FreeRTOS

協作者

- 2015 年春季
- 蕭奕凱, 張瑋豪, 余誌偉, 王皓昱, 洪煒凱
- 2014 年春季
- 梁穎睿, 李奇霖, 方威迪, 陳盈伸

共筆

- 2015 年春季
- Hackpad
- GitHub
- 2014 年春季
- Hackpad
- GitHub

目錄

FreeRTOS 架構

官方網站：http://www.freertos.org/ (http://www.freertos.org/)

FreeRTOS 是一個相對其他作業系統而言較小的作業系統。最小化的 FreeRTOS 核心僅包括 3 個 .c 文件(tasks.c、queue.c、list.c)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 binary（二進位碼）小於 10 KB。

FreeRTOS 的程式碼可以分為三個主要區塊：任務、通訊和硬體界面。

- 任務 (Task): FreeRTOS 的核心程式碼約有一半是用來處理多數作業系統首要關注的問題：任務，任務是擁有優先權的用戶所定義的 C 函數。task.c 和 task.h 負責所有關於建立、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！它帶出了 FreeRTOS 的第二項議題：通訊。FreeRTOS 核心程式碼大約有 40% 是用來處理通訊的。queue.c 和 queue.h 負責處理 FreeRTOS 的通訊，任務和中斷(interrupt)使用佇列(佇列，queue)互相發送數據，並且使用 semaphore 和 mutex 來派發 critical section 的使用信號。
- 硬體界面：有近 9000 行的程式碼組成基本的 FreeRTOS，這部份是與硬體無關的(hardware-independent)，同一份程式碼在不同硬體平台上的 FreeRTOS 都可以運行。大約有 6% 的 FreeRTOS 核心代碼，在與硬體無關的 FreeRTOS 核心和與硬體相關的程式碼間扮演著墊片(shim (http://en.wikipedia.org/wiki/Shim_%28computing%29))的角色。我們將在下個部分討論與硬體相關的程式碼。[#]_

.. [#] 用 cloc (http://cloc.sourceforge.net/) 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)

原始碼

- 官方下載：https://sourceforge.net/projects/freertos
- 最新版本：v8.2.1

- 核心程式碼：(Source)
- tasks.c：主要掌管 task 的檔案
- queue.c：管理 task 間 communication (message queue 的概念)
- list.c：提供系統與應用實作會用到的 list 資料結構

```
- 選擇性檔案：timer.c、croutine.c (co-routine)、event_groups.c
```

- 與硬體相關的檔案：以 ARM Cortext-M3 為例，可在 Source/portable/GCC/ARM_CM3 中找到
- portmacro.h：定義了與硬體相關的變數，如資料型態定義，以及與硬體相關的函式呼叫名稱定義(以 portXXXXX 命名)等，統一各平臺的函式呼叫
- port.c：定義了包含與硬體相關的程式碼實作

- FreeRTOSConfig.h：包含 clock speed, heap size, mutexes 等等都在此定義(需自行建立)

回目錄

資料型態及命名規則

在不同硬體裝置上，通訊埠設定上也不同，定義在`portmacro.h`標頭檔內，有兩種特殊資料型態portTickType以及portBASE_TYPE。

- 資料型態
- portTickType : 用以儲存tick的計數值，可以用來判斷block次數
- portBASE_TYPE : 定義為架構基礎的變數，隨各不同硬體來應用，如在32-bit架構上，其為32-bit型態，最常用以儲存極限值或布林數。

FreeRTOS明確的定義變數名稱以及資料型態，不會有unsigned以及signed搞混使用的情形發生。

- 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
- portBASE_TYPE 或其他（如 struct）：以 x 為字首
- pointer 有一個額外的字首 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

- 函式：以回傳值型態與所在檔案名稱為開頭(prefix)
- vTaskPriority() 是 task.c 中回傳值型態為 void 的函式
- xQueueReceive() 是 queue.c 中回傳值型態為 portBASE_TYPE 的函式
- 只能在該檔案中使用的 (scope is limited in file) 函式，以 prv 為開頭 (private)

- 巨集名稱：巨集在FreeRTOS裡皆為大寫字母定義，名稱前小寫字母為巨集定義的地方
- portMAX_DELAY : portable.h
- configUSE_PREEMPTION : FreeRTOSConfig.h

一般巨集回傳值定義pdTRUE 及 pdPASS為1 , pdFALSE 及 pdFAIL 為0。

Run FreeRTOS on STM32F4-Discovery

回目錄

任務

任務 (task) 是在 FreeRTOS 中執行的基本單位，每個 task 都是由一個 C 函數所組成，意思是你需要先定義一個 C 的函數，然後再用 xTaskCreate() 這個 API 來建立一個 task，這個 C 函數有幾個特點，它的返回值必須是 void，其中通常會有一個無限迴圈，所有關於這個 task 的工作都會在迴圈中進行，而且這個函數不會有 return，FreeRTOS 不允許 task 自行結束(使用 return 或執行到函數的最後一行)

Task 被建立出來後，它會配置有自己的堆疊空間和 stack variable(就是 function 中定義的變數)

一個典型的 task function 如下：

.. code-block:: c

```
void ATaskFunction(void *pvParameters)
{
   int i = 0;   // 每個用這個函數建立的 task 都有自己的一份 i 變數

   while(1)
   { /* do something here */ }

   /*
    * 如果你的 task 就是需要離開 loop 並結束
    * 需要用 vTaskDelete 來刪除自己而非使用 return 或自然結束(執行到最後一行)
    * 這個參數的 NULL 值是表示自己
    */
   vTaskDelete(NULL);
}
```

Task 的狀態

- Ready：準備好要執行的狀態
- Running：正在由 CPU 執行的狀態
- Blocked：等待中的狀態(通常是在等待某個事件)
- Suspended：等待中的狀態(透過 API 來要求退出排程)

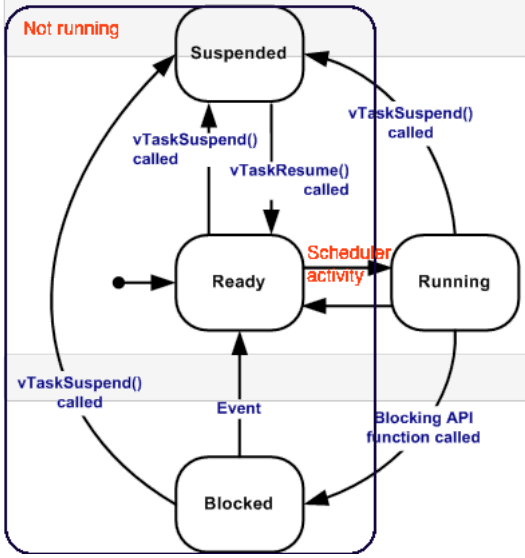
每一種狀態 FreeRTOS 都會給予一個 list 儲存（除了 running)

- 建立 task 的函數

.. code-block:: c

```
portBASE_TYPE xTaskCreate( pdTASK_CODE pvTaskCode,
                           const signed portCHAR * const pcName,
                           unsigned portSHORT usStackDepth,
```

```
                                    void *pvParameters,
                                    unsigned portBASE_TYPE uxPriority,
                                    xTaskHandle *pxCreatedTask );
```



- pvTaskCode：就是我們定義好用來建立 task 的 C 函數
- pcName：任意給定的 task name，這個名稱只被用來作識別，不會在 task 管理中被採用
- usStackDepth：堆疊的大小
- pvParameters：要傳給 task 的參數陣列，也就是我們在 C 函數宣告的參數
- uxPriority：定義這個任務的優先權，在 FreeRTOS 中，0 最低，(configMAX_PRIORITIES – 1) 最高
- pxCreatedTask：handle (http://zh.wikipedia.org/wiki/%E5%8F%A5%E6%9F%84)，是一個被建立出來的 task 可以用到的識別符號

- 刪除 task 的函數

.. code-block:: c

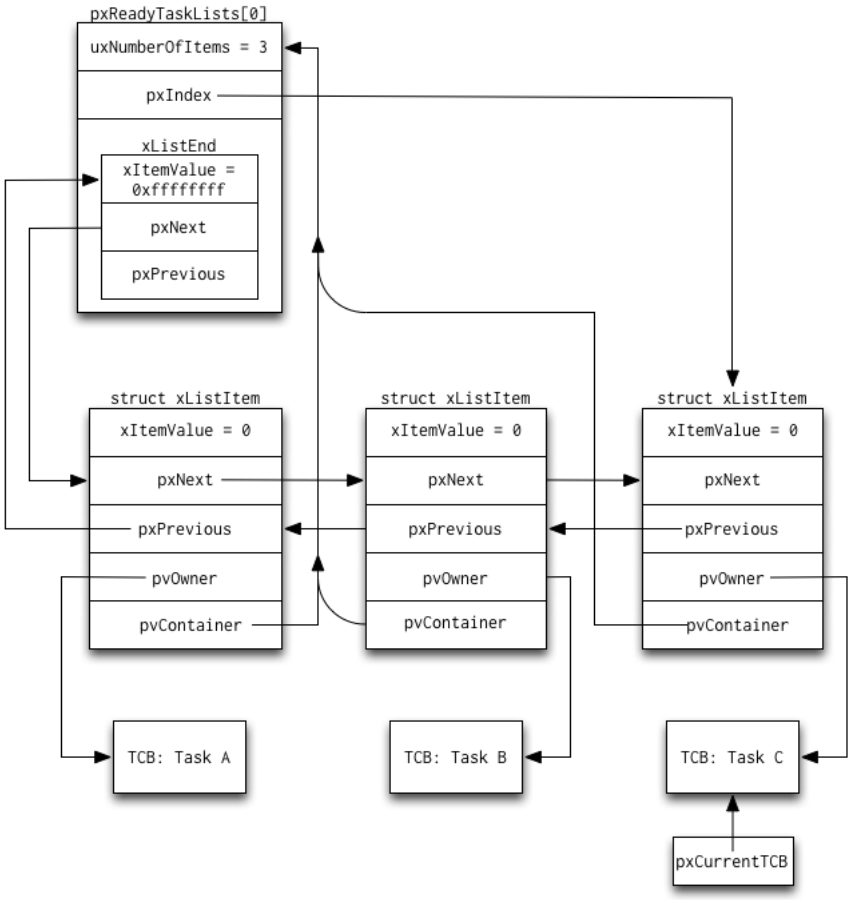
```
void vTaskDelete( xTaskHandle pxTaskToDelete );
```

- pxTaskToDelete: 利用handle去識別出哪一個task。 這種可能性存在於如果在 loop 中發生執行錯誤(fail)，則需要跳出迴圈並終止(自己)執行，此時就需要使用 vTaskDelete 來刪除自己，發生錯誤的例子：

1. 假如今天一個 task 是要存取資料庫，但是資料庫或資料表不存在，則應該結束 task
2. 假如今天一個 client task 是要跟 server 做連線(listening 就是 loop)，卻發現 client 端沒有網路連線，則應結束 task

Ready list 的資料形態

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



OS 會在進行 context switch 時選出下一個欲執行的 task

下面是在 ready list 中依照優先權選取執行目標的程式部分，FreeRTOS 的優先權最小爲 0，數字越大則優先權越高

task.c

.. code-block:: c

```
#define taskSELECT_HIGHEST_PRIORITY_TASK()
{
    /* 選出含有 ready task 的最高優先權 queue */
    while( listLIST_IS_EMPTY( &( pxReadyTasksLists[ u
xTopReadyPriority ] ) ) )
    {
        configASSERT( uxTopReadyPriority );
//如果找不到則 assert exception
        --uxTopReadyPriority;
    }

    /* listGET_OWNER_OF_NEXT_ENTRY indexes through th
e list, so the tasks of
       the same priority get an equal share of the pr
ocessor time. */
        listGET_OWNER_OF_NEXT_ENTRY( pxCurrentTCB, &(
pxReadyTasksLists[ uxTopReadyPriority ] ) );
}
```

從前面的圖我們可以知道一個 ready task list 中的每個索引各自指向了一串 task list，所以 listGET_OWNER_OF_NEXT_ENTRY 就是在某個 ready task list 索引中去取得其中 task list 裡某個 task 的 TCB

include/list.h

.. code-block:: c

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

- Task Control Block (TCB)的資料結構(tasks.c)

.. code-block:: c

```
/* In file: tasks.c */
typedef struct tskTaskControlBlock
{
    volatile portSTACK_TYPE *pxTopOfStack;                  /* 指向 task 記憶體堆疊最後一個項目的位址，這必須是 struct 中的第一個項目 (有關 offse
t) */
    xListItem    xGenericListItem;                          /* 用來記錄 task 的 TCB 在 FreeRTOS ready 和 blocked queue 的位置 */
    xListItem    xEventListItem;                            /* 用來記錄 task 的 TCB 在 FreeRTOS event queue 的位置 */
    unsigned portBASE_TYPE uxPriority;                      /* task 的優先權 */
    portSTACK_TYPE *pxStack;                                /* 指向 task 記憶體堆疊的起始位址 */
    signed char    pcTaskName[ configMAX_TASK_NAME_LEN ];   /* task 被建立時被賦予的有意義名稱(為了 debug 用)*/

    #if ( portSTACK_GROWTH > 0 )
    portSTACK_TYPE *pxEndOfStack;                           /* stack overflow 時作檢查用的 */
    #endif

    #if ( configUSE_MUTEXES == 1 )
    unsigned portBASE_TYPE uxBasePriority;                  /* 此 task 最新的優先權 */
    #endif
} tskTCB;
```

- pxTopOfStack , pxEndOfStack：記錄 stack 的大小
- uxPriority , uxBasePriority：前者記錄目前的優先權 ,後者記錄原本的優先權（可能發生在 Mutex)
- xGenericListItem , xEventListItem：當一個任務被放入 FreeRTOS 的一個列表中，FreeRTOS 在 TCB 中插入指向這個任務的 pointer 的地方

xTaskCreate() 函數被呼叫的時候，一個任務會被建立。FreeRTOS 會為每一個任務分配一個新的 TCB target，用來記錄它的名稱、優先權和其他細節，接著配置用戶所請求的 HeapStack 空間（假設有足夠使用的記憶體），並在 TCB 的 pxStack 成員中記錄 Stack 的記憶體起始位址。

- 配置TCB及stack的函式 - prvAllocateTCBAndStack()(tasks.c)

.. code-block:: c

```
/* stack 記憶體由高到低 */
#if( portSTACK_GROWTH > 0 )
{
        /* pvPortMalloc 就是做記憶體配置*/
        pxNewTCB = ( TCB_t * ) pvPortMalloc( sizeof( TCB_t ) );

        /* 如果pxNewTCB為空*/
        if( pxNewTCB != NULL )
        {
                /* 做記憶體對齊動作  */
                pxNewTCB->pxStack = ( StackType_t * ) pvPortMallocAligned( ( ( ( size_t ) usStackDepth ) * sizeof( StackType_t ) ), puxS
tackBuffer );

                /* Stack 仍為NULL的話，則刪除TCB，並且指向NULL */
                if( pxNewTCB->pxStack == NULL )
                {

                        vPortFree( pxNewTCB );
                        pxNewTCB = NULL;
                }
        }
}
```

在進行pvPortMalloc時候，會先進行vTaskSuspendAll(); ，藉由不發生context swiitch的swap out動作，配置記憶體空間，等到配置完成再呼叫xTaskResumeAll();

- pvPortMalloc在port.c裡面定義，基本上就是做記憶體配置，根據各不同port去實作pvPortMalloc。
- pvPortMallocAligned在在FreeRTOS.h裡面可以看到define為如果判斷未配置空間，則進行pvPortMalloc，如果有則直接使用puxStackBuffer。

- 硬體層次的設定

爲了便於排程，創造新 task 時，stack 中除了該有的資料外，還要加上『空的』 register 資料(第一次執行時理論上 register 不會有資料)，讓新 task 就像是被 context switch 時選的 task 一樣，依照前述變數的命名規則，下面是實作方式

portable/GCC/ARM_CM4F/port.c

.. code-block:: 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，這個過程中必須暫時停止 interrupt 功能，以免在 list 還沒設定好前就被中斷設定(例如 systick)

而 ARM Cortex-M4 處理器在 task 遇到中斷時，會將 register 的內容 push 進該 task 的 stack 的頂端，待下次執行時再 pop 出去，以下是在 port.c 裡的實作

portable/GCC/ARM_CM4F/port.c

.. code-block:: c

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

        __asm volatile
        (
        "       mrs r0, psp                                                     \n" // psp: Process Stack Pointer
        "       isb                                                             \n"
        "                                                                       \n"
        "       ldr     r3, pxCurrentTCBConst                                   \n" /* Get the location of the current TCB. */
        "       ldr     r2, [r3]                                                \n"
        "                                                                       \n" // tst used "and" to test.
        "       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: db means "decrease before"
        "       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 TC
B. */
        "                                                                       \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" // r3 now is switched to the higher priority task
        "                                                                       \n"
        "       ldr r1, [r3]                                                    \n" /* The first item in pxCurrentTCB is the task top of stack.
*/
        "       ldr r0, [r1]                                                    \n" // this r0 is "pxTopOfStack"
        "                                                                       \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 v
fp 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" //number X must be a power of 2. That is 2, 4, 8, 16, and so
on...
        "       .align 2                                                        \n" //on a memory address that is a multiple of the value X
        "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，該呼叫的函數名稱為 ulPortSetInterruptMask()

portable/GCC/ARM_CM4F/port.c

.. code-block:: 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 遮罩掉所有的 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 (http://www.freertos.org/RTOS-Cortex-M3-M4.html)

當使用 vTaskCreate() 將 task 被建立出來以後，需要使用 vTaskStartScheduler() 來啟動排程器決定讓哪個 task 開始執行，當 vTaskStartScheduler() 被呼叫時，會先建立一個 idle task，這個 task 是為了確保 CPU 在任一時間至少有一個 task 可以執行 (取代直接切換回 kernel task) 而在 vTaskStartScheduler() 被呼叫時自動建立的 user task，idle task 的 priority 為 0 (lowest)，目的是為了確保當有其他 user task 進入 ready list 時可以馬上被執行

vTaskStartScheduler (tasks.c)

.. code-block:: c

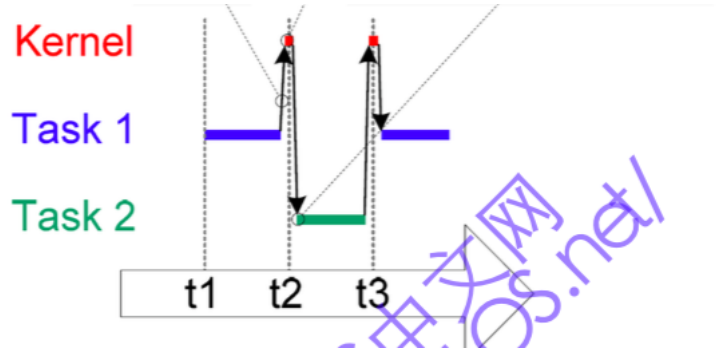
```
/* Add the idle task at the lowest priority. */
#if ( INCLUDE_xTaskGetIdleTaskHandle == 1 )
{
    /* Create the idle task, storing its handle in xIdleTaskHandle so it can
    be returned by the xTaskGetIdleTaskHandle() function. */
    xReturn = xTaskCreate( prvIdleTask, "IDLE", tskIDLE_STACK_SIZE,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

接著才呼叫 xPortStartScheduler() 去執行 task

- Blocked

Task 的 blocked 狀態通常是 task 進入了一個需要等待某事件發生的狀態，這個事件通常是執行時間到了(例如 systick interrupt)或是同步處理的回應，如果像一開始的 ATaskFunciton() 中使用 while(1){} 這樣的無限迴圈來作等待事件，會占用 CPU 運算資源，也就是 task 實際上是在 running，但又沒做任何事情，占用著資源只為了等待 event，所以比較好的作法是改用 vTaskDelay()，當 task 呼叫了 vTaskDelay()，task 會進入 blocked 狀態，就可以讓出 CPU 資源了

使用 infinite loop 的執行時序圖



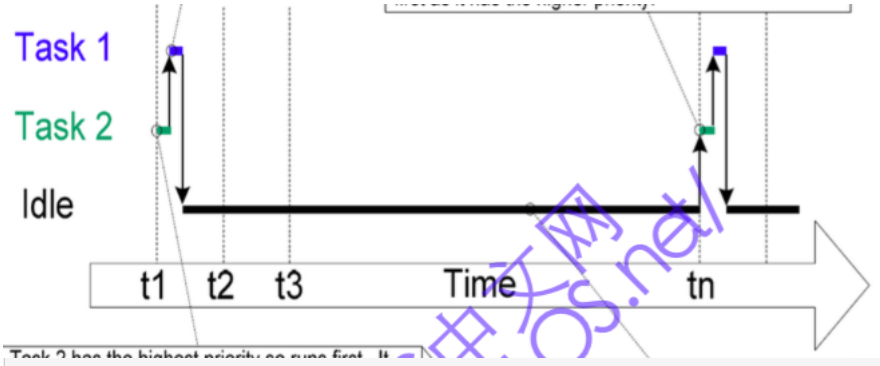
使用 vTaskDelay() 的執行時序圖

- vTaskDelay()：這個函式的參數如果直接給數值，是 ticks，例如 vTaskDelay(250) 是暫停 250 個 ticks 的意思，由於每個 CPU 的一個 tick 時間長度不同，FreeRTOS 提供了 portTICK_RATE_MS 這個巨集常數，可以幫我們轉換 ticks 數為毫秒 (ms)，也就是說 vTaskDelay(250/portTICK_RATE_MS) 這個寫法，就是讓 task 暫停 250 毫秒(ms)的意思 (v8.2.1 改名為 portTICK_PERIOD_MS)
- Suspended

  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：

  .. code-block:: c




```
 void vAFunction( void )
 {
     TaskHandle_t xHandle;
     // Create a task, storing the handle.
     xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NUL
L, 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 ).
     //...
```

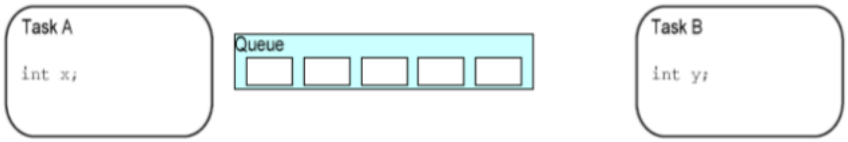
```
    // Suspend ourselves.
    vTaskSuspend( NULL );
    // We cannot get here unless another task calls vTaskResume
    // with our handle as the parameter.
}
```

回目錄

通訊

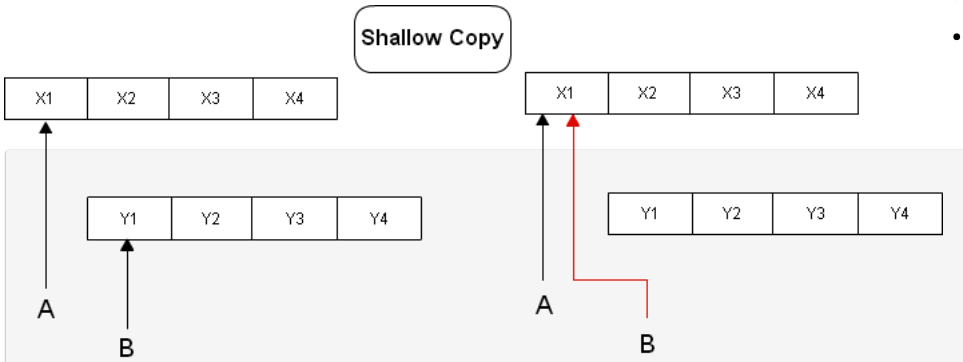
在 FreeRTOS 中，task 之間的溝通是透過把資料傳送到 queue 和讀取 queue 中資料實現的

- Queue

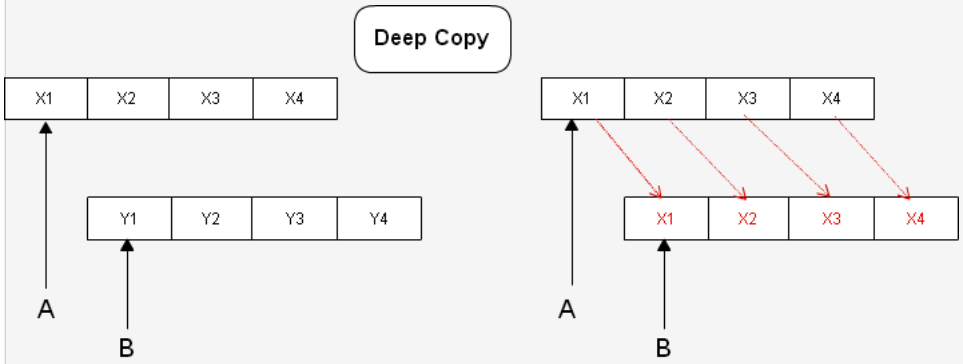


FreeRTOS 的 task 預設是採用 deep copy 的方式來將資料送到 queue 中，也就是會把資料按照字元一個一個複製一份到 queue 中，當要傳遞的資料很大時，建議不要這樣傳遞，改採用傳遞資料指標的方式，如 shallow copy。好處是可以直接做大資料複製，缺點就是會影響到記憶體內容。不管你選擇哪一個方式放入 queue 中，FreeRTOS 只關心 data 的大小，不關心是哪一種 data copy 的方式。

- Shallow Copy



- Deep Copy



- Queue 的結構

queue.c

.. code-block:: c

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

  signed char *pcHead;                    /* P
oints to the beginning of the queue
                                             s
torage area. */

  signed char *pcTail;                    /* P
oints to the byte at the end of the
                                             q
  ueue storage area. One more byte is

  allocated than necessary to store the
                                              qu
  eue 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;
```

由於一個 queue 可以被多個 task 寫入（即 send data），所以有 xTasksWaitingToSend 這個 list 來追蹤被 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，系統就會檢查 xTasksWaitingToSend，看看是否有等待中的 task 在 list 裡，並在這些 task 中選出一個 priority 最高的，讓它恢

復執行來進行寫入資料的動作，若這些 task 的 priority 都一樣，那會挑等待最久的 task。

有許多 task 要從 queue 中讀取資料時也是一樣（即receive data），若 queue 中沒有任何 item，而同時還有好幾個 task 想要讀取資料，則這些 task 會被加入 xTasksWaitingToReceive 裡，每當有一個 item 被放入 queue 中，系統一樣去檢查 xTasksWaitingToReceive，看看是否有等待中的 task 在 list 裡，並在這些 task 中選出一個 priority 最高的，讓它恢復執行來進行讀取資料的動作，若這些 task 的 priority 都一樣，那會挑等待最久的 task。

- Queue 的用法

/CORTEX_M4F_STM32_DISCOVERY/main.c

.. code-block:: 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 Source/queue.c

FreeRTOS 中也可使用 queue 來實作 semaphore 和 mutex：

```
* Semaphores - 用來讓一個 task 喚醒喚醒(wake)另一個 task，例如: producer 和 consumer
* Mutexes - 用來對共享資源(critical section)做互斥存取
```

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/

- 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。

需要用到 ARM Cortex-M4F 特有的機制，才能實做 semaphore，這個機制為『在存取 uxMessagesWaiting 時必須確保同一時間只能有一個 task 在做更改（進出入 critical section）』，要防止一次兩個 task 進入修改的方法如下：

portable/GCC/ARM_CM4F/port.c

.. code-block:: 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，所以用 overloadind 來達到較好的使用率：

queue.c

.. code-block:: c

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

```
- uxQueueType 若為 0，表示這個 queue 已經被用來當作 mutex
- pxMutexHolder 用來實作 priority inheritance
```

補充: http://embeddedgurus.com/barr-code/2008/03/rtos-myth-3-mutexes-are-needed-at-the-task-level/

- 生產者與消費者

使用 FreeRTOS 的 semaphore 和 mutex 來實作生產者與消費者問題：

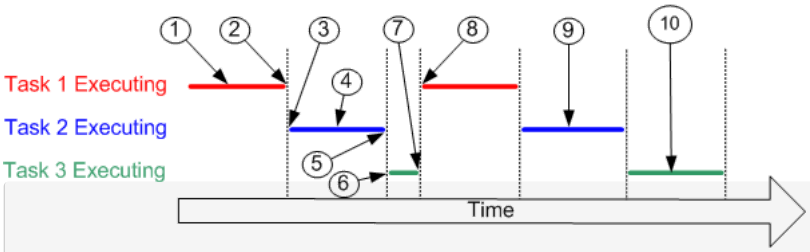
.. code-block:: c

```
SemaphoreHandle_t xMutex = NULL;
SemaphoreHandle_t empty = NULL;
SemaphoreHandle_t full = NULL;
xQueueHandle buffer = NULL;
long sendItem = 1;
long getItem = -1;

void Producer1(void* pvParameters){
        while(1){
                // initial is 10, so producer can push 10 item
                if( xSemaphoreTake(empty, portMAX_DELAY) == pdTRUE ){

                        if( xSemaphoreTake(xMutex, portMAX_DELAY) == pdTRUE ){
                        /******************** enter critical section ********************/
                                xQueueSend( buffer, &sendItem, 0 );
                                USART1_puts("add item, buffer = ");
                                itoa( (long)uxQueueSpacesAvailable(buffer), 10);
                                sendItem++;
                        /******************** exit critical section ********************/
                                xSemaphoreGive(xMutex);
                        }
                        // give "full" semaphore
                        xSemaphoreGive(full);
                }
                vTaskDelay(90000);
        }
}

void Consumer1(void* pvParameters){
        while(1){
                // initial is 0 so consumer can't get any item
                if( xSemaphoreTake(full, portMAX_DELAY) == pdTRUE ){

                        if( xSemaphoreTake(xMutex, portMAX_DELAY) == pdTRUE ){
                        /******************** enter critical section ********************/
                                xQueueReceive( buffer, &getItem, 0 );
                                USART1_puts("get: ");
                                itoa(getItem, 10);
                        /******************** exit critical section ********************/
                                xSemaphoreGive(xMutex);
                        }
                        // give "empty" semaphore
                        xSemaphoreGive(empty);
                }
                vTaskDelay(80000);
        }
}
```

回目錄

排程

- 基本概念



FreeRTOS 中除了由 kernel 要求 task 交出 CPU 控制權外，task 也能夠可以自行交出 CPU 控制權

Delay(sleep): 暫停執行一段時間

/CORTEX_M4F_STM32_DISCOVERY/main.c

.. code-block:: c

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

                iii = 0;
        }
}
```

使用 vTaskDelay(ticks) 會將目前 task 的 ListItem 從 ReadyList 中移除並放入 DelayList 或是 OverflowDelayList 中(由現在的 systick 加上欲等待的 systick 有無 overflow 決定)，但 task 不是在呼叫了 vTaskDelay() 後馬上交出 CPU 控制權，而是在下一次的 systick interrupt 才釋出

wait(block): 等待取得資源或事件發生

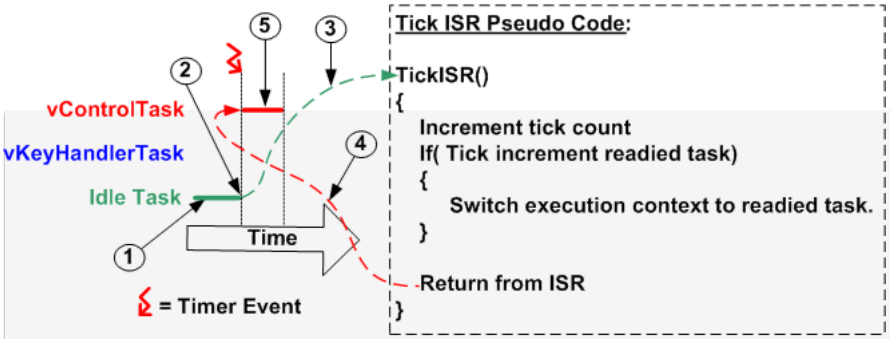
.. code-block:: c

```
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 );
                }
        }
}
```

使用 xQueueReceive(xQueue, *pvBuffer, xTicksToWait)，等待時間還沒到 portMAX_DELAY(FreeRTOS 最長的等待時間)時，task 會被放入 EventList 中等待取得資源或事件發生。若等待時間到了 portMAX_DELAY，則會被移到 SuspendList 中繼續等待

- RTOS tick



portable/GCC/ARM_CM4F/port.c

實作細節

.. code-block:: c

```
void xPortSysTickHandler( void )
{
    /* The SysTick runs at the lowest interrupt prio
rity, so when this interrupt
    executes all interrupts must be unmasked.  There
is therefore no need to
    save and then restore the interrupt mask value a
s 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 );
}
```

.. code-block:: c

```
BaseType_t xTaskIncrementTick( void )
{
    TCB_t * pxTCB;
    TickType_t xItemValue;
    BaseType_t xSwitchRequired = pdFALSE;

    traceTASK_INCREMENT_TICK( xTickCount );
    if( uxSchedulerSuspended == ( UBaseType_t ) pdFALSE )
    {
        ++xTickCount;

        {
            const TickType_t xConstTickCount = xTickCount;

            if( xConstTickCount == ( TickType_t ) 0U )
            {
                taskSWITCH_DELAYED_LISTS();
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }

            if( xConstTickCount >= xNextTaskUnblockTime )
            {
                for( ;; )
                {
                    if( listLIST_IS_EMPTY( pxDelayedTaskList ) != pdFALSE )
                    {
                        xNextTaskUnblockTime = portMAX_DELAY;
                        break;
                    }
                    else
                    {
                        pxTCB = ( TCB_t * ) listGET_OWNER_OF_HEAD_ENTRY( pxDelayedTaskList );
                        xItemValue = listGET_LIST_ITEM_VALUE( &( pxTCB->xGenericListItem ) );

                        if( xConstTickCount < xItemValue )
                        {
                            xNextTaskUnblockTime = xItemValue;
                            break;
                        }
                        else
                        {
                            mtCOVERAGE_TEST_MARKER();
                        }

                        ( void ) uxListRemove( &( pxTCB->xGenericListItem ) );
```

```
                        if( listLIST_ITEM_CONTAINER( &( pxTCB->xEventListItem ) ) != NULL )
                        {
                            ( void ) uxListRemove( &( pxTCB->xEventListItem ) );
                        }
                        else
                        {
                            mtCOVERAGE_TEST_MARKER();
                        }

                        prvAddTaskToReadyList( pxTCB );

                        #if (  configUSE_PREEMPTION == 1 )
                        {
                            {
                                xSwitchRequired = pdTRUE;
                            }
                            else
                            {
                                mtCOVERAGE_TEST_MARKER();
                            }
                        }
                        #endif /* configUSE_PREEMPTION */
                    }
                }
            }
        }

        #if ( ( configUSE_PREEMPTION == 1 ) && ( configUSE_TIME_SLICING == 1 ) )
        {
            if( listCURRENT_LIST_LENGTH( &( pxReadyTasksLists[ pxCurrentTCB->uxPriority ] ) ) > ( UBaseType_t ) 1 )
            {
                xSwitchRequired = pdTRUE;
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        #endif /* ( ( configUSE_PREEMPTION == 1 ) && ( configUSE_TIME_SLICING == 1 ) ) */
    }
    else
    {
        ++uxPendedTicks;
    }

    #if ( configUSE_PREEMPTION == 1 )
    {
        if( xYieldPending != pdFALSE )
        {
            xSwitchRequired = pdTRUE;
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }

    return xSwitchRequired;
}
```

- Starvation 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)

(FreeRTOS 排程採用的是 multi-level queue, so?)

回目錄

中斷處理

這裡主要討論的是 ARM Cortext-M3 的中斷(Exception)，有以下幾種：

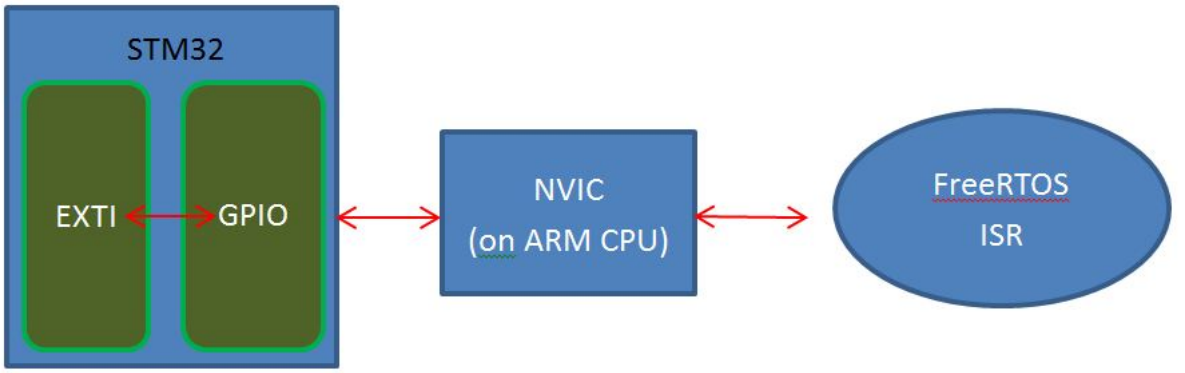
- Reset：按下 reset 後會從向量表中指定的位址，用 privileged thread mode開始執行
- NMI：簡單說就是除了 reset 外最強的 exception
- Hard Fault：做 exception handler 發生不可預期的悲劇時會觸發這個例外
- Memory management fault：memory protection 失敗時，這個例外是 MPU 觸發的，用來禁止裝置進入絕對不可進入的記憶體區域
- Bus Fault：執行指令或是 data transaction 時發生記憶體相關的錯誤，可能是 bus 的問題
- Usage fault：執行未定義的指令、不合法的未對齊存取、指令執行時出現無效的狀態、exception return 時出錯、除以 0 (需要自行設定)
- SVCall：SVC instruction 執行的 supervisor call（在OS的環境下）
- PendSV：在 OS 的環境下，通常是用於 context switch
- SysTick：當 timer 倒數到 0 時觸發，也可以用軟體中斷，OS 會把這個 exception 當成 system tick。
- Interrupt(IRQ)：週邊裝置觸發的 exception，或是軟體產生的 request

以下主要討論 interrupt 要做到處理中斷，一個作法是把 task 中用來等待 interrupt 發生的片段用 semaphore 框住，一個 ISR(interrupt service routine)被用來給(task) semaphore，以及釋放一個用來處理 handler task 來處理 ISR，這使得 ISR 的使用上更為簡化

在處理中斷上軟硬體的介接關係圖如下：

- Nested Vectored Interrupt Controller(NVIC)

NVIC 是 Cortex-M3 的一部分，他是一組用來管理各種硬體周邊發生中斷訊號的暫存器，EXIT 則是 STM32 板子上管理各種硬體發生中斷的控制器，EXIT 會通過 GPIO 把中斷訊號送給 Cortext-M3 上的 NVIC 暫存器， FreeRTOS 在透過操作 NVIC 與呼叫 interrupt service routine 來處理硬體中斷。




```
* 能夠定義 1~240 種 interrupt (FreeRTOS on ARM Cortex-M3 與 Cortex-M4 原始定義了 107 個 interrupt)
    * Cortex-M4 提供 240 個 Interrupt Priority Registers(IPR) 去記錄
* 可自定義的 interrput 優先權，從 0~255(0v爲最大優先權，最小為 255)
    * PreemptionPriority: 4 bits
    * SubPriority: 4 bits
* 處理器(硬體實作)在 interrupt 發生時會『自動』將當前狀態 stack 起來，interrupt 結束後再 unstack 回來，藉此減少 interrupt latency
* 只需要將資料放入 Software Trigger Interrupt Register(STIR)，就能夠觸發 interrupt
```

Table 4.2. NVIC register summary

Address	Name	Type	Required privilege	Reset value	Description
0xE000E100 - 0xE000E11C	NVIC_ISER0- NVIC_ISER7	RW	Privileged	0x00000000	*Interrupt Set-enable Registers*
0XE000E180- 0xE000E19C	NVIC_ICER0- NVIC_ICER7	RW	Privileged	0x00000000	*Interrupt Clear-enable Registers*
0XE000E200- 0xE000E21C	NVIC_ISPR0- NVIC_ISPR7	RW	Privileged	0x00000000	*Interrupt Set-pending Registers*
0XE000E280- 0xE000E29C	NVIC_ICPR0- NVIC_ICPR7	RW	Privileged	0x00000000	*Interrupt Clear-pending Registers*
0xE000E300- 0xE000E31C	NVIC_IABR0- NVIC_IABR7	RW	Privileged	0x00000000	*Interrupt Active Bit Registers*
0xE000E400- 0xE000E4EF	NVIC_IPR0- NVIC_IPR59	RW	Privileged	0x00000000	*Interrupt Priority Registers*
0xE000EF00	STIR	WO	Configurable [a]	0x00000000	*Software Trigger Interrupt Register*

在 Lab40

(http://wiki.csie.ncku.edu.tw/embedded/Lab40) 中，visualizer/main.c 中就有定義了 interrupt 的 priority 和取得 interrupt 的種類：

.. code-block:: c

```
#define NVIC_INTERRUPTx_PRIORITY ( ( volatile unsigned char *) 0xE000E400 )
    ...
int get_interrupt_priority(int interrupt)
{
    if (interrupt < 240)    // ARM 有 240 種 external interrupt(0~239)
            /* 根據 interrupt 設定 priority，這個常數宣告為 char*，故等同 array */
            return NVIC_INTERRUPTx_PRIORITY[interrupt];
    return -1;
}
```

- External Interrupt(EXTI)

各 interrupt handler 的排序和名稱定義(並非實作內容)放置在 startup_stm32f429_439xx.s 中，實作則放在其他地方，FreeRTOS 使用者可以定義的外部中斷通道爲 EXTI_Line0 到 EXTI_Line15，不過 EXTI_Line10~15 和 EXTI_Line5~9 被設定爲同一外部中斷通道，這表示 Line10 和 Line15 會呼叫同一個 handler，如果 Line10 和 Line15 需要有不同的任務，則要在 EXTI_Line10_15 的 handler 內做觸發來源的判斷

file:startup_stm32f429_439xx.s line:158

CORTEX_M4F_STM32_DISCOVERY/startup/startup_stm32f4xx.s

.. code-block:: c

```
  .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：

```
1. 和 GPIO 連接，作爲觸發來源 (有關 GPIO 的介紹請參照下一節)
```

```
2. 設定 EXTI

    * 設定哪條 Line
    * 模式(Interrupt, Event)
    * 被觸發的條件(Rising, Falling, Rising&falling)
    * LineCmd(ENABLE 表示設定該通道，DISABLE 表示關閉該通道)
3. 設定 NVIC
    * IRQ_Channel
    * PreemptionPriority，SubPriority
    * ChannelCmd( 同 LineCmd 的用途 )
```

在 FreeRTOS 中操作 EXTI 的實作如下：

CORTEX_M4F_STM32_DISCOVERY/main.c

.. code-block:: 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 = 0x0F;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x0F;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
```

回目錄

硬體驅動

這裡是要透過 LED 的例子介紹用 GPIO 讓硬體驅動的方法

- GPIO 簡介

GPIO 是讓開發者可藉由改變暫存器特定位置的資料內容，來控制各種周邊的硬體，或者藉由外部的輸入，來改變暫存器內容，讓硬體得知其變化後做反應

在 STM32 的開發板上可以看到許多針腳，一般 STM32F429 的板子有 7 GPIO ports，分別是 Port A, B, C, D, E, F 和 G，每個 port 有自己的暫存器序列如下，各有 32 pins(對應 32 bits)： * GPIO port mode register (GPIOx_MODER) * GPIO port output type register (GPIOx_OTYPER) * GPIO port output speed register (GPIOx_OSPEEDR) * GPIO port pull-up/pull-down register (GPIOx_PUPDR) * GPIO port input data register (GPIOx _IDR) * GPIO port outp ut data register (GPIOx_ODR) * GPIO port bit set/reset register (GPIOx _BSRR) * GPIO port bit reset register (GPIOx_BRR) * GPIO port configuration lock register (GPIOx_LCKR) * GPIO alternate function low register (GPIOx_AFRL) * GPIO alternate function high register (GPIOx_AFRH) 例如當 GPIOx 的 `x` 是 A 時，表示我們現在存取的是 GPIO Port A

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

.. code-block:: 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，並交給 GPIO_Init() 去做記憶體的設定

- 底層實作

繼續上面 LED 初始的過程為例，其中有涉及硬體設定的 function 爲 RCC_AHB1PeriphClockCmd() 和 GPIO_Init()

先討論 RCC_AHB1PeriphClockCmd()：

.. code-block:: c

```
/* 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，這讓 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，但設定項目較多，設定過程也較為複雜

GPIO_Init()：

CORTEX_M4F_STM32_DISCOVERY/Libraries/STM32F4xx_StdPeriph_Driver/src/stm32f4xx_gpio.c

.. code-block:: 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));
    }
  }
}
```

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

回目錄

效能評估

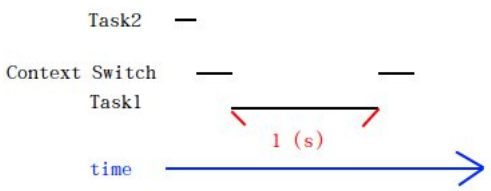
- Context switch

Context switch 是指 task A 要交出 CPU 使用權給 task B 時，OS 會將 task A 當前的狀態和暫存器內的資料存放到記憶體，再將先前 task B 的狀態從記憶體讀取至暫存器的過程

想得知 FreeRTOS 中 context switch 時間，我們設計了一個測試方法： CORTEX_M4F_STM32_DISCOVERY/main.c

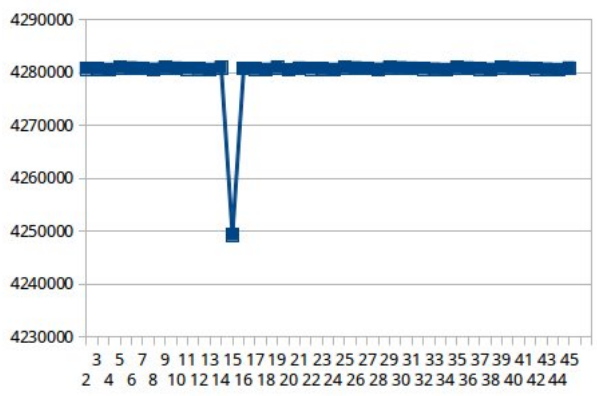
.. code-block:: c

```
uint32_t iii = 0;

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

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

void Task3( void* pvParameters )
{
        vTaskDelay( 300000 );
        while(1){
                itoa(iii, 10);
                while(1){}
        }
}

int main( void )
{
    ...
        xTaskCreate( Task1, (signed char*)"Task1", 128, NULL, tskIDLE_PRIORITY+1, NULL );
        xTaskCreate( Task2, (signed char*)"Task2", 128, NULL, tskIDLE_PRIORITY+2, NULL );
        xTaskCreate( Task3, (signed char*)"Task3", 128, NULL, tskIDLE_PRIORITY+3, NULL );
    ...
}
```



1. 首先建立 task1 和 task2，其中 task2 的優先權大於 task1 的優先權。task2 先執行，並且(task2)馬上就呼叫 vTaskDelay 使 task2 移至 block 狀態 1 秒，此時就會發生 context switch，切換成 task1 執行，這 1 秒的時間內，task1 不斷的將全域變數 iii 做 ++，直到 1 秒結束後，回到 task2 執行，再由 task2 印出 iii 的值，並把 iii 重新設為 0，此為一個週期。此動作可得到 iii 在 1 秒內可跑到多少，設 1 秒可跑至 k 值。
2. 建立 task3 並設定其優先權高於 task2，task3 會執行 vTaskDelay 300 秒，當 300 秒結束後，會中斷 task1 所執行的 iii++，再由 task3 印出 iii 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 iii 值做成圖表，其中平均 iii 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

  我們測量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

  .. code-block:: 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）

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

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

- realtime capability

回目錄

測試環境架設

安裝

```
1. 請先安裝 st-link 以及 openOCD，可參考[此頁](https://stm32f429.hackpad.com/NOTE-WbiooOfkaoR)

2. git clone https://github.com/Justinsanity/freertos-basic.git

3. git checkout porting

4. make

5. 將 stm32 f4 - discovery 接上電腦

6. make flash

7. done
```

Porting 解說

- 工具：
- stlink：用來燒錄或 GDB server 的工具
- openocd：用來執行 GDB server 與啟用 semihosting

- FreeRTOS version: 8.2.1
- Board: STM32F429 Discovery

- Porting FreeRTOS 到 STM32F429-Discovery 主要是有幾個重點

1. Utility: 來自 STM32F429I-Discovery_FW_V1.0.1(官方 driver)，用途是提供一些操作硬體周邊的函式庫(API)，例如 LCD
2. FreeRTOS：FreeRTOS 的 source code，在下載回來的包中 FreeRTOSV8.2.1/FreeRTOS/Source
3. CORTEX_M4F_SK: 在下載回來的包中 FreeRTOSV8.2.1/FreeRTOS/Demo，用途是提供平台 CORTEX_M4F_SK 上的驅動函式庫，是屬於 FreeRTOS 軟體方開發的接口

- 應用程式的開發

以 Lab39 的 freertos-basic 中 FreeRTOS 應用程式原始碼 src/ 與 include/ 為例，我們整合成 app/，並把 src/ 中的 main.c 單獨拉出來，延續 myFreeRTOS (branch: game) 的檔案架構下，應用程式 game/ 和 main.c 是放在這裡 CORTEX_M4F_SK 中，所以專案結構如下：

.. code-block::

```
.
|___CORTEX_M4F_SK
|               |__ app/
|               |__ main.c
|               |__ others C files
```

```
|___ FreeRTOS
|___ Utility
```

Hint：main.c 有樣板，在 STM32F429I-Discovery_FW_V1.0.1/Projects/Template

參考手冊：http://www.st.com/web/en/resource/technical/document/reference_manual/DM00031020.pdf

API：Utilities/STM32F429I-Discovery

回目錄

問題討論一

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

.. code-block:: c

#define configMAX_PRIORITIES (5)

Q4:為什麼要用doubly linked list

因為doubly linked list在插入新ITEM時擁有常數的時間複雜度O(1)，而Singly linked list則是O(n)

Q5：為什麼FREERTOS在FORK之後是回傳一個STRUCT 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：

.. code-block:: c

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：

.. code-block:: c

```
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：

.. code-block:: c

```
/* 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：

.. code-block:: c

```
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

回目錄

Q9：誰把New Task 接到 Ready List

.. code-block:: c

```
GDB Trace result

Breakpoint 1, xTaskGenericCreate (pxTaskCode=0x80003b1 <GameTask>, pcName=0x800ea84 "GameTask", usStackDepth=128, pvParameters=0x0, uxPr
iority=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_SIZ
E_TYPE ) ~portBYTE_ALIGNMENT_MASK  ) ); /*lint !e923 MISRA exception.  Avoiding casts between pointers and integers is not practical.  S
ize 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

.. code-block:: c

```
#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

.. code-block:: c

```
#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

.. code-block:: c

```
#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

.. code-block:: c

```
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

回目錄

Q10:arm conditional code?

conditional code用法為附加在某些條件指令之後，用來定義指令執行的代碼

Mnemonic	Meaning after ARM data processing instruction	Meaning after VFP VCMP instruction
EQ	Equal	Equal
NE	Not equal	Not equal, or unordered
CS or HS	Carry set or Unsigned higher or same	Greater than or equal, or unordered
CC or LO	Carry clear or Unsigned lower	Less than
MI	Negative	Less than
PL	Positive or zero	Greater than or equal, or unordered
VS	Overflow	Unordered (at least one NaN operand)
VC	No overflow	Not unordered
HI	Unsigned higher	Greater than, or unordered
LS	Unsigned lower or same	Less than or equal
GE	Signed greater than or equal	Greater than or equal
LT	Signed less than	Less than, or unordered
GT	Signed greater than	Greater than
LE	Signed less than or equal	Less than or equal, or unordered
AL	Always (normally omitted)	Always (normally omitted)

Q11:Thumb state bit? execution state bit?

EPSR-Execution Program Status Register內有存Thumb state bit 和 execution state bits， 其中execution state bits包含兩個重疊的區域：**

1. If-Then (IT) instruction
2. Interruptible-Continuable Instruction (ICI) field)

- about IT

IT（If - Then）指令由緊連IT的1～4條後續指令所組成（IT block）。
http://web.eecs.umich.edu/prabal/teaching/eecs373-f10/readings/ARMv7-M_ARM.pdf p.148p.149
IT instruction example：

.. code-block:: c

```
if (R4 == R5)
{
  R7 = R8 + R9;
  R7 /= 2;
}
else
{
  R7 = R10 + R11;
  R7 *= 2;
}
```

converts to http://wiki.csie.ncku.edu.tw/embedded/freertos

.. code-block:: c

```
CMP R4, R5
ITTEE EQ
ADDEQ R7, R8, R9     ; if R4 = R5, R7 = R8 + R9
ASREQ R7, R7, #1     ; if R4 = R5, R7 /= 2
ADDNE R7, R10, R11   ; if R4 != R5, R7 = R10 + R11
LSLNE R7, R7, #1     ; if R4 != R5, R7 *=2
```

- about ICI

多暫存器(multy register)讀取（LDM）和寫入（STM）是可以中斷的，ICI用來保存該執行過程中，下一個暫存器的編號。

Q12:R0~R3 , R12 , LR 這些對應到function call是哪裡?

Q13:R4~R11用在甚麼時候?

Q14:接續Q5,問FreeRTOS設計概念,回去看OS的fork部分

摘自且翻譯恐龍書八版P110～112： * 父程序（parent process）產生子程序（child process），這些新的程序被產生（fork()）後，會形成程序樹（tree of processes）。

- 一般而言，一個程序會需要一些資源（resource），子程序可以直接獲得資源或是子程序被限制在父程序的資源裡。「限制子程序在父程序資源裡」可以防止因為產生太多子程序而發生超載（overloading）。
- 典型的方法是，在呼叫fork()之後，父程序和子程序其中一個可以使用exec()來呼叫一個新的程式，取代自己的記憶體空間，這個方法的好處是，父、子程序可以跑不同的程式並且還可以做溝通（communicate）。

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

Table 2, Core registers and AAPCS usage

Q15:VFP有幾個暫存器

ARM 浮點數架構（VFP，全名Vector Floating-Point）為對浮點運算的操作提供的硬體支援。

http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0439b/Chdhfiah.html

上面表格式VFP會用到的REG

另外FPU擁有獨立的暫存器有32個（32bit)（S0~S31)

所以在PORT.C void xPortPendSVHandler(void)

.. code-block:: c

```
"    tst r14, #0x10                         \n" /* Is t
he task using the FPU context?  If so, push high vfp r
egisters. */
"    it eq                                 \n"
"    vstmdbeq r0!, {s16-s31}                \n"
```

就會把HIGH的部份堆起來（藉由STMDB達成）

Q16:!是什麼(組語)

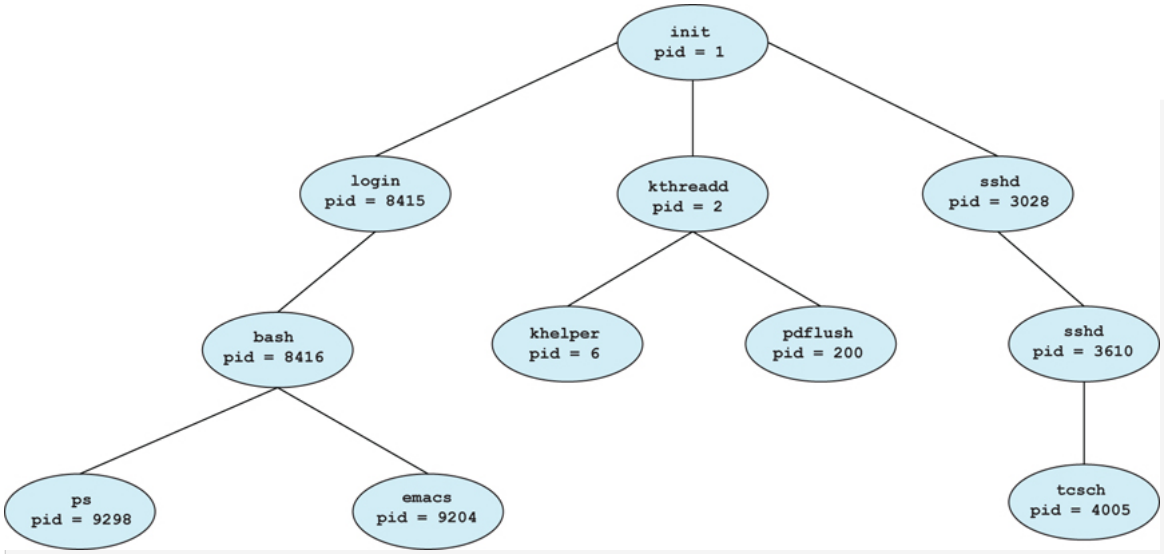
Note that the exclamation mark in ARM assembly code means that the index operation is performed before applying the real instruction.

For example, str r2, [r3, #-4]! means: store r2 value to the ptr {r3-4} and r3 = r3 -4.

指令集架構：

.. code-block:: c

```
ADD R0, R0, #1
; R0←R0+1
STR  R0, [R1]
; R0→[R1]  //將R0的值傳送到以R1的值為位址的記憶體中
ADD R0, R1, R2 LSL #3
; R0←R1 + R2*8  //R2中的運算元左移3位元
LDMIA  R0, {R1, R2, R3}
; [R0]→R1

; [R0+4]→R2

; [R0+8]→R3
比較三種：
LDR R0, [R1, #8]
; R0←[R1+8]
LDR R0, [R1], #8
; R0←[R1]

; R1←R1+8
LDR R0, [R1, #8]!
; R0←[R1+8]
; R1←R1+8
```



Address	Name	Type	Reset	Description
0xE000EF34	FPCCR	RW	0xC0000000	FP Context Control Register
0xE000EF38	FPCAR	RW	-	FP Context Address Register
0xE000EF3C	FPDSCR	RW	0x00000000	FP Default Status Control Register
0xE000EF40	MVFR0	RO	0x10110021	Media and VFP Feature Register 0, MVFR0
0xE000EF44	MVFR1	RO	0x11000011	Media and VFP Feature Register 1, MVFR1

- 關於STMxx和LDMxx指令

Data Block	Data Block	說明	對應	stack(即使用R13)	說明	stack(即使用R13)	說明
STMDA	LDMDA	每次傳送後地址減4	<===>	STMED	空遞減	LDMFA	滿遞減
STMIA	LDMIA	每次傳送後地址加4	<===>	STMEA	空遞增	LDMFD	滿遞增
STMDB	LDMDB	每次傳送前地址減4	<===>	STMFD	滿遞減	LDMEA	空遞減
STMIB	LDMIB	每次傳送前地址加4	<===>	STMFA	滿遞增	LDMED	空遞增

Q17:attribute((naked)) naked是幹嘛?

function經過compiler compile後都會在function entry和exit加入一些code，如save used registers，add return code.

但是如果不想要compiler加上這些code，就可以在function宣告時加上attribut : naked

.. code-block:: c

```
void funA(void) __attribute__ ( (naked))
{
...
 asm("ret");
}
```

Q18: vPortEnterCritical 程式碼為何能確保一次只有一個task進入critical section uxCriticalNesting++ ?

.. code-block:: c

```
funcA(){
    vPortEnterCritical();
    ...
```

```
    funcB();
    ...
    vPortExitCritical();
}
funcB(){
    vPortEnterCritical();
    ...
    vPortExitCritical();
}
```

Q19 :schedule 那邊| 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|作更正

更新在上面內容

Q20:關於 xTaskIncrementTick() 程式碼

更新在上面內容

Q21: queue實作是不是cycle？

是，參考

.. code-block:: c

```
pxQueue->pcWriteTo += pxQueue->uxItemSize;
                if( pxQueue->pcWriteTo >= pxQueue->pcTail ) /*lint !e946 MISRA exception justified as comparison of pointers is the clea
nest solution. */
                {
                        pxQueue->pcWriteTo = pxQueue->pcHead;
                }
                else
                {
                        mtCOVERAGE_TEST_MARKER();
                }
```

回目錄

問題討論二

(分兩部分純粹是為了目錄連結方便)

Q22: Lab40 (http://wiki.csie.ncku.edu.tw/embedded/Lab40) **的 visualizer/main.c 的 get_time() 裡 scale 為什麼是 microsecond ？**

問題是這樣的，為了計算 context switch 的時間，我們需要先取的系統的時間(系統啟動至今歷經的時間)，之前在 Lab40 有使用過 tick count 來取得系統時間，在 visualizer/main.c 中的 get_time() 最後 return 的 `xTaskGetTick() + ( reload - current / reload)` 是目前系統已經執行的 ticks 加上目前系統歷經的 count downs 數 (1 / x tick，讀作『x 分之一 tick』)，這段就是用 tick 來表示目前經歷的時間，要把這個 ticks 轉成 human readable time，也就是要讓 ticks 轉換成 second，一個作法就是乘上『單位量級(scale)』，這個 scale 定義在 return 之前，請參考以下片段：

<visualizer/main.c>

```
unsigned int get_time()
{
        static unsigned int const *reload = (void *) 0xE000E014;
        static unsigned int const *current = (void *) 0xE000E018;
        static const unsigned int scale = 1000000 / configTICK_RATE_HZ;
                                        /* microsecond */

        return xTaskGetTickCount() * scale +
               (*reload - *current) / (*reload / scale);
}
```

我們在意的是為什麼 scale 單位是 microsecond 呢?

先往上追 configTICK_RATE_HZ 的定義：

<visualizer/FreeRTOSConfig.h>

```
#define configCPU_CLOCK_HZ              ( ( unsigned long ) 72000000 )
#define configTICK_RATE_HZ              ( ( portTickType ) 100 )
#define configMAX_PRIORITIES            ( 5 )
```

所以 scale =

1000000/100 (tick/sec) = 10000 但是這樣我們還是不知道該給 scale 下什麼單位，因為 configTICK_RATE_HZ 只是頻率，也就是『每單位時間內有幾次tick』，官方並沒有定義這個單位時間是什麼。

但是我們注意到使用手冊中提到，FreeRTOS 在管理 task 的時候，有一個 API – vTaskDelay() 可以用來讓 Task 暫停一段時間，使用方法如下：

```
void vTaskFunction( void * pvParameters )
{
// Block for 500ms.
const portTickType xDelay = 500 / portTICK_RATE_MS;

 for( ;; )
 {
  // Simply toggle the LED every 500ms, blocking between each toggle.
  vToggleLED();
  vTaskDelay( xDelay );
 }
}
```

這段程式碼中使用了 vTaskDelay() 來暫停 task，文件說明這個函數的參數如果是整數常數，單位是 ticks，也就是說傳入整數如：vTaskDelay(500)，會暫停 500 ticks，而上述程式碼我們傳入的參數是 500/portTICK_RATE_MS，註解說這樣就可以讓 task 暫停 500 ms(10^-3 second)，所以 n / portTICK_RATE_MS 是 ms，然而此時的 n 應該不是 ticks 了(the result of our discussion)

接著我們去追portTICK_RATE_MS 的定義： <freertos-basic/freertos/libraries/FreeRTOS/portable/GCC/ARM_CM3/portmacro.h>

(in v8.2.1, portTICK_RATE_MS was renamed to portTICK_PERIOD_MS)

至此，我們推倒如下：

.. code-block:: text

```
vTaskDelay() 的 parameter 是以 tick 為單位
vTaskDelay ( 500 / portTICK_RATE_MS )
```

```
/* Architecture specifics. */
#define portSTACK_GROWTH                        ( -1 )
#define portTICK_PERIOD_MS                      ( ( TickType_t ) 1000 / configTICK_RATE_HZ )
#define portBYTE_ALIGNMENT                      8
/*-----------------------------------------------------------*/
```

```
= vTaskDelay ( 500 / 1000 / configTICK_RATE_HZ )
= vTaskDelay (50)
即 delay 50 ticks

因 500 / portTICK_RATE_MS = 500 ms =  50 ticks
另 500 ms = 500 / portTICK_RATE_MS 意思是 500/portTICK_RATE_MS 可以視為 500 ms
又 500 / portTICK_RATE_MS = 50 ticks => 500 (ms) = 50(tick) * portTICK_RATE_MS
得 x ticks * portTICK_RATE = 10 * x ms
```

```
又 scale  = 1000000 / configTICK_RATE_MS
= 1000 * (1000 / configTICK_RATE_MS)
= 1000 * portTICK_RATE_MS

所以 ticks * scale = 1000 * ticks * portTICK_RATE 可以得到 us 的值  #

其實就是說，ticks 是電腦歷經的時間，我們說 『x ticks 等於 y ms』
所以 get_time 就是要把 x tick 轉成 y ms，做法是把 x  ticks 乘上 1000/config
現在已經得到  y ms
進一步要把 x tick 的轉成 us
因 1 ms = 1000 us
=> x ticks = y ms = 1000*y*(1/1000) ms
所才要再乘上 1000
```

原來的盲點是，以為把『量級』乘 1000，這樣會使 1ms 變成 1000ms，實際上是把『數值』乘 1000，當 1 時是對應 ms 的值，1000 則是對應 us 的值

- subproblem: return 的 式子 不精準，因為 (reload - current) / reload always 0 ，(因為 int / int)，而這部分有試著去改善，但是不能 return float/double (WHY ?)

  jserv：進行除法運算，FPU 可能還沒啟用

回目錄

Q23. FreeRTOS 如何追到自己?

- 為何知道 vTaskDelete(NULL) 的 NULL 是表示 task 本身?**

在 task.c (line 400) 之中，我們找出了 vTaskDelete(NULL) 的實作是 macro:

.. code-block:: c

```
void vTaskDelete( TaskHandle_t xTaskToDelete )
{
    TCB_t *pxTCB;

    taskENTER_CRITICAL();
    {
        /* If null is passed in here then it is the calling task that is being deleted. */
        pxTCB = prvGetTCBFromHandle( xTaskToDelete );
```

- 在什麼情況下，會執行到 vTaskDelete(NULL) 呢? 遇無窮迴圈時會執行此code嗎?

因為 FreeRTOS 不准 task function 有 return，也不准執行到最後一行，因此如果不需要此 task，則應該刪除此task (參考 : http://redmilk525study.blogspot.tw/2014/09/freertos-task-management.html (http://redmilk525study.blogspot.tw/2014/09/freertos-task-management.html))

有一個可能性是，如果在 loop 中發生執行錯誤 (fail)，則需要跳出迴圈並終止(自己)執行，此時就需要使用 vTaskDelete 來刪除自己，發生錯誤的例子：

```
1. 假如今天一個 task 是要存取資料庫，但是資料庫或資料表不存在，則應該結束 task
2. 假如今天一個 client task 是要跟 server 做連線( listening 就是 loop)，卻發現 client 端沒有網路連線，則應結束 task
```

Q24. 這個 usStackDepth 如何去計算他佔用空間?

在 FreeRTOS 裡面，usStackDepth 設定 100，假設每個 stack 為 16 bits 寬，則他所佔用的記憶體空間為 200 bytes，這個值指定的是 stack 的堆疊空間 (stack size) 可以保留多少 word(4 bytes)，如果設定為 100，則系統將會分配 400 bytes 給 task，沒有簡單的方法可以計算一個任務到底需要多少空間，大多數做法都是先簡單地給予一個初估的值，然後再隨著任務運作的訊息來調整。

Q25. 為何 configMAX_PRIORITIES 要 -1 ?

這個問題是在想：如果 configMAX_PRIORITIES 設定為 10，為什麼不可以分配 10 給一個 task?

從 pxReadyTasksLists[uxTopReadyPriority] 看起來

```
/* 選出含有 ready task 的最高優先權 queue */
while( listLIST_IS_EMPTY( &( pxReadyTasksLists[ uxTopReadyPriority ] ) ) )
{
    configASSERT( uxTopReadyPriority );          //如果找不到則 assert exception
    --uxTopReadyPriority;
}
```

ready list 應該是每個 index 表示不同的 priority (換句話說，同一個 index 的 ready list 中的 tasks 應該是具有相同 priority 的)

所以，為了配置 ready list 的長度，只能從 0 ~ priority - 1，也就是說，你若要讓 ready list 長度是 10 (分別代表著 priority 0 - 9)，則你雖然要把 configMAX_PRIORITY 設定為 10，分配權限時最高卻只能到 configMAX_PRIORITY，其實我只是簡單的理解上是array 如果要有1$_{10個優先權定義，事實上寫程式時是定義0}$9 而已，的確是由0開始數沒錯。

回目錄

Q26.真正放至 ready list 的是什麼? 是放入 handle?

<freertos/libraries/FreeRTOS/include/task.h#L345>

.. code-block:: c

```
#define xTaskCreate( pvTaskCode, pcName, usStackDepth, pvParameters, uxPriority, pxCreatedTask )
        xTaskGenericCreate( ( pvTaskCode ), ( pcName ), ( usStackDepth ), ( pvParameters ), ( uxPriority ),
        ( pxCreatedTask ), ( NULL ), ( NULL ) )
```

(這三行是同一行，未方便閱讀而斷行)

在這裡把 xTaskGenericCreate() 包裝成 xTaskCreate()，所以 main 中呼叫 vTaskCrate() 實際上是呼叫 xTaskGenericCreate()

然後是呼叫 <freertos/libraries/FreeRTOS/tasks.c#L551> 的 xTaskGenericCreate，在 xTaskGenericCreate 中我們找到把 task 加入 readyList 的地方(#704)：

.. code-block:: c prvAddTaskToReadyList(pxNewTCB);

所以實際上是把該 task 的 TCB 加入 readyList

那要怎麼辨識 task 呢？ ((`configASSERT( pxTaskCode );` 並不是註冊 taskCode，`configASSERT()` 只是用來測試的)) 在 line 621 (xTaskGenericCreate)：

.. code-block:: c

```
/* Setup the newly allocated TCB with the initial state of the task. */
prvInitialiseTCBVariables( pxNewTCB, pcName, uxPriority, xRegions, usStackDepth );
```

這裡看到帶有 pcName 卻沒有 taskCode，我們已知 pcName 只用做開發識別，不用做系統管理，所以追到下面

.. code-block:: c

```
#if( portUSING_MPU_WRAPPERS == 1 )
{
    pxNewTCB->pxTopOfStack = pxPortInitialiseStack( pxTopOfStack, pxTaskCode, pvParameters, xRunPrivileged );
}
#else /* portUSING_MPU_WRAPPERS */
{
    pxNewTCB->pxTopOfStack = pxPortInitialiseStack( pxTopOfStack, pxTaskCode, pvParameters );
}
```

應該是因為在 TCB 中記錄了 stack 的起始位址，這裡有 pxTaskCode，不是用 handle！(不過 handle 是否也是指向 pxTopOfStack???)

- handle: (台：控制代碼、中：句柄)實際的意義是一個用來存取該 task 的『指標』，他是提供外部系統或程式來參照目前 task 用的

舉個例子：刪除 task

.. code-block:: c

```
vTaskCreate(..., taskA, ...)
```

當我們要呼叫 vTaskDelete 來刪除 taskA 時，為了讓 vTaskDelete 存取到 taskA 的記憶體位址，我們要傳入一個指向 taskA 的指標到 vTaskDelete 中，這個指標就是 handle

建立 handle 的方法如下：

.. code-block:: c

```
xTaskHandle handleTaskA;    // 建立一個用來指向 taskA 的 handle 指標變數
vTaskCreate(.., taskA, ..., handleTaskA);   // 傳入這個指標變數，讓 vTaskCreate 可以記錄他

/* 因為是 pass by address，handleTaskA 大概在 vTaskCreate() 中只是被修改值，所以離開 vTaskCreate 後就可以直接存取到 handleTaskA 的內容 */
vTaskDelete(handleTaskA);   // 傳入 handle 以供 vTaskDelete 參考
```

Q27. FreeRTOS 在哪裡去更新 pcCurrentTCB 這個 pointer ? 指向最新正在運行的 task?

這個問題是要探討如何用 scheduler，still working…to be continue.

Q28. 為何不使用 linked-list 去紀錄 priority value?

因為 linked-list 要去排序的話，有 N 個，有可能會有 n^2 或者是 nlogn 的時間複雜度，且 array 可以做 random access，這對以 priority 作為 index 的 ready list 來說，會比 linked-list 更直接的存取到指定的 priority

使用 uxTopReadyPrioirty 是因為用空間換時間，搜尋會只有常數時間複雜度

Q29. xListEnd在哪?

xLIST 中的 xListEnd 是 xLIST（linked-list？）的尾巴

Q30. FreeRTOS 的event 定義是什麼? 是同步機制還是非同步?

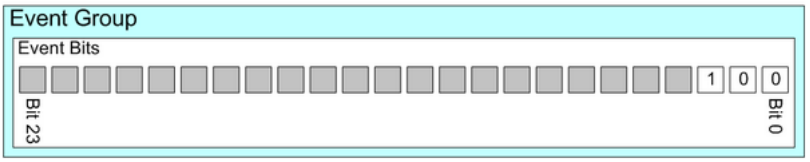
```
-Event bits are used to indicate if an event has occurred or not. Event bits are often referred to as event flags.
-來源：[http://www.freertos.org/FreeRTOS-Event-Groups.html](http://www.freertos.org/FreeRTOS-Event-Groups.html)
```

group 中有很多 bit，分別代表 task 正在等待的一個 event，並不一定所有的 bit 都會用到，看 task 需要等待多少 task event 例如：

.. code-block:: text

```
A message has been received and is ready for processing
The application has queued a message that is ready to be sent to a network
It is time to send a heartbeat message onto a network
```

TCB 結構中還有兩個 xListItem：xGenericListItem 和 xEventListItem



An event group containing 24-event bits, only three of which are in use

.. code-block:: c

```
xListItem    xGenericListItem;               /* 用來記錄 task 的 TCB 在 FreeRTOS ready 和 blocked queue 的位置 */
xListItem    xEventListItem;                 /* 用來記錄 task 的 TCB 在 FreeRTOS event queue 的位置 */
```

不論 xGenericListItem 或 xEventListItem，都是 xListItem_t 型態，這個結構定義在 list.h：

.. code-block:: c

```
struct xLIST_ITEM
{
    listFIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE
    configLIST_VOLATILE TickType_t xItemValue;
    struct xLIST_ITEM * configLIST_VOLATILE pxNext;
    struct xLIST_ITEM * configLIST_VOLATILE pxPrevious;
    void * pvOwner;
    void * configLIST_VOLATILE pvContainer;
    listSECOND_LIST_ITEM_INTEGRITY_CHECK_VALUE
};
typedef struct xLIST_ITEM ListItem_t;
```

回目錄

Q31. 如何知道 task現在處於哪個狀態? 對應狀態圖的程式碼在哪裡?

Task 有四種狀態 blocked (‘B’), ready (‘R’), deleted (‘D’) or suspended (‘S’)

vTaskList(buf) 將task的所有資訊寫進buf內，之後再把buf的內容print出來

.. code-block:: c

```
fio_printf(1, "\n\rName          State   Priority  Stack  Num\n\r");
fio_printf(1, "*********************************************\n\r");
fio_printf(1, "%s\r\n", buf + 2);
```

就可以知道task處於哪個狀態

.. code-block:: text

```
vTaskList() calls uxTaskGetSystemState(), then formats the raw data generated by uxTaskGetSystemState() into a human readable
(ASCII) table that shows the state of each task, including the task's stack high water mark (the smaller the high water mark
number the closer the task has come to overflowing its stack). Click here to see an example of the output generated.
In the ASCII table the following letters are used to denote the state of a task:
    'B' - Blocked
    'R' - Ready
    'D' - Deleted (waiting clean up)
    'S' - Suspended, or Blocked without a timeout
vTaskList() is a utility function provided for convenience only. It is not considered part of the kernel.
SeevTaskGetRunTimeStats() for a utility function that generates a similar table of run time task utilisation information.

Parameters:

pcWriteBuffer  A buffer into which the above mentioned details will be written, in ASCII form. This buffer is assumed to be
               large enough to contain the generated report. Approximately 40 bytes per task should be sufficient.
```

Q32. 畫一下 pxStack 以及 pxTopOfStack 的記憶體位置圖(用gdb去trace)

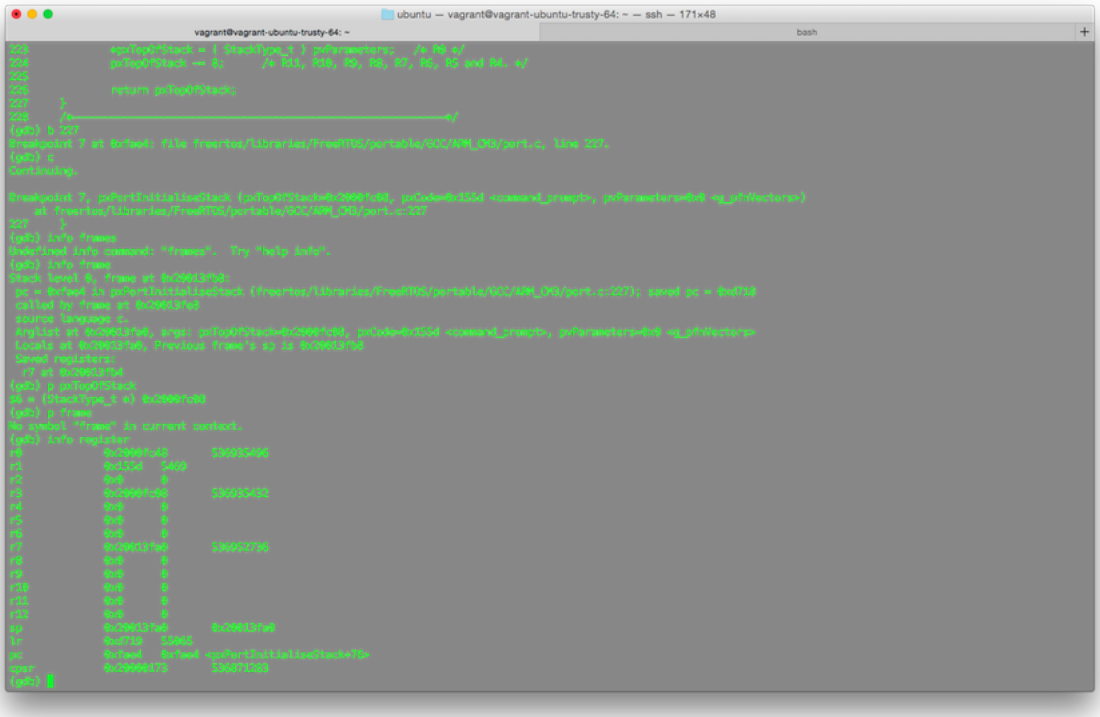
2015 春季班 v8.2.1 版本

<freertos/libraries/FreeRTOS/portable/GCC/ARM_CM3/port.c>

.. code-block:: c

```
/*
 * See header file for description.
 */
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. */
    pxTopOfStack--; /* Offset added to account for the way the MCU uses the stack on entry/exit of interrupts. */
    *pxTopOfStack = portINITIAL_XPSR;        /* xPSR */
    pxTopOfStack--;
    *pxTopOfStack = ( StackType_t ) pxCode;        /* PC */
    pxTopOfStack--;
    *pxTopOfStack = ( StackType_t ) portTASK_RETURN_ADDRESS;        /* LR */
    pxTopOfStack -= 5;        /* R12, R3, R2 and R1. */
    *pxTopOfStack = ( StackType_t ) pvParameters;        /* R0 */
    pxTopOfStack -= 8;        /* R11, R10, R9, R8, R7, R6, R5 and R4. */

    return pxTopOfStack;
}
```

參考了 Lab33 (http://wiki.csie.ncku.edu.tw/embedded/Lab33)使用 “(gdb) target: remote” + “info registers” 來觀察暫存器內容，而在 freertos-basic/tool/gdbscript 中就已經有 target: remote 3333 了，freertos/mk/qemu.mk 呼叫 qemu 並且直接進入 gdb，因此可以使用 info registers 來觀察，下圖是 pxPortStackInitialise() 執行完畢以後的暫存器狀態圖



址，因此他也是存取 TCB

回目錄

參考資料

- The Architecture of Open Source Applications: FreeRTOS (http://www.aosabook.org/en/freertos.html)
- 簡體中文翻譯 (http://www.ituring.com.cn/article/4063)
- Study of an operating system: FreeRTOS
- FreeRTOS 即時核心實用指南

Q33. 在 FreeRTOS 裡面，其 pid 在哪裡? 如何知道 task 的 id 資訊?

FreeRTOS 建立新 task 的方式與 Linux 不同，前者是用 vTaskCreate()，如果有回傳，只會回傳 pdPass 或 error 來表示 task 建立成功與否，如果要讓程式辨識這個 task，在 create 時就要給予 vTaskCreate 一個用來記錄該 task TCB 起始位址的 handle 指標，因此 FreeRTOS 沒有 PID

FreeRTOS 透過一個 pxCurrentTCB 的指標來存取目前指向的 task 基本資訊，也就是說如果要知道 task 的資訊，都是去存取 TCB，2007 年的資料顯示，好像可以透過存取 pxCurrentTCB->uxTCBNumber 來取得 task 的編號，但是目前的 TCB 已經沒有這個項目了

至於考慮使用 handle 來取得 task id 也是不對的，因為 handle 指向的是 TCB 的位