FreeRTOS架構

FreeRTOS是一個相對較小的系統。最小化的FreeRTOS核心僅包括3個（.c）文件和少數標頭檔，總共不到9000行程式碼，還包括了註解和空行。一個典型的編譯後（二進制）binary小於10KB。

FreeRTOS的程式碼可以分解為三個主要區塊：任務、通訊，和硬體界面。

- 任務 (Task): 大約有一半的FreeRTOS的核心程式碼用來處理多數作業系統首要關注的問題：任務。任務是給定優先權的用戶定義的C函數。 task.c和task.h完成了所有有關創造，排程，和維護任務的繁重工作。
- 通訊 (Communication): 任務很重要，不過任務間可以互相通訊則更為重要！它給我們帶來FreeRTOS的第二項任務：通訊。大約40%的FreeRTOS核心程式碼是用來處理通訊的。 queue.c和queue.h是負責處理FreeRTOS的通訊的。任務和中斷使用隊列互相發送數據，並且使用semaphore和mutex來發送critical section的使用情況。
- 硬體界面：接近9000行的程式碼拼湊起基本的FreeRTOS，是和硬體無關的；相同的程式碼都能夠運行，大約有6%的FreeRTOS的核心代碼，在硬體無關的FreeRTOS核心與硬體相關的程式碼間扮演著墊片的角色。我們將在下個部分討論硬體相關的程式碼。[1]

與平台相關的檔案

- portmacro.h
    - 定義了硬體相關變數，如資料形態定義，以及硬體相關的函式呼叫的名稱定義(以portXXXXX爲名)等，統一各平臺呼叫函式的
- port.c
    - 定義了包含和硬體相關的程式碼的實作
- FreeRTOSConfig.h
    - 包含Clock speed, heap size, mutexes等等都在此定義

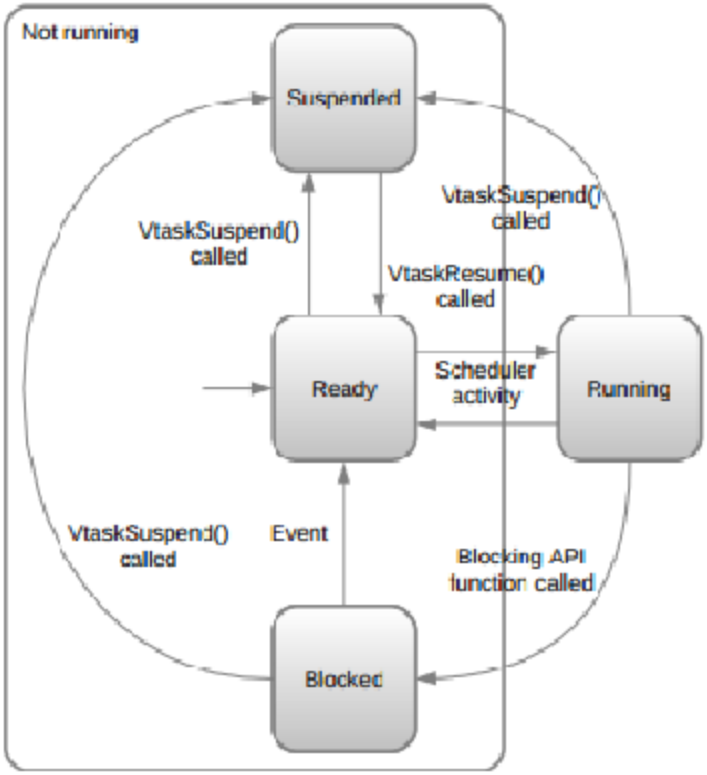
命名規則

- 變數
    - char類型：以 c 為字首
    - short類型：以 s 為字首

    - long類型：以 l 為字首
    - float類型：以 f 為字首
    - double類型：以 d 為字首
    - Enum變數：以 e 為字首
    - 其他（如struct）：以 x 為字首
    - pointer有一个額外的字首 p , 例如short類型的pointer字首為 ps
    - unsigned類型的變數有一個額外的字首 u , 例如unsigned short類型的變數字首為 us
- Functions
    - 文件內：以 prv 為字首
    - API：以其return類型為字首，按照對變數的定義
    - 名字：以其所在的檔案名稱開頭。如vTaskDelete即在Task.c文件中名稱

Run FreeRTOS on STM32F4-Discovery

Task的狀態



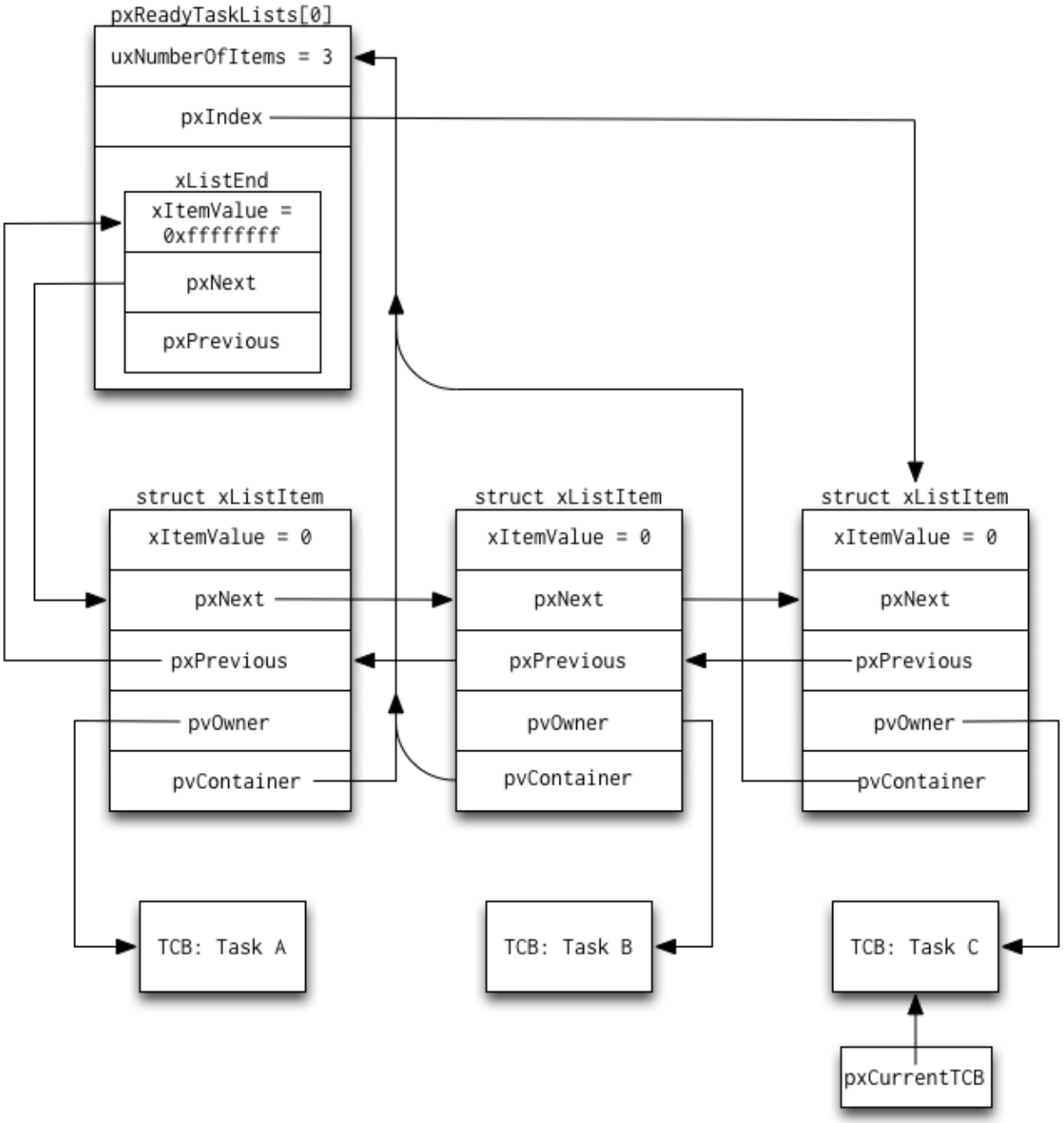
- Ready : 準備好要執行的狀態

- Running : 正在給CPU執行的狀態
- Block : 等待中的狀態
- Suspended :等待中的狀態

每一種狀態狀態FreeRTOS都會給予一個list儲存（除了runnning)

Ready list的資料形態

FreeRTOS使用ready list去管理待準備好要執行的tasks，而ready list的資料儲存方式如下圖



- Context Switch 時選出下一個欲執行的task

下面是在ready list中依照優先度選取執行目標的程式其中，FreeRTOS的優先度排序最小優先權爲0，數字越大則優先權越高

task.c (taskSELECT_HIGHEST_PRIORITY_TASK)

```
while( listLIST_IS_EMPTY( &( pxReadyTasksLists[ uxTopReadyPriority ] ) ) )
```

```
{
    configASSERT( uxTopReadyPriority );
    --uxTopReadyPriority;
}
listGET_OWNER_OF_NEXT_ENTRY( pxCurrentTCB, &( pxReadyTasksLists[ uxTopReadyPriority ] )
);
```

include/list.h

```
#define listGET_OWNER_OF_NEXT_ENTRY( pxTCB, pxList )
{
    List_t * const pxConstList = ( pxList );
    /* Increment the index to the next item and return the item, ensuring */
    /* we don't return the marker used at the end of the list.  */
    ( pxConstList )->pxIndex = ( pxConstList )->pxIndex->pxNext;
    if( ( void * ) ( pxConstList )->pxIndex == ( void * ) &( ( pxConstList )->xListEnd )
)  \
    {
        ( pxConstList )->pxIndex = ( pxConstList )->pxIndex->pxNext;
    }
    ( pxTCB ) = ( pxConstList )->pxIndex->pvOwner;
}
```

- 創造全新task

TCB的資料結構：

tasks.c

```
/* In file: tasks.c */
typedef struct tskTaskControlBlock
{

    volatile portSTACK_TYPE *pxTopOfStack;                  /* Points to the location of

the last item placed on

the tasks stack.  THIS

MUST BE THE FIRST MEMBER

OF THE STRUCT. */
```

```
    xListItem    xGenericListItem;                          /* List item used to place
                                                                                    the
TCB in ready and

blocked queues. */

    xListItem    xEventListItem;                            /* List item used to place
                                                                                the TCB
in event lists.*/

    unsigned portBASE_TYPE uxPriority;                      /* The priority of the task

where 0 is the lowest

priority. */

    portSTACK_TYPE *pxStack;                                /* Points to the start of
                                                                                    the
stack. */

    signed char    pcTaskName[ configMAX_TASK_NAME_LEN ];   /* Descriptive name given

to the task when created.

Facilitates debugging

only. */

    #if ( portSTACK_GROWTH > 0 )
    portSTACK_TYPE *pxEndOfStack;                          /* Used for stack overflow

checking on architectures

where the stack grows up

from low memory. */
    #endif

    #if ( configUSE_MUTEXES == 1 )
    unsigned portBASE_TYPE uxBasePriority;                 /* The priority last

assigned to the task -
```

```
used by the priority

inheritance mechanism. */
  #endif

} tskTCB;
```

pxTopOfStack , pxEndOfStack :紀錄Stack的大小

uxPriority , uxBasePriority ：紀錄優先權 ,而後者是紀錄原本的優先權（可能發生再Mutux)

xGenericListItem , xEventListItem : 當一個任務被放入到FreeRTOS的一個列表中 ，被插入pointer的地方

xTaskCreate()函數被呼叫的時候，一個任務被建立。 FreeRTOS為一個任務新分配一個TCB target，來記錄它的名稱，優先級，和其他細節，接著分配用戶請求的總的HeapStack（假設有足夠使用的記憶體）和在TCB的pxStack成員中記錄Stack的記憶體起點。

爲了讓排程方便，以及第一次出場的task所以創造新task時，stack中除了該有的資料外，還要加上“空的”register資料(第一次執行，理論不會有register資料)，讓新task就像是一般被context switch的task一樣按照對變數的定義下面是實作方式

portable/GCC/ARM_CM4F/port.c

```
/* In file: port.c */
StackType_t *pxPortInitialiseStack( StackType_t *pxTopOfStack, TaskFunction_t pxCode,
void  *pvParameters )
{

/* Simulate the stack frame as it would be created by a context switch
    interrupt. */

    /* Offset added to account for the way the MCU uses the stack on entry/exit
    of interrupts, and to ensure alignment. */
    pxTopOfStack--;

    *pxTopOfStack = portINITIAL_XPSR;    /* xPSR */
    pxTopOfStack--;
    *pxTopOfStack = ( StackType_t ) pxCode;     /* PC */
    pxTopOfStack--;
    *pxTopOfStack = ( StackType_t ) portTASK_RETURN_ADDRESS;  /* LR */
```

```
    /* Save code space by skipping register initialisation. */
    pxTopOfStack -= 5;      /* R12, R3, R2 and R1. */
    *pxTopOfStack = ( StackType_t ) pvParameters;      /* R0 */

    /* A save method is being used that requires each task to maintain its
    own exec return value. */
    pxTopOfStack--;
    *pxTopOfStack = portINITIAL_EXEC_RETURN;

    pxTopOfStack -= 8;      /* R11, R10, R9, R8, R7, R6, R5 and R4. */

    return pxTopOfStack;
}
```

在TCB完成初始化後，要把該TCB接上其他相關的list結構，這個過程中必須暫時停止interrupts功能，以免在list還沒設定好前，被其他的task使用停止

而當ARM Cortex-M4處理器在task遇上中斷時，會將register之內容push上該task的stack的頂端，待下次運行時pop回去 以下是在 port.c裡的實作

portable/GCC/ARM_CM4F/port.c

```
/* In file: port.c */
void xPortPendSVHandler( void )
{      /* This is a naked function. */
    __asm volatile
    (

    "  mrs r0, psp                          \n"
    "  isb                                  \n"
    "                                       \n"
    "  ldr    r3, pxCurrentTCBConst              \n"           /* Get the location of the
current TCB. */
    "  ldr    r2, [r3]                      \n"
    "                                       \n"
    "  tst r14, #0x10                           \n"           /* Is the task using the
FPU context?
                                                                                    If
so, push high vfp registers. */
    "  it eq                                \n"
    "  vstmdbeq r0!, {s16-s31}                  \n"
```

```
    "                                   \n"
    "  stmdb r0!, {r4-r11, r14}         \n"           /* Save the core registers. */
    "                                   \n"
    "  str r0, [r2]                     \n"           /* Save the new top of stack into
the

first member of the TCB. */
    "                                   \n"
    "  stmdb sp!, {r3}                      \n"
    "  mov r0, %0                       \n"
    "  msr basepri, r0                      \n"
    "  bl vTaskSwitchContext                \n"
    "  mov r0, #0                       \n"
    "  msr basepri, r0                      \n"
    "  ldmia sp!, {r3}                      \n"
    "                                   \n"
    "  ldr r1, [r3]                     \n"           /* The first item in pxCurrentTCB
is the

task top of stack. */
    "  ldr r0, [r1]                     \n"
    "                                   \n"
    "  ldmia r0!, {r4-r11, r14}         \n"           /* Pop the core registers. */
    "                                   \n"
    "  tst r14, #0x10                       \n"           /* Is the task using the
FPU context?

If so, pop the high vfp registers too. */
    "  it eq                            \n"
    "  vldmiaeq r0!, {s16-s31}              \n"
    "                                   \n"
    "  msr psp, r0                      \n"
    "  mrs r0, psp                      \n"
    "  isb                              \n"
    "                                   \n"
    "  ldr    r3, pxCurrentTCBConst         \n"           /* Get the location of the
current TCB. */
    "  ldr    r2, [r3]                  \n"
    "                                   \n"
    "  tst r14, #0x10                       \n"           /* Is the task using the
FPU context?
                                                                                  If
so, push high vfp registers. */
```

```
    "  it eq                                \n"
    "  vstmdbeq r0!, {s16-s31}                      \n"
    "                                       \n"
    "  stmdb r0!, {r4-r11, r14}             \n"           /* Save the core registers. */
    "                                       \n"
    "  str r0, [r2]                         \n"           /* Save the new top of stack into
the

first member of the TCB. */
    "                                       \n"
    "  stmdb sp!, {r3}                              \n"
    "  mov r0, %0                           \n"
    "  msr basepri, r0                              \n"
    "  bl vTaskSwitchContext                        \n"
    "  mov r0, #0                           \n"
    "  msr basepri, r0                              \n"
    "  ldmia sp!, {r3}                              \n"
    "                                       \n"
    "  ldr r1, [r3]                         \n"           /* The first item in pxCurrentTCB
is the

task top of stack. */
    "  ldr r0, [r1]                         \n"
    "                                       \n"
    "  ldmia r0!, {r4-r11, r14}             \n"           /* Pop the core registers. */
    "                                       \n"
    "  tst r14, #0x10                               \n"           /* Is the task using the
FPU context?

If so, pop the high vfp registers too. */
    "  it eq                                \n"
    "  vldmiaeq r0!, {s16-s31}                      \n"
    "                                       \n"
    "  msr psp, r0                          \n"

    "  isb                                  \n"
    "                                       \n"
    #ifdef WORKAROUND_PMU_CM001 /* XMC4000 specific errata workaround. */
      #if WORKAROUND_PMU_CM001 == 1
    "               push { r14 }            \n"
    "               pop { pc }              \n"
      #endif
    #endif
```

```
    "                                   \n"
    "  bx r14                           \n"
    "                                   \n"
    "  .align 2                         \n"
    "                                   \n"
    "  bx r14                           \n"
    "                                   \n"
    "  .align 2                         \n"
    "pxCurrentTCBConst: .word pxCurrentTCB\n"
    ::"i"(configMAX_SYSCALL_INTERRUPT_PRIORITY)
    );
}
```

Interrupt的實作，是將CPU中控制interrupt權限的暫存器(basepri)內容設爲最高，此時將沒有任何interrupt可以被呼叫該處呼叫的函數名為ulPortSetInterruptMask()

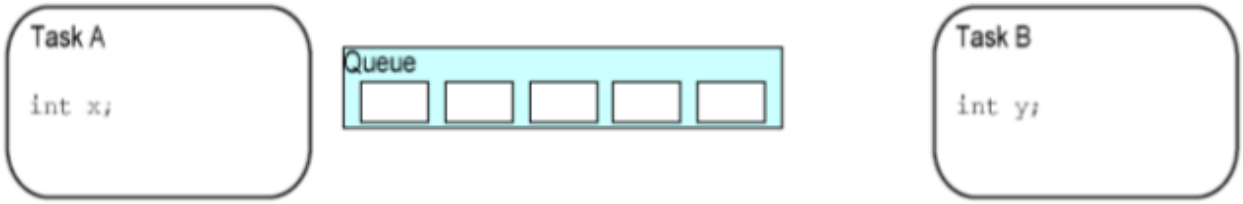
portable/GCC/ARM_CM4F/port.c

```
__attribute__(( naked )) uint32_t ulPortSetInterruptMask( void )
{
        __asm volatile
        (
                "       mrs r0, basepri         \n"
                "       mov r1, %0              \n"
                "       msr basepri, r1         \n"
                "       bx lr                   \n"
                :: "i" ( configMAX_SYSCALL_INTERRUPT_PRIORITY ) : "r0", "r1"
        );

        /* This return will not be reached but is necessary to prevent compiler
        warnings. */
        return 0;
```

藉此Mask所有的Interrupt(所有優先權低於 configMAX_SYSCALL_INTERRUPT_PRIORITY 的TASK將無法被執行

參照：http://www.freertos.org/RTOS-Cortex-M3-M4.html

Communication

- Queue
- 

使用Queue來傳送data（variable或是pointer，但是用deep copy一份到queue裡，而不是用pointer指向variable） Queue的結構如下

queue.c

```
/* In file: queue.c */
typedef struct QueueDefinition{

  signed char *pcHead;                    /* Points to the beginning of the queue
                                             storage area. */

  signed char *pcTail;                    /* Points to the byte at the end of the
                                             queue storage area. One more byte is
                                             allocated than necessary to store the
                                           queue items; this is used as a marker. */

  signed char *pcWriteTo;                 /* Points to the free next place in the
                                             storage area. */

  signed char *pcReadFrom;                /* Points to the last place that a queued
                                             item was read from. */

  xList xTasksWaitingToSend;              /* List of tasks that are blocked waiting
                                             to post onto this queue.  Stored in
                                             priority order. */

  xList xTasksWaitingToReceive;           /* List of tasks that are blocked waiting
                                             to read from this queue. Stored in
                                             priority order. */

  volatile unsigned portBASE_TYPE uxMessagesWaiting;/* The number of items currently
                                                      in the queue. */
```

```
  unsigned portBASE_TYPE uxLength;                    /* The length of the queue
                                                         defined as the number of
                                                         items it will hold, not the
                                                         number of bytes. */

  unsigned portBASE_TYPE uxItemSize;                  /* The size of each items that
                                                         the queue will hold. */

} xQUEUE;
```

由於queue可以被許多個task進行寫入資料（即send data），當queue放滿了但卻還有task要進行寫入時，則使用xTasksWaitingToSend這個list來追蹤這些被block住的task（等待寫入資料到queue裡的task），每當有一個item從queue中被移除，xTasksWaitingToSend就會被檢查是否有task在此list，並且會在這些被block的task中挑出一個priority最高的task，將他解除block然後進行寫入資料的動作，若這些task的priority都一樣高，那將會挑出等待最久的task。

相同的，許多task進行讀取資料（即receive data）亦是如次，若queue沒有任何item在裡面，而同時還有好幾個task想要讀取資料，則這些task會被插在xTasksWaitingToReceive裡，xTasksWaitingToReceive這個list用來追蹤這些被block住的task（等待從queue中讀取資料的task）。

Queue在FreeRTOS的實作如下：

/CORTEX_M4F_STM32_DISCOVERY/main.c

```
/*file: ./CORTEX_M4F_STM32F407ZG-SK/main.c*/
/*line:47*/
xQueueHandle MsgQueue;
/*line:214*/
void QTask1( void* pvParameters )
{
        uint32_t snd = 100;

        while( 1 ){
                xQueueSend( MsgQueue, ( uint32_t* )&snd, 0 );
                vTaskDelay(1000);
        }
}

void QTask2( void* pvParameters )
{
```

```
        uint32_t rcv = 0;
        while( 1 ){
                if( xQueueReceive( MsgQueue, &rcv, 100/portTICK_RATE_MS ) == pdPASS  &&
rcv == 100)
                {
                        STM_EVAL_LEDToggle( LED3 );
                }
        }
}
```

關於QUEUE的操作函式定義在./queue.h

FreeRTOS也可使用Queue來實作Semaphores 和 Mutexes：

Semaphores按照對變數的定義

semaphore用來讓一個task喚醒喚醒(wake)另一個task做事，例如:producer和consumer

Mutexes

mutex用來對共享資源做互斥存取

- 實作semaphore
- N-element semaphore，只需同步uxMessagesWaiting，且只需關心有多少queue entries被佔用，其中uxItemSize為0，item和data copying是不需要的
- 需要 ARM Cortex-M4F 特有的機制，才能實做 semaphore 嗎？若是，其機制為何？
- 需要，在存取uxMessagesWaiting時必須一次只能一個task去做更改（進、出入critical section），要防止一次兩個task進入修改的機制如下：

portable/GCC/ARM_CM4F/port.c

```
/* In file: port.c */
void vPortEnterCritical( void )
{
```

```
    portDISABLE_INTERRUPTS();
    uxCriticalNesting++;
    __asm volatile( "dsb" );
    __asm volatile( "isb" );
}

/*-----------------------------------------------------------*/

void vPortExitCritical( void )
{
    configASSERT( uxCriticalNesting );
    uxCriticalNesting--;
    if( uxCriticalNesting == 0 )
    {
        portENABLE_INTERRUPTS();
    }
}
```

- 實作mutex
- 因為pcHead和pcTail不需要，所以overloadind來達到較好的使用率：

queue.c

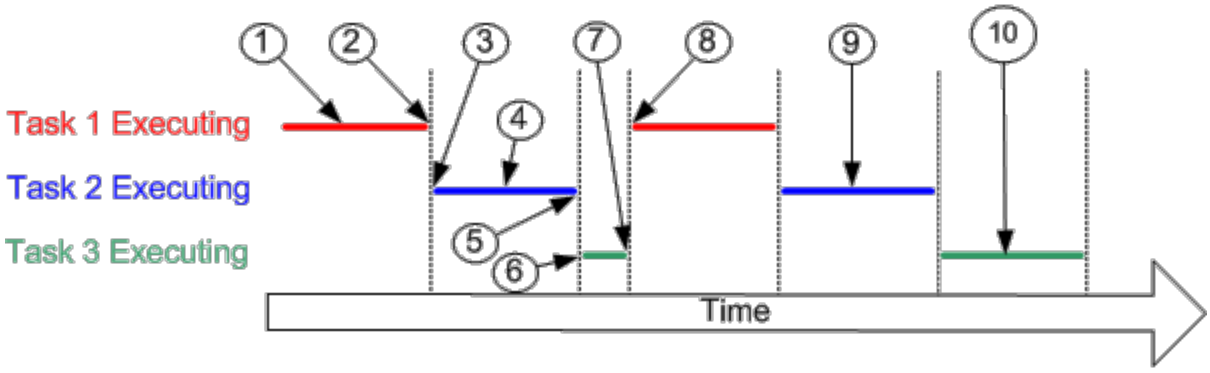
```
/* Effectively make a union out of the xQUEUE structure. */
#define uxQueueType           pcHead
#define pxMutexHolder         pcTail
```

uxQueueType若為0，表示this queue is being used for a mutex pxMutexHolder用來實作priority inheritance

mutex 和 semaphore 的差異，請參見這篇短文: http://embeddedgurus.com/barr-code/2008/01/rtos-myth-1-mutexes-and-semaphores-are-interchangeable/ 另外: http://embeddedgurus.com/barr-code/2008/03/rtos-myth-3-mutexes-are-needed-at-the-task-level/

Scheduling

- 基本概念



在一般非即時作業系統上，通常每個task都會分到相同的CPU使用時間，RTOS則不盡然，後續將提到相關資訊 除了由kernel要求task交出CPU控制權外，各task也能夠選擇自行交出CPU控制權，舉凡

Delay(sleep): 停止運行一段特定時間

/CORTEX_M4F_STM32_DISCOVERY/main.c

```
void Task2( void* pvParameters )
{
	while( 1 ){
		vTaskDelay( 1000 );
		itoa(iii, 10);
		iii = 0;
	}
}
```

. wait(block): 等待取得某資源

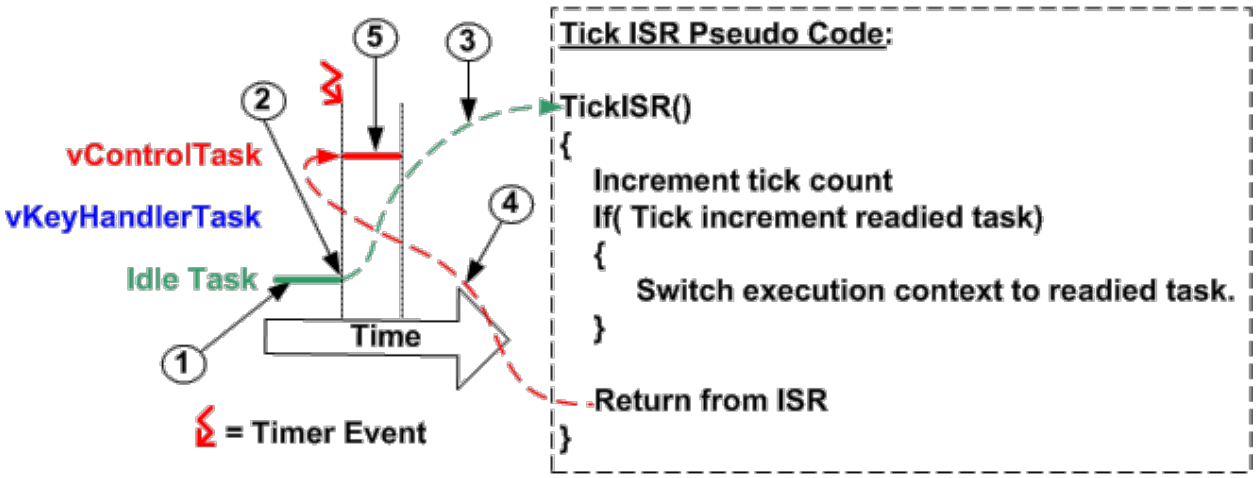
```
void QTask2( void* pvParameters )
{
	uint32_t rcv = 0;
	while( 1 ){
		if( xQueueReceive( MsgQueue, &rcv, 100/portTICK_RATE_MS ) == pdPASS  &&
rcv == 100)
		{
			STM_EVAL_LEDToggle( LED3 );
		}
```

```
	}
}
```

等待事件觸發(例如鍵盤輸入)

```
void Task1( void* pvParameters )
{
	while( 1 ){
		iii++;
		while( STM_EVAL_PBGetState( BUTTON_USER ) ){
			iii++;
				STM_EVAL_LEDOn(LED4);
		}
	}
}
```

.

- RTOS tick
- 

portable/GCC/ARM_CM4F/port.c

```
void xPortSysTickHandler( void )
{
	/* The SysTick runs at the lowest interrupt priority, so when this interrupt
	executes all interrupts must be unmasked.  There is therefore no need to
	save and then restore the interrupt mask value as its value is already
	known. */
	( void ) portSET_INTERRUPT_MASK_FROM_ISR();
	{
```

```
            /* Increment the RTOS tick. */
            if( xTaskIncrementTick() != pdFALSE )
            {
                    /* A context switch is required.  Context switching is performed in
                    the PendSV interrupt.  Pend the PendSV interrupt. */
                    portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT;
            }
    }
    portCLEAR_INTERRUPT_MASK_FROM_ISR( 0 );
}
```

- Starvation

FreeRTOS為MultiLevel Queue，若priority高的task霸佔CPU，對於priority較低的task則無法執行，便會發生Starvation(低優先權很長一段時間都無法獲得CPU執行)

Interrupt handler

This is a reason why several functions exists in two versions: one for regular tasks and another is intended to interrupts handler.

For this reason, it is necessary to make interrupts handlers' execution as short as possible. On way to achieve this goal consists in the creation of tasks waiting for an interrupt to occur with a semaphore, and let this safer portion of code actually handle the interrupt. An ISR "gives" a semaphore and unblock a 'Handler" task that is able to handler the ISR, making the ISR execution much shorter >>External Interrupt(EXTI)

所有interrupt handler的排序和名稱定義(並非實作內容)放置在startup_stm32f429_439xx.s中，實作則需要在其他地方宣告

使用者可以定義的外部中斷觸發條件爲EXTI_Line0至EXTI_Line15，不過EXTI_Line10~15和EXTI_Line5~9被設定爲觸發同一外部中斷，實際上能夠定義的中斷處理只有EXTI_Line0, 1, 2, 3, 4, 5~9, 10~15，總計七個

file:startup_stm32f429_439xx.s line:158

CORTEX_M4F_STM32_DISCOVERY/startup/startup_stm32f4xx.s

```
.word     EXTI0_IRQHandler                   /* EXTI Line0                   */
.word     EXTI1_IRQHandler                   /* EXTI Line1                   */
```

```
.word     EXTI2_IRQHandler                  /* EXTI Line2                  */
.word     EXTI3_IRQHandler                  /* EXTI Line3                  */
.word     EXTI4_IRQHandler                  /* EXTI Line4                  */

.word     EXTI9_5_IRQHandler                /* External Line[9:5]s         */

.word     EXTI15_10_IRQHandler              /* External Line[15:10]s       */
```

EXTI使用前必須：

- 欲使用的GPIO之初始設定
- 和GPIO連接(作爲觸發來源)
- 設定EXTI
- 模式(Interrupt, Event)
- 被觸發的條件(Rising, Falling, Rising&falling)
- LineCmd(尚未查明)
- 設定NVIC通道，權限

EXTI在FreeRTOS的實作如下：file:./CORTEX_M4F_STM32F407ZG-SK/main.c line:106

CORTEX_M4F_STM32_DISCOVERY/main.c

```
/* Configure PA0 pin as input floating */
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
GPIO_Init(GPIOA, &GPIO_InitStructure);

/* Connect EXTI Line0 to PA0 pin */
SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOA, EXTI_PinSource0);

/* Configure EXTI Line0 */
EXTI_InitStructure.EXTI_Line = EXTI_Line0;
EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;
EXTI_InitStructure.EXTI_LineCmd = ENABLE;
EXTI_Init(&EXTI_InitStructure);

/* Enable and set EXTI Line0 Interrupt to the lowest priority */
NVIC_InitStructure.NVIC_IRQChannel = EXTI0_IRQn;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0xFF;
```

```
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0xFF;
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
NVIC_Init(&NVIC_InitStructure);
```

硬體驅動原理

- 簡介
    - 參考 STM32Cube_FW_F4_V1.1.0/Projects/STM32F429I-Discovery/Examples/GPIO/GPIO_EXTI/readme.txt

能夠讓開發者藉由改變記憶體特定位置的資料內容，來控制各種周邊

或者，藉由外部的輸入，來改變記憶體內容，讓硬體得知其變化，來做反應

使用GPIO前，必須預先設定行爲和細節，以stm32f429i_discovery.c內的LED初始作業來看

Utilities/STM32F429I-Discovery/stm32f429i_discovery.c

```
/* In file: stm32f429i_discovery.c */
void STM_EVAL_LEDInit(Led_TypeDef Led)
{
  GPIO_InitTypeDef  GPIO_InitStructure;

  /* Enable the GPIO_LED Clock */
  RCC_AHB1PeriphClockCmd(GPIO_CLK[Led], ENABLE);

  /* Configure the GPIO_LED pin */
  GPIO_InitStructure.GPIO_Pin = GPIO_PIN[Led];
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
  GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_Init(GPIO_PORT[Led], &GPIO_InitStructure);
}
```

使用者開啟使用到的GPIO之bus的clock，然後使用GPIO_InitStructure(struct資料型態)來儲存相關設定，並交與GPIO_Init()去做記憶體的設定

- 底層實作

用上面LED初始的過程當舉例，其中接觸到硬體設定的function爲

RCC_AHB1PeriphClockCmd()和GPIO_Init()

先從RCC_AHB1PeriphClockCmd()開始討論

```
/* In file: stm32f4xx.h */
typedef struct
{
...
__IO uint32_t AHB1ENR;       /*!< RCC AHB1 peripheral clock register,
Address offset: 0x30 */
...
} RCC_TypeDef;

#define RCC                  ((RCC_TypeDef *) RCC_BASE)
#define RCC_BASE               (AHB1PERIPH_BASE + 0x3800)
#define AHB1PERIPH_BASE        (PERIPH_BASE + 0x00020000)
#define PERIPH_BASE            ((uint32_t)0x40000000) /*!< Peripheral base address in the
alias region

/* In file: stm32f4xx_rcc.h */
#define RCC_AHB1Periph_GPIOA             ((uint32_t)0x00000001)

/* In file: stm32f4xx_rcc.c */
void RCC_AHB1PeriphClockCmd(uint32_t RCC_AHB1Periph, FunctionalState NewState)
{
  /* Check the parameters */
  assert_param(IS_RCC_AHB1_CLOCK_PERIPH(RCC_AHB1Periph));

  assert_param(IS_FUNCTIONAL_STATE(NewState));
  if (NewState != DISABLE)
  {
    RCC->AHB1ENR |= RCC_AHB1Periph;
  }
  else
  {
    RCC->AHB1ENR &= ~RCC_AHB1Periph;
  }
}
```

不難發現，GPIO周邊的設定在實作上相當單純，即在預先定義好的記憶體區段上，按照設計者定義之設定將參數寫在該處

值得一提的是，GPIO的初始化和設定皆是在執行時期進行，即使在程式運作中依然能夠重新定

義甚至關閉周邊，讓MCU的使用更為彈性

GPIO_Init()的執行也脫離不了寫入資料至記憶體一事，但設定項目更多，設定過程也相對複雜

GPIO_Init()

CORTEX_M4F_STM32_DISCOVERY/Libraries/STM32F4xx_StdPeriph_Driver/src/stm32f4xx_gpio.c

```
/* In file: stm32f4xx_gpio.c */
void GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct)
{
  uint32_t pinpos = 0x00, pos = 0x00 , currentpin = 0x00;

  /* Check the parameters */
  assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
  assert_param(IS_GPIO_PIN(GPIO_InitStruct->GPIO_Pin));
  assert_param(IS_GPIO_MODE(GPIO_InitStruct->GPIO_Mode));
  assert_param(IS_GPIO_PUPD(GPIO_InitStruct->GPIO_PuPd));

  /* ------------------------ Configure the port pins ---------------- */
  /*-- GPIO Mode Configuration --*/
  for (pinpos = 0x00; pinpos < 0x10; pinpos++)
  {
    pos = ((uint32_t)0x01) << pinpos;
    /* Get the port pins position */
    currentpin = (GPIO_InitStruct->GPIO_Pin) & pos;

    if (currentpin == pos)
    {
      GPIOx->MODER  &= ~(GPIO_MODER_MODER0 << (pinpos * 2));
      GPIOx->MODER |= (((uint32_t)GPIO_InitStruct->GPIO_Mode) << (pinpos * 2));

      if ((GPIO_InitStruct->GPIO_Mode == GPIO_Mode_OUT) || (GPIO_InitStruct->GPIO_Mode ==
GPIO_Mode_AF))
      {
        /* Check Speed mode parameters */
        assert_param(IS_GPIO_SPEED(GPIO_InitStruct->GPIO_Speed));

        /* Speed mode configuration */
        GPIOx->OSPEEDR &= ~(GPIO_OSPEEDER_OSPEEDR0 << (pinpos * 2));
        GPIOx->OSPEEDR |= ((uint32_t)(GPIO_InitStruct->GPIO_Speed) << (pinpos * 2));

        /* Check Output mode parameters */
```

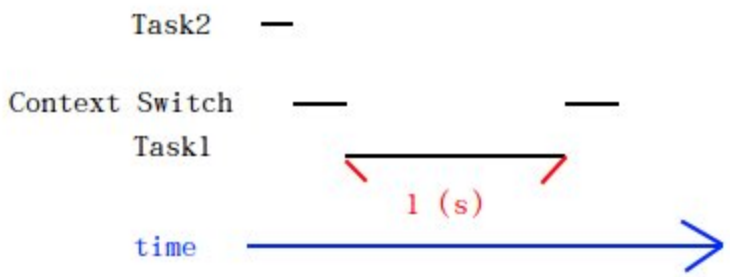
```
        assert_param(IS_GPIO_OTYPE(GPIO_InitStruct->GPIO_OType));

        /* Output mode configuration*/
        GPIOx->OTYPER  &= ~((GPIO_OTYPER_OT_0) << ((uint16_t)pinpos)) ;
        GPIOx->OTYPER |= (uint16_t)(((uint16_t)GPIO_InitStruct->GPIO_OType) <<
((uint16_t)pinpos));
      }

      /* Pull-up Pull down resistor configuration*/
      GPIOx->PUPDR &= ~(GPIO_PUPDR_PUPDR0 << ((uint16_t)pinpos * 2));
      GPIOx->PUPDR |= (((uint32_t)GPIO_InitStruct->GPIO_PuPd) << (pinpos * 2));
    }
  }
}
```

效能表現

- Context switch

(context switch是指taskA要交出CPU使用權給taslB時，將taslA當前的state和register內資料存放至記憶體內，將先前taskB的state從記憶體讀取至register內的過程)

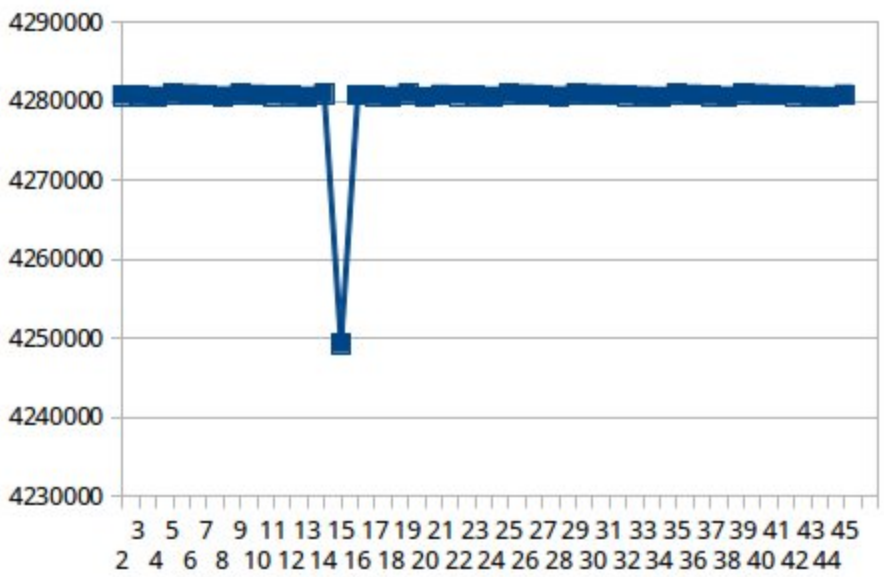
我們想得知FreeRTOS的context switch時間，並想出一個測試方法：
CORTEX_M4F_STM32_DISCOVERY/main.c



1. 首先建立task1和task2，其中task2的priority大於task1的priority。task2先執行時，馬上就進行vTaskDelay使task2移至block狀態1秒，這時就會發生context switch，換成task1執行，這1秒的時間，task1不斷的進行i++，直到1秒結束後，回到task2執行，再由task2印出i值，並把i重新設0，此為一個週期。此動作可得到i在一秒時可跑至多少，設一秒可跑至k值。
2. 設定一個task3其priority高於task2，讓task3執行vTaskDelay 300秒，當300秒結束後，會中斷task1所執行的i++。再由task3印出i值，設其為final_i，k值與final_i值的

差額，即為context switch的總時間。

下圖為隨機挑出45個i值做成圖表，其中平均i值為：4280015



接著我們測出的final_i值，平均為：3913853，故可得到 (4280015 - 3913853)/ 4280015 = 0.0855 (秒)

0.0855秒代表在300秒的測試內的所有context switch時間之總和

而因為一個週期（第一個步驟）會經過2個context switch（上圖），我們測300內共有600個context switch，故我們測出每個context switch約為：0.0855 / 600 = 142.5（us）

- interrupt latency

我們的架構為是手動設定一個external interrupt，發生在BUTTON_USER按下時，下面程式是我們的實作：

CORTEX_M4F_STM32_DISCOVERY/main.c

```
i = 0;
while( STM_EVAL_PBGetState( BUTTON_USER ) ){
    i++;
}
```

當BUTTON_USER按下後，會先執行i++直到interruptHandler處理interrupt，讀i值即可得知interrupt latency，而實作結果發現i依舊為0。

- IPC（Inter-Process Communication） throughput

測試程式，在第167行可以改要執行的時間

SysTick最小只能設到1 / 100000 （十萬分之一）秒

若設到1 / 1000000 （百萬分之一）秒，則會連將data copy至queue裡都來不及執行

環境設置：

1. SysTick為1 / 100000 （十萬分之一）秒
2. Queue的length為10000個
3. Queue的ItemSize為uint32_t

- 測試單向（send）

若使用一個task只執行send data的話，在100 SysTicks時間內可以丟入約740個，在1000 SysTicks時間內可以丟入約7500個，

則1 SysTick內平均send 7.5個，故throughput約為：7.5 * 100000 * 4 = 3 （Mbytes/s）

- 測試雙向（send與receive）

若加入一個task來receive data，且priority和send data的priority相同

1000 SysTicks下可以接收到2962個，

則1 SysTick平均接收2.962個，故throughput約為：2.962 * 100000 * 4 = 1.185（Mbytes/s）

- 測試把每個ItemSize做變動

若每個ItemSize為uint16_t，則throughput約為：2.893 * 100000 * 2 = 0.579（Mbytes/s）

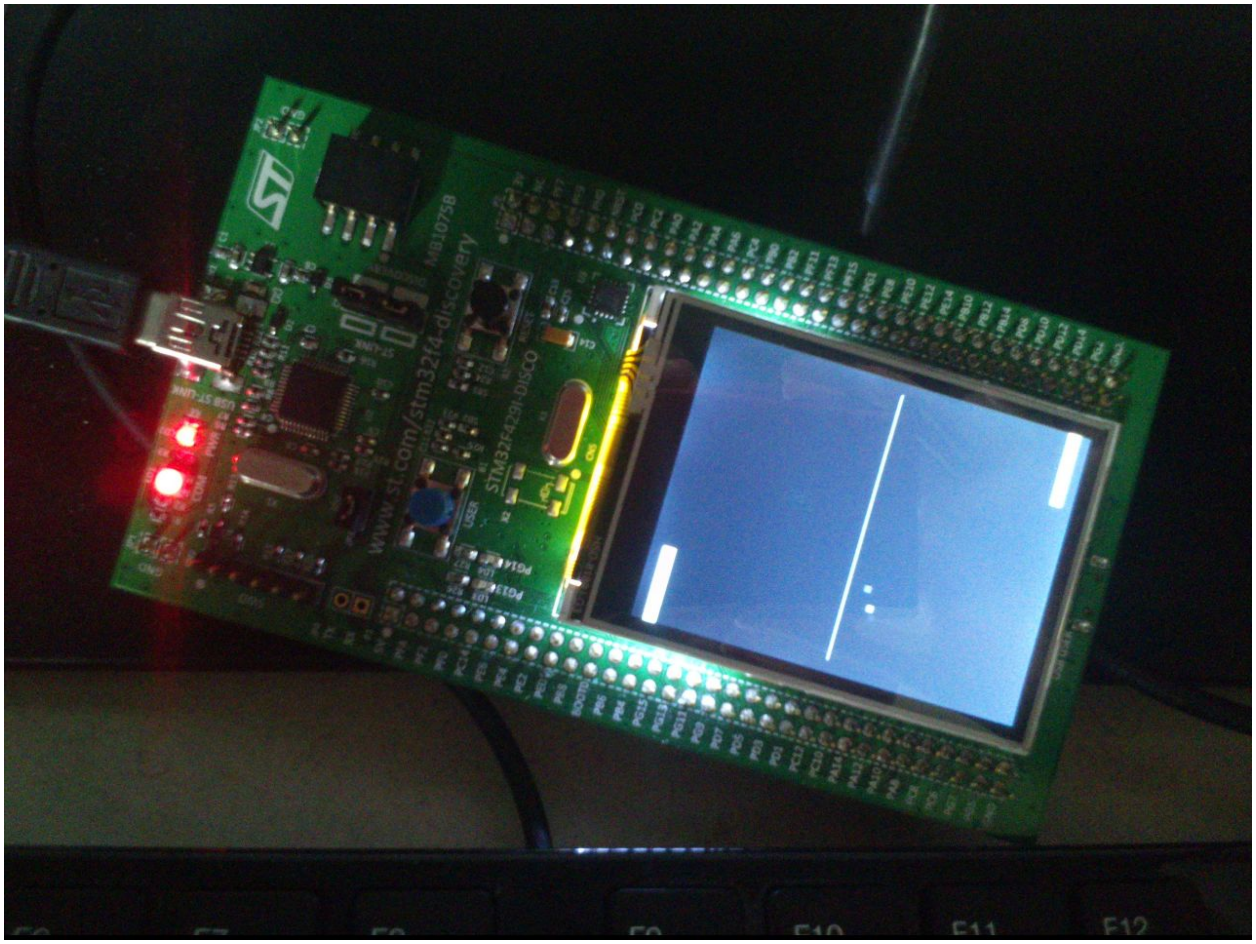
若每個ItemSize為uint64_t，則throughput約為：2.823 * 100000 * 8 = 2.258（Mbytes/s）

SysTick	snd	rcv	snd和rcv的差	ItemSize	throughput
1000	3193	2893	300	uint16_t	0.579
1000	3211	2962	249	uint32_t（原size）	1.185
1000	2992	2823	169	uint64_t	2.258

以上三者比較，在uint64_t時有最好的throughput，且snd和rcv相差最小。

- realtime capability

測試環境架設



安裝

1. 請先安裝st-link以及openOCD，可參考此頁
2. git clone https://github.com/TheKK/myFreeRTOS.git
3. git checkout game
4. make
5. 將stm32 f4 - discovery接上電腦
6. make flash
7. done

解說

- In CORTEX_M4F_STM32F407ZG-SK/game/game.c

主要函式	用途	備註
GAME_EventHandler1	用來偵測按鈕狀態，改變玩家一的行動方式	
GAME_EventHandler2	用來偵測觸控狀態，改變玩家二的行動方式	
GAME_EventHandler3	尚未完成，且沒有用上的函式	
GAME_Update	運算遊戲邏輯且更新資料(ex:球的座標位置)	
GAME_Render	將遊戲物件繪製與畫面上	

- In CORTEX_M4F_STM32F407ZG-SK/game/game.c

在main function中，首先初始化有用上的GPIO裝置，再來呼叫函式xTaskCreate()來建立欲排程的task，xTaskCreate()的輸入參數如下

```
BaseType_t xTaskCreate(
    TaskFunction_t pvTaskCode,          /* 欲排程的函式名稱 */
    const char * const pcName,          /* 在FreeRTOS中顯示的名稱，方便debug用 */
    unsigned short usStackDepth,      /* stack可以容納之變數的數量(不是設定stack的大小) */
    void *pvParameters,                       /* 建立task時的額外參數 */
    unsigned BaseType_t uxPriority,  /* 優先權 */
    TaskHandle_t *pvCreatedTask     /* 該task的參照，可用以檢查是否建立成功 */
);
```

最後，呼叫vTaskStartScheduler()正式開始排程

問題討論

**Q1:進入Suspend 的條件

如果一個task會被block很久或者是會有一段時間用不到那就會被丟到suspend狀態。 情境：設有一個taskPrint這個task只做print資料，而有好幾個taskOperator負責做運算，若運算要很久，則可以把taskPrint先丟入suspend狀態中，直到所有運算皆完成後，再喚醒taskPrint進入ready狀態，最後將資料print出來。

vTaskSuspend使用範例：

```
void vAFunction( void )

{

    TaskHandle_t xHandle;

    // Create a task, storing the handle.

    xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle );

    // ...

    // If xHandle will wait for a long time.

    // Use the handle to suspend the created task.

    vTaskSuspend( xHandle );

    // ...

    // The created task will not run during this period, unless

    // another task calls vTaskResume( xHandle ).

    //...

    // Suspend ourselves.

    vTaskSuspend( NULL );

    // We cannot get here unless another task calls vTaskResume

    // with our handle as the parameter.

}
```

*Q2:Suspend相關程式碼

FreeRTOS提供vTaskSuspend()和vTaskResume()這兩個API來提供我們可以讓task進入suspend狀態。FreeRTOS還有另一個API為vTaskSuspendAll()而主要用途為當某一個task在執行時，某期間內可以讓scheduler被suspend，防止context switch發生，實作方法為控制uxSchedulerSuspended的變數，vTaskSuspendAll()讓uxSchedulerSuspended＋1 進入suspendall的狀態 當程式碼那段執行完時再用vTaskresumeall讓uxSchedulerSuspended-1這用法再很多task裡會用到

例如：vTaskDelay，vTaskDelayUntil為了讓Tasks能順利接上DelayedList而不被中斷，當scheduler被suspend時，context switch會被pending，而在scheduler被suspend的情況下，interrupt不可更改TCB的xStateListItem。而PendingReadyList的用法也是當scheduler被suspend時若這種時候interrupt要解除一個task的block狀態的話，則interrupt需將此task的event list item放至xPendingReadyList中，好讓scheduler解除suspend時，可將xPendingReadyList內的task 放入ready list裡。

Q3:Priority範圍且定義在哪裡

在./CORTEX_M4F_STM32F407ZG-SK/FreeRTOSconfig.h裡

```
#define configMAX_PRIORITIES		( 5 )
```

Q4:為什麼要用DOUBLE LIST LINK

因為DOUBLE LIST LINK在插入新ITEM時擁有常數的時間成本，而SINGLE LIST LINK 則是N (N = 插入位置項數）

Q5：為什麼FREERTOS在FORK之後是回傳一個STRUCK 而不是PID

追朔了xTaskCreate的程式碼，發現他是執行 xTaskGenericCreate這個function，而xTaskGenericCreate是在function裡malloc完成TCB之後，返回值有兩個：

- pdPASS
- errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY

資料型態為BaseType_t，宣告在portmacro.h裡：

```
typedef long BaseType_t;
```

所以他的回傳值用途：回傳告知在malloc memory時是否成功。

而linux使用回傳PID的原因在於parent使用wait()來等待child，當child執行結束後會呼叫exit()，parent即可以clean up child process。若parent沒有使用wait()的話，會造成parent可能已經先結束了，這樣造成child變成zombie。

FreeRTOS的task create：

```
xTaskCreate( vTaskCode, "NAME", STACK_SIZE, &ucParameterToPass, tskIDLE_PRIORITY,
&xHandle );
```

其中Handle存的是新創的TCB這個structure的位址，將來要刪除此task的話可以用如下方法：

```
/* Use the handle to delete the task. */
if( xHandle != NULL
{
    vTaskDelete( xHandle );
}
```

而Linux的parent和child為相同的位址空間，若回傳為child的位址，將來parent要把child刪除時，便也把自己給刪除了…所以linux使用的是PID而不是structure的位址。

Q6：STACK位置的排列，如何存放

存放順序：

xPSR

PC : Program counter 內容存放處理器要存取的下一道指令位址 LR : link rigisiter ：保存副程式的返回位址 R12 : Intra-Procedure-call scratch register R3 : parameters R2 : parameters R1 : parameters R0 : parameters

portINITIAL_EXEC_RETURN : 每個task要自己維護自己的返回值

- R11
- R10
- R9
- R8
- R7
- R6
- R5
- R4

註：xPSR：Composite of the 3 PSRs，

APSR-Application Program Status Register–condition flags

（存negative、zero、carry、overflow等）

IPSR-Interrupt Program Status Register–Interrupt/Exception No.

（存目前發生Exception的ISR Number ）

EPSR-Execution Program Status Register

（存Thumb state bit 和 execution state bits(If-Then (IT) instruction和 Interruptible-Continuable Instruction (ICI) field)）

Q7：LR(Link Rigisiter) 的用處

當一個task A 執行被中斷時（可能system tick 或是高優先權的Task出現）用來紀錄Task A執行到哪裡的位置，當其他程式執行完時，能返回繼續成行Task A

Q8：為什麼是R12 R3 R2 R1 要預留起來？

R0~R3用來暫存Argument 的 scratch rigister （4個register的原因是為了handle values larger than 32 bits）

R0 R1 亦可暫存 subroutine 的result值

R12 : 作為The Intra-Procedure-call scratch register.

而為什麼是這幾個，因為叫方便使用

R12（IP）用法：

```
mov     ip, lr
bl      lowlevel_init
mov     lr, ip
```

先將lr暫存存入ip

bl跳至其他branch的地方

branch結束後使用lr跳回第三行，將ip存回lr

P.S. 關於veneer： ARM 能支援 32-bit 和 16-bit 指令互相切換（THUMB 是 ARM 的 16-bit 指令集），其中切換的程式段叫 veneer

Q9：誰把New Task 接到 Ready List

GDB Trace result

```
Breakpoint 1, xTaskGenericCreate (pxTaskCode=0x80003b1 <GameTask>, pcName=0x800ea84
"GameTask", usStackDepth=128, pvParameters=0x0, uxPriority=1, pxCreatedTask=0x0,
puxStackBuffer=0x0,
xRegions=0x0) at /home/kk/myPrograms/embedded/myFreeRTOS/tasks.c:516
516         {
(gdb) next
520                 configASSERT( pxTaskCode );
(gdb)
516         {
(gdb)
520                 configASSERT( pxTaskCode );
(gdb)
521                 configASSERT( ( ( uxPriority & ( ~portPRIVILEGE_BIT ) ) <
configMAX_PRIORITIES ) );
(gdb)
525                 pxNewTCB = prvAllocateTCBAndStack( usStackDepth, puxStackBuffer );
(gdb)
572                         prvInitialiseTCBVariables( pxNewTCB, pcName, uxPriority,
xRegions, usStackDepth );
(gdb)
551                                 pxTopOfStack = pxNewTCB->pxStack + ( usStackDepth - (
uint16_t ) 1 );
(gdb)
572                         prvInitialiseTCBVariables( pxNewTCB, pcName, uxPriority,
xRegions, usStackDepth );
(gdb)
551                                 pxTopOfStack = pxNewTCB->pxStack + ( usStackDepth - (
uint16_t ) 1 );
(gdb)
572                         prvInitialiseTCBVariables( pxNewTCB, pcName, uxPriority,
xRegions, usStackDepth );
(gdb)
551                                 pxTopOfStack = pxNewTCB->pxStack + ( usStackDepth - (
uint16_t ) 1 );
(gdb)
552                                 pxTopOfStack = ( StackType_t * ) ( ( (
portPOINTER_SIZE_TYPE ) pxTopOfStack ) & ( ( portPOINTER_SIZE_TYPE )
```

```
~portBYTE_ALIGNMENT_MASK  ) ); /*lint !e923 MISRA exception.  Avoiding casts between
pointers and integers is not practical.  Size differences accounted for using
portPOINTER_SIZE_TYPE type. */
(gdb)
572                         prvInitialiseTCBVariables( pxNewTCB, pcName, uxPriority,
xRegions, usStackDepth );
(gdb)
584                                 pxNewTCB->pxTopOfStack = pxPortInitialiseStack(
pxTopOfStack, pxTaskCode, pvParameters );
(gdb)
588                         if( ( void * ) pxCreatedTask != NULL )
(gdb)
602                         taskENTER_CRITICAL();
(gdb)
604                                 uxCurrentNumberOfTasks++;
(gdb)
605                                 if( pxCurrentTCB == NULL )
(gdb)
604                                 uxCurrentNumberOfTasks++;
(gdb)
605                                 if( pxCurrentTCB == NULL )
(gdb)
604                                 uxCurrentNumberOfTasks++;
(gdb)
605                                 if( pxCurrentTCB == NULL )
(gdb)
609                                         pxCurrentTCB =  pxNewTCB;
(gdb)
611                                         if( uxCurrentNumberOfTasks == ( UBaseType_t )
1 )
(gdb)
(gdb)
616                                                 prvInitialiseTaskLists();
(gdb)
645                                 uxTaskNumber++;
(gdb)
655                                 prvAddTaskToReadyList( pxNewTCB );
(gdb)
645                                 uxTaskNumber++;
```

```
(gdb)
655                                     prvAddTaskToReadyList( pxNewTCB );
(gdb)
645                                     uxTaskNumber++;
(gdb)
655                                     prvAddTaskToReadyList( pxNewTCB );
(gdb)
645                                     uxTaskNumber++;
(gdb)
655                                     prvAddTaskToReadyList( pxNewTCB );
(gdb)
645                                     uxTaskNumber++;
(gdb)
650                                             pxNewTCB->uxTCBNumber = uxTaskNumber;
(gdb)
655                                     prvAddTaskToReadyList( pxNewTCB );
(gdb)
660                             taskEXIT_CRITICAL();
(gdb)
670                             if( xSchedulerRunning != pdFALSE )
(gdb)
657                                     xReturn = pdPASS;
(gdb)
690
```

- prvInitialiseTaskLists(void)

只有在list未被初始化時，才會被執行。預設會建立pxReadyTasksLists，xDelayedTaskList1，xDelaye traceMOVED_TASK_TO_READY_STATEdTaskList2，xPendingReadyList，依照使用者設定可以選擇是否建立xTasksWaitingTermination和xSuspendedTaskList

- prvAddTaskToReadyList(pxNewTCB)

將pxNewTCB接上pxReadyTasksLists，prvAddTaskToReadyList()程式碼如下

```
#define prvAddTaskToReadyList( pxTCB ) \
        traceMOVED_TASK_TO_READY_STATE( pxTCB ) \
        taskRECORD_READY_PRIORITY( ( pxTCB )->uxPriority ); \
        vListInsertEnd( &( pxReadyTasksLists[ ( pxTCB )->uxPriority ] ), &( ( pxTCB )->xGenericListItem ) )
```

- traceMOVED_TASK_TO_READY_STATE

```
#ifndef traceMOVED_TASK_TO_READY_STATE
        #define traceMOVED_TASK_TO_READY_STATE( pxTCB )
#endif
```

自定義函式，無預設定義。Debug用

- taskRECORD_READY_PRIORITY

```
#define taskRECORD_READY_PRIORITY( uxPriority  ) \
{ \
    if( ( uxPriority ) > uxTopReadyPriority ) \
    { \
        uxTopReadyPriority = ( uxPriority ); \
    } \
} /* taskRECORD_READY_PRIORITY */
```

檢查目前task的priority是否高於“當前最高優先權”。如果是，將更新當前最高優先權。

- vListInsertEnd

```
void vListInsertEnd( List_t * const pxList, ListItem_t * const pxNewListItem )
{
ListItem_t * const pxIndex = pxList->pxIndex;

        /* Insert a new list item into pxList, but rather than sort the list,
        makes the new list item the last item to be removed by a call to
        listGET_OWNER_OF_NEXT_ENTRY(). */
        pxNewListItem->pxNext = pxIndex;
        pxNewListItem->pxPrevious = pxIndex->pxPrevious;
        pxIndex->pxPrevious->pxNext = pxNewListItem;
        pxIndex->pxPrevious = pxNewListItem;

        /* Remember which list the item is in. */
        pxNewListItem->pvContainer = ( void * ) pxList;
        ( pxList->uxNumberOfItems )++;
}
```

將pxNewListItem插入至pxList的最後面

參考資料

- The Architecture of Open Source Applications: FreeRTOS
    - 簡體中文翻譯
- Study of an operating system: FreeRTOS
- FreeRTOS 即時核心實用指南

---

1. 用 cloc 統計 FreeRTOS 8.0.0 的 include/ *.c portable/GCC/ARM_CM4F/ 等目錄，可得不含註解、空白行的行數為 6566，而統計與平台有關的部份 (portable/GCC/ARM_CM4F/ 目錄)，則是 435 行。計算可得: 435 / 6566 = 0.06625 = 6%，與描述相符，但原本的 9000 行是指含註解的說法 (實際為 8759 行)↩