552     }
553     else
554     {
555         HAL_GPIO_WritePin(GPIOB,LD2_Pin,GPIO_PIN_SET);
556     }
557
558     osDelay(1);
559 }
560 /* USER CODE END StartProductTask */
561 }
562
563 /**
564  * @brief This function is executed in case of error occurrence.
565  * @retval None
566  */
567 void Error_Handler(void)
568 {
569     /* USER CODE BEGIN Error_Handler_Debug */
570     /* User can add his own implementation to report the HAL error return state */
571     __disable_irq();
572     while (1)
573     {}
574 }

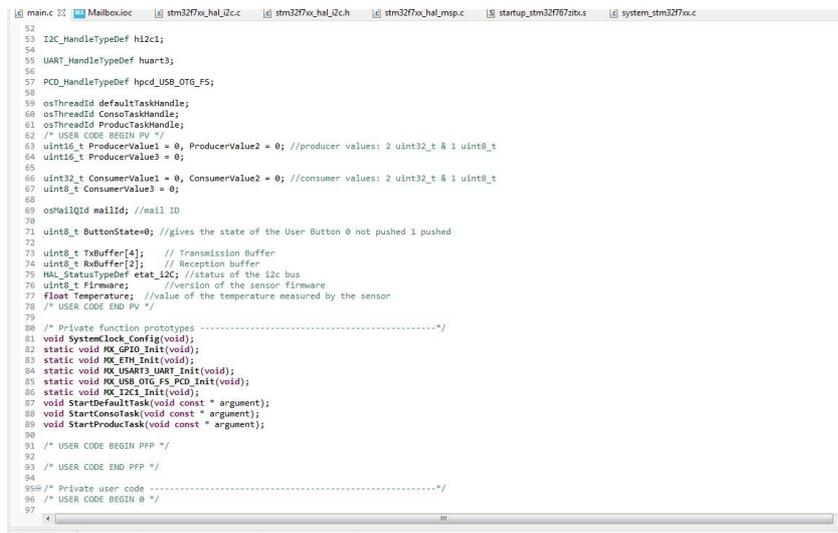
```

Figure 8: The consumer task code

If the User button state is equal to 0 (I also test the values of var1 and var2 fields but here they should be always equal to 0) then I switch off the led 1 and the vibration motor. Otherwise, both the led1 and the vibration moto is switched on.

You can test this code before use the more complex example using the sensor. To do that compile the code, download the execution code into the board and press the User Button.

After making this first simple example, I wrote another code more “complex” in order to send sensor values from one task to another one by using the mailbox. First, I defined new variables to manipulate sensor data as you can see in the figure below.



```
52
53 I2C_HandleTypeDefDef hi2c1;
54
55 UART_HandleTypeDefDef huart3;
56
57 PCD_HandleTypeDefDef hpcd_USB_OTG_FS;
58
59 osThreadId defaultTaskHandle;
60 osThreadId ConsoTaskHandle;
61 osThreadId ProducTaskHandle;
62 /* USER CODE BEGIN PV */
63 uint16_t ProducerValue1 = 0, ProducerValue2 = 0; //producer values: 2 uint32_t & 1 uint8_t
64 uint16_t ProducerValue3 = 0;
65
66 uint32_t ConsumerValue1 = 0, ConsumerValue2 = 0; //consumer values: 2 uint32_t & 1 uint8_t
67 uint8_t ConsumerValue3 = 0;
68
69 osMailQId mailId; //mail ID
70
71 uint8_t ButtonState=0; //gives the state of the User Button 0 not pushed 1 pushed
72
73 uint8_t TxBuffer[4]; // Transmission Buffer
74 uint8_t RxBuffer[2]; // Reception buffer
75 Mail_StatusTypeDef stat_I2C; //status of the i2c bus
76 uint8_t Firmware; //version of the sensor firmware
77 float Temperature; //value of the temperature measured by the sensor
78 /* USER CODE END PV */
79
80 /* Private function prototypes -----*/
81 void SystemClock_Config(void);
82 static void MX_GPIO_Init(void);
83 static void MX_ETH_Init(void);
84 static void MX_USART3_UART_Init(void);
85 static void MX_USB_OTG_FS_PCD_Init(void);
86 static void MX_I2C1_Init(void);
87 void StartDefaulTask(void const * argument);
88 void StartConsoTask(void const * argument);
89 void StartProducTask(void const * argument);
90
91 /* USER CODE BEGIN PFP */
92
93 /* USER CODE END PFP */
94
95 /* Private user code -----*/
96 /* USER CODE BEGIN 0 */
97
```

Figure 9: New variable definitions

Now I modified the production task in order to add the sensor data acquisition. The sensor uses an I²C bus connection to transmit its information to a master. In our case, the master is the μ controller and it sends to the sensor some commands in order to obtain the wanted values. I used the SI7021 sensor that permits to obtain humidity and temperature values. To use this sensor, see its datasheet because here I only use the command to obtain the firmware version and the temperature value.

To obtain the different values of the sensor we have to use some predefined commands. To read the firmware version we must send the command 0x84 then 0xB8. To do that you must use the I²C bus, transmit these values from the master (μ controller) to the sensor, and then listen the sensor answer. Afterward, I want to read the temperature value from the sensor. In order to realize the read you must send to the sensor the following commands: 0xE3,0XF3. The answer of the sensor is stored into the mailbox in order to send these values to the consumer task. The figure below shows you the C code to do both the read of the firmware version and of the temperature value.

```

552 void StartProductTask(void const * argument)
553 {
554     /* USER CODE BEGIN StartProductTask */
555     Amail_TypeDef *pMail; //pointer to the Amail structure
556     /* Infinite loop */
557     for(;;)
558     {
559         *ButtonState= HAL_GPIO_ReadPin(USER_Btn_GPIO_Port, USER_Btn_Pin); //read the state of the User button and store it into the ButtonState variable
560         ProducerValue3=ButtonState; //store the button state into the producer value 3*/
561
562         /* Sensor acquisition*/
563
564         etat_i2c=HAL_I2C_IsDeviceReady(&hi2c1,SI7021_DEFAULT_ADDRESS, 2,10);
565
566         TxBuffer[0]=0x04; //command to read the firmware version
567         TxBuffer[1]=0x08; //command to read the temperature
568         HAL_I2C_Master_Transmit(&hi2c1,SI7021_DEFAULT_ADDRESS,(uint8_t *)TxBuffer,2,10); //Send the command to the sensor
569         HAL_I2C_Master_Receive(&hi2c1,SI7021_DEFAULT_ADDRESS,(uint8_t *)RxBuffer,2,10); //Read the firmware version (1 byte of information but a second one is sent even if i
570         ProducerValue4=RxBuffer[0]; //Store the value in order to send it with the mailbox
571
572         TxBuffer[0]=0xE3; //command to read the temperature
573         TxBuffer[1]=0xF3; //command to read the temperature
574         HAL_I2C_Master_Transmit(&hi2c1,SI7021_DEFAULT_ADDRESS,(uint8_t *)TxBuffer,2,10); //Send the command to the sensor
575         HAL_I2C_Master_Receive(&hi2c1,SI7021_DEFAULT_ADDRESS,(uint8_t *)RxBuffer,2,10); //Read the temperature (2 bytes)
576
577         ProducerValue2=RxBuffer[0]; //Store the value in order to send it with the mailbox
578         ProducerValue3=RxBuffer[1]; //Store the value in order to send it with the mailbox
579
580         pMail = osMailAlloc(mailId, osWaitForever); /* Allocate memory */
581         pMail->var1 = ProducerValue1; /* Set the mail content */
582         pMail->var2 = ProducerValue2;
583         pMail->var3 = ProducerValue3;
584
585         if(osMailPut(mailId, pMail) != osOK) // verify that the mail is ok led 2 on or toggles led2 otherwise
586         {
587             HAL_GPIO_TogglePin(GPIOB,LD2_Pin);
588         }
589         else
590         {
591             HAL_GPIO_WritePin(GPIOB,LD2_Pin,GPIO_PIN_SET);
592         }
593
594         osDelay(1);
595     }
596     /* USER CODE END StartProductTask */
597 }

```

Figure 10: Code to read the firmware version and the temperature value

It is important to notice that in order to read a value from the sensor you have to, first send to it a command by using the I²C Transmit function then listen to the answer of the sensor by using the I²C Receive function.

Then, we have to modify the consumer task code. Here, I write a code allows us to compute the temperature value by using the answer of the sensor. Indeed, the sensor returns us a value on 2 bytes so we must concatenate this value to be able to compute the real temperature value see the data sheet to know how compute this value). Afterward, I compare both the temperature and the firmware value in order to switch on the led 1 and the vibration motor. The code is shown below.

```

498 * @param argument: Not used
499 * @retval None
500 */
501 /* USER CODE END Header_StartConsoTask */
502 void StartConsoTask(void const * argument)
503 {
504     /* USER CODE BEGIN StartConsoTask */
505     osEvent event;
506     Amail_TypeDef *pMail;
507     uint32_t Mesured_Temperature;
508
509     /* Infinite loop */
510     for(;;)
511     {
512         /* Get the message from the queue */
513         event = osMailGet(mailId, osWaitForever); /* wait for mail */
514
515         if(event.status == osEventMail)
516         {
517             pMail = event.value;
518
519             if((pMail->var1 == 0x20) && (pMail->var2 == 10) || (pMail->var3 == 20)) //if all the values of the email is equal to 0
520             {
521                 HAL_GPIO_WritePin(GPIOB,LD1_Pin,GPIO_PIN_RESET);
522                 HAL_GPIO_WritePin(GPIOB,LD2_Pin,GPIO_PIN_SET);
523                 HAL_GPIO_WritePin(GPIOC,GPIO_PIN_7,GPIO_PIN_RESET);
524             }
525             else
526             {
527                 Firmware=pMail->var1;
528                 Mesured_Temperature=pMail->var2<<8;
529                 Mesured_Temperature=Mesured_Temperature + pMail->var3;
530                 Temperature=((175.72*Mesured_Temperature)/65536)-46.85;
531
532                 if(Firmware==0x20 && Temperature<60)
533                 {
534                     HAL_GPIO_WritePin(GPIOB,LD1_Pin,GPIO_PIN_SET);
535                     HAL_GPIO_WritePin(GPIOB,LD2_Pin,GPIO_PIN_RESET);
536                     HAL_GPIO_WritePin(GPIOC,GPIO_PIN_7,GPIO_PIN_SET);
537                 }
538             }
539
540             osDelay(1);
541         }
542     }
543 }

```

Figure 11: Consumer task code

A video showing the code test can be found here: <https://youtu.be/5MFwuZELhwM>