TP3

Utilisation des Queues

(plateforme basée sur un pic24F)

1. Objectifs :

Apprendre à utiliser des queues (files d'attente FIFO) pour un besoin de communication entre tâches et partage de ressources:

- Création de queue adaptée aux besoins
- lecture à partir de ou écriture dans une queue : fonctions bloquantes ou non bloquantes
- Combinaison des priorités de tâches avec l'utilisation d'une queue pour garantir le contrôle de flux.
- Utilisation d'une file pour le partage de la ressource UART entre deux tâches.

2. Mode opératoire :

2.1. Installation de la librairie de périphérique pour pic24F et dsPic :

Dans ces TP on utilisera le périphérique UART du PIC24F, ainsi il faut installer la librairie de périphérique pour PIC24F fourni par Microchip, en utilisant ce lien pour le système Windows :
http://ww1.microchip.com/downloads/en//softwarelibrary/pic24%20mcu%20dspic%20peripheral%20lib/peripheral-libraries-for-pic24-and-dspic-v2.00-windows-installer.exe

Pour les autres systèmes consultez la page :
https://www.microchip.com/SWLibraryWeb/product.aspx?product=PIC24%20MCU%20dsPIC%20Peripheral%20Lib

2.2. Utilisation d'une copie du projet « template » créé en TP1

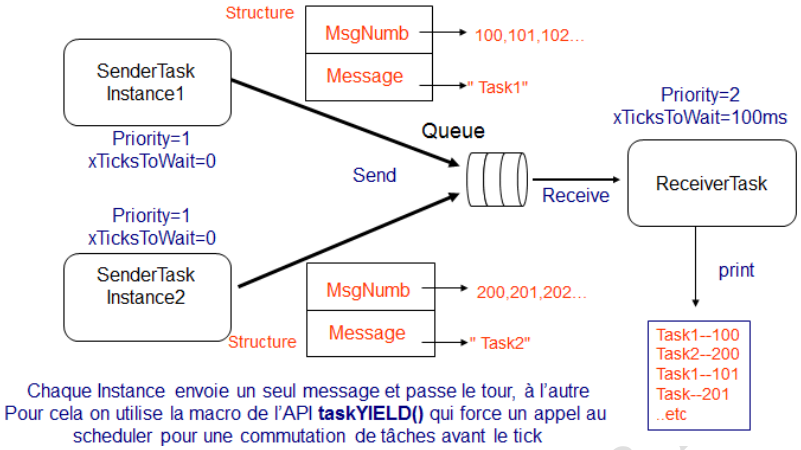
Dans Mplab, ouvrir le projet template crée en TP1 et faites une copie et **renommer** le en TP3 par exemple. **Dans la suite de ces TP, on travaillera avec cette copie.**

NB : éviter des noms de projets avec des espaces et/ou des caractères spéciaux comme les caractères avec accents ; éviter aussi de placer votre projet dans un dossier avec un chemin d'accès trop long et/ou contenant des espaces et/ou caractères accentués. Ceci peut vous causer des problèmes lors de compilation/débuggage.

2.3. Utilisation d'une queue exploitée par plusieurs émetteurs vers un seul récepteur

2.3.1. Utilisation du simulateur de MPLABX pour la vérification des résultats

1. L'objectif est de faire communiquer deux instances d'une même tâches 'Sender' avec une tâche 'Receiver', via une file comme le montre la figure suivante :



2. Le projet TP3.X utilisera des Queues, il faut ajouter la directive #include "queue.h "

```
#include "xc.h"
#include "FreeRTOS.h"
#include "task.h"
#include "list.h"
#include "queue.h"
#include <stdio.h>
#include <string.h>
```

pour printf et strcpy

3. Définissez le type des items qui seront stockés dans la file [1], les items échangés entre tâches[2], les paramètres des deux premières tâches[3] et le handle de la queue [4].

```
20  //functions declaration
21  void vSenderTask(void *pv1);
22  void vReceiverTask(void *pv2);
23  void vApplicationIdleHook(void);
24  typedef struct {
25      unsigned short usMsgNum;
26      unsigned char Message[10];
27  } ItemStruct;
28
29  //Task's message
30  ItemStruct Item1, Item2;
31  //Tasks' Parameters pointers
32  void *p1, *p2;
33  //Handle of Queue
34  xQueueHandle xQueue;
35
36  int main(int argc, char** argv) {
```

1

2

3

4

Ce Handle doit être déclaré comme variable globale car il sera utilisé par trois tâches

4. Le projet doit contenir 3 tâches, deux basées sur la même fonction tâche vSenderTask( ) [1] et la troisième basée sur la fonction tâche vReceiverTask( ) [2]

```
63  void vSenderTask(void *pv) {
64      portBASE_TYPE xStatus;
65
66      while (1) { // une tâche est une boucle infinie
67          xStatus = xQueueSendToBack(xQueue, pv, 0); //ne se bloque jamais
68          if (xStatus != pdPASS) //Queue is full
69          {
70              printf("queue full\n");
71          }
72          else {
73              ((ItemStruct *) pv)->usMsgNum++; //increments Message Number
74          }
75          taskYIELD(); //forces yielding
76      }
77  }
```

1

Demande une commutation de tâches pour passer la main à une autre tâche de même niveau de priorité

```
79  void vReceiverTask(void *pv) {
80
81      portBASE_TYPE xStatus;
82      ItemStruct AnItem; //received item
83      portTickType xTicksToWait = pdMS_TO_TICKS(100);
84      for (;;) {
85          if (uxQueueMessagesWaiting(xQueue) != 0) printf("should be empty\r\n");
86
87          xStatus = xQueueReceive(xQueue, &AnItem, xTicksToWait);
88          if (xStatus == pdPASS) {
89              printf("From :%s Msg Numb: %d\r\n", AnItem.Message, AnItem.usMsgNum);
90
91          } else printf("Cannot receive frome Queue!!\r\n");
92
93      }
94
95  }
```

2

convertit le nombre le nombre de ms en nombre de ticks

5. La fonction main sera comme suit :

```
int main(int argc, char** argv) {
    //Queue creation
    xQueue = xQueueCreate(3, sizeof ( ItemStruct));
    //Tasks' Items Initialization
    ItemStruct Item1, Item2;
    Item1.usMsgNum = 100;
    strcpy((char *)Item1.Message, (char *)"Task1");//string copy
    p1 = &Item1;
    Item2.usMsgNum = 200;
    strcpy((char *) Item2.Message, (char *) "Task2");
    p2 = &Item2;
```

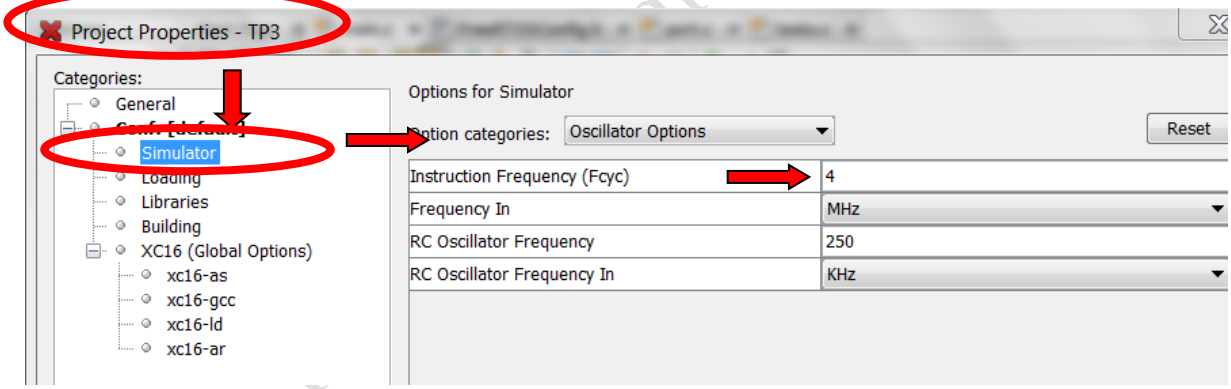
```
    // Sender Tasks  Creation
    xTaskCreate(vSenderTask, "Sd1", 150, p1, 1, &Htask1);
    xTaskCreate(vSenderTask, "Sd2", 150, p2, 1, &Htask2);
    // Receiver Task Creation
    xTaskCreate(vReceiverTask, "Rcv", 150, NULL, 2, &Htask3);
    //Run the scheduler
    vTaskStartScheduler();
    return (0);
}
```

NB : la fonction main() utilise la fonction de copy de chaine de caractères strcpy( ) il faut alors ajouter #include " string.h".

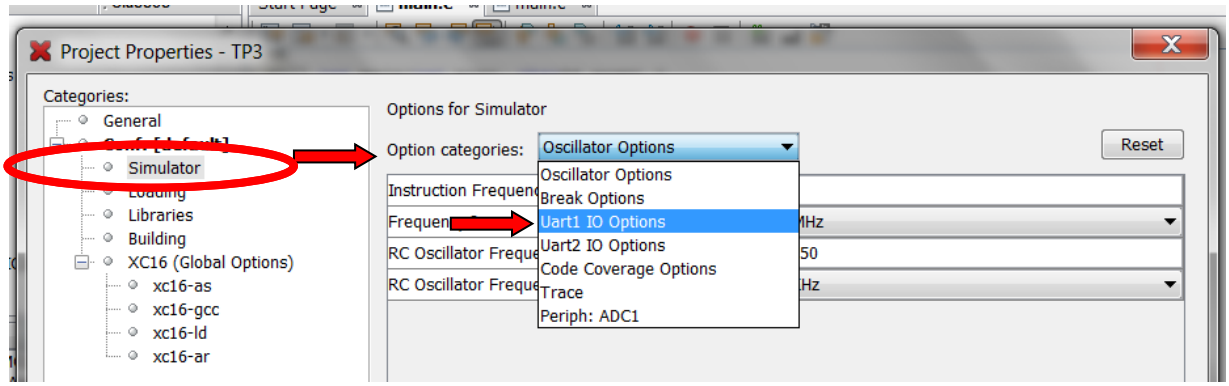
6. Adoptez le réglage pour la fréquence des ticks d'horloge suivant dans freertosconfig.h

```
#define configTICK_RATE_HZ                    ( ( TickType_t ) 1000 )
#define configCPU_CLOCK_HZ                    ( ( unsigned long ) 4000000 )   /* Fos
```

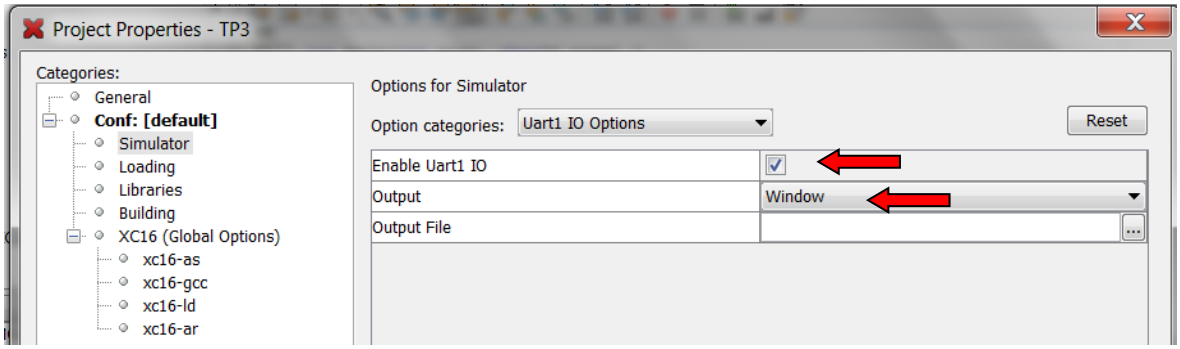
7. **Le projet utilise la fonction C standard : printf, qui permet en principe de faire un affichage vers une sortie standard qui est un écran sur un PC; Pour une carte à base de microcontrôleur la sortie standard peut être redirigée vers une interface série UART (Universel Asynchronous Receiver Transmitter)** . Le simulateur de MPLAB supporte ce type d'interface et permet de voir le résultat de cette fonction sur écran : pour cela on va rediriger le port UART1 simulé de MPLABX ; Modifier les propriétés du projet comme suit :



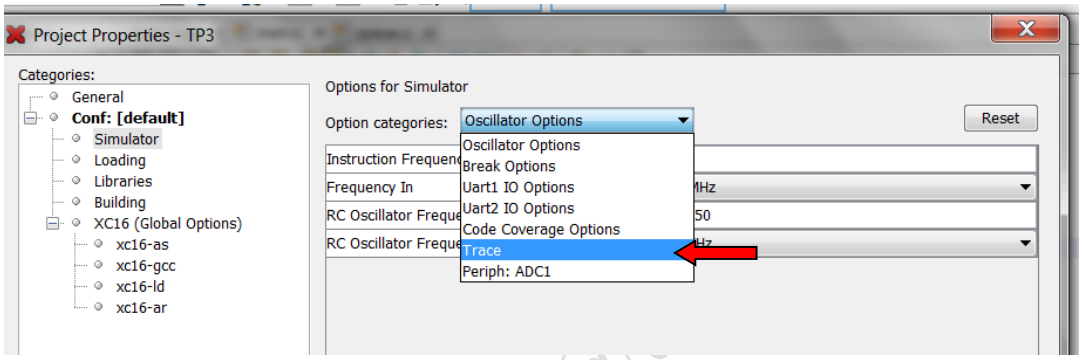
Le simulateur de MPLABX supporte la simulation de ce port et permet de rediriger les fonctions d'entrée standard depuis un fichier ou sortie standard vers fichier ou fenêtre. Pour cela faites la configuration suivante :



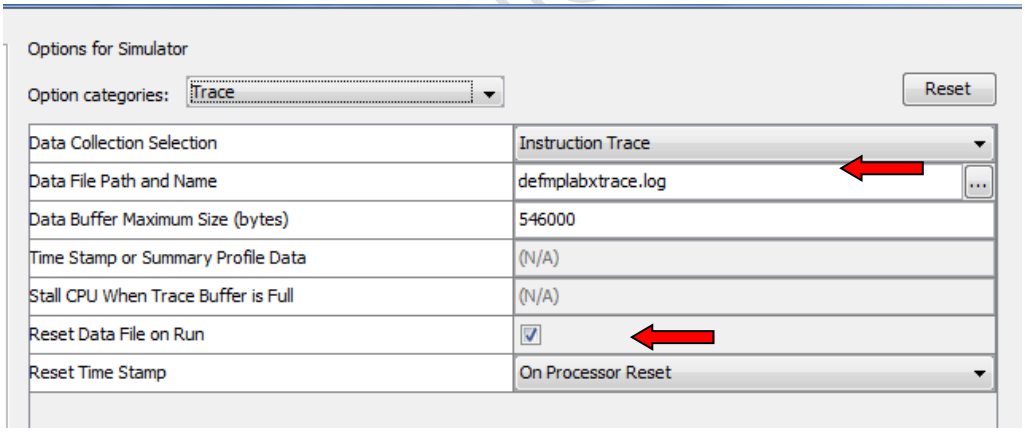
puis activez le port UART1 , avec sortie vers écran (fenêtre)



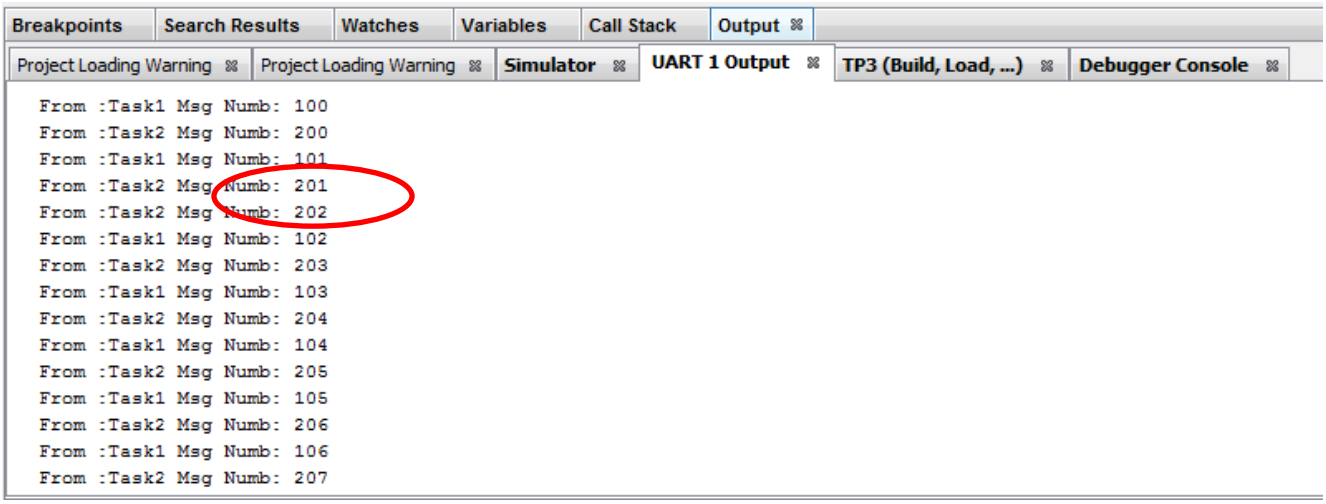
Activez la trace



Puis



8. Validez les modifications et fermez la fenêtre des propriétés de TP3.
9. Construisez le projet et le debuggez en pas à pas non détaillé en mettant deux points d'arrêts l'un dans la fonction vSenderTask et l'autre dans la fonction vReceiverTask ; Surveillez aussi les contenu des variables *pv , pv->usMsgNum pour savoir quel sender est en cours d'envoi. Le résultat obtenu au niveau de la fenêtre UART1 Output, montre une certaine alternance (mais pas toujours garantie !) entre les deux senders :



D'après ce résultat la file qui ne contient que 3 emplacements ne se remplie jamais ; elle contient au plus un item. Les items sont reçus (presque) en alternance. En effet, dès qu'une tâche Sender écrit dans la file, la tâche Receiver est débloquée et puisqu'elle est plus prioritaire elle passe à l'état Running pour lire et afficher l'item puis tente de lire un autre item et se bloque à nouveau en attente d'un autre item.

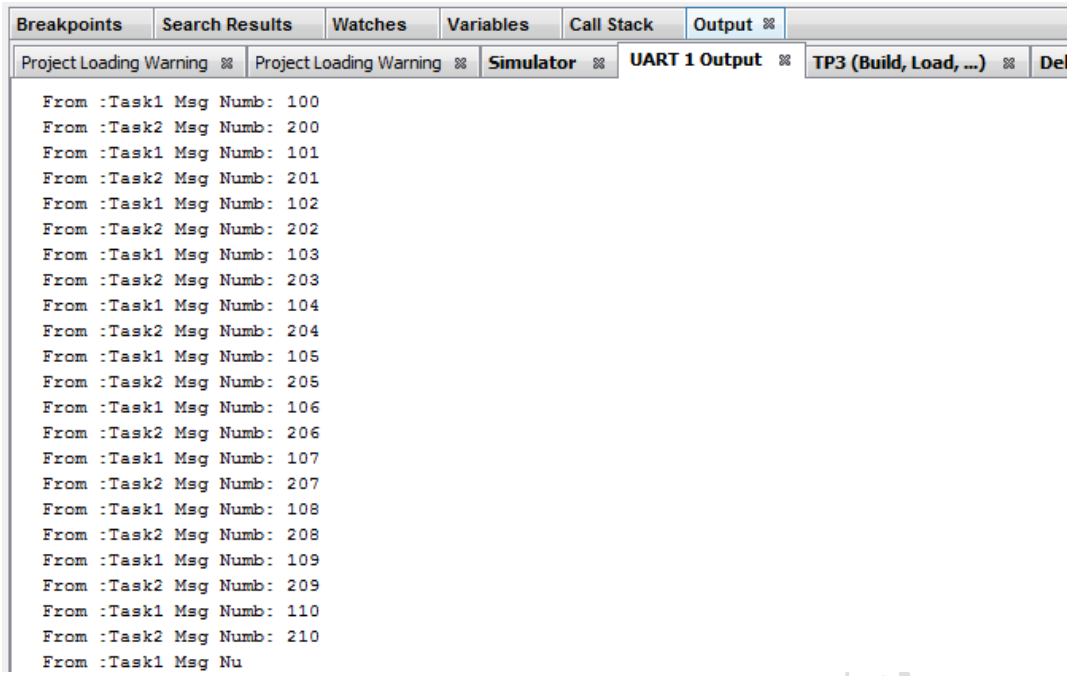
NB : à chaque essai depuis le début, il vaut mieux effacer le contenu de la fenêtre UART1 Output par un clic droit suivi de l'option clear.

10. L'appel à la fonction de commutation forcée de tâches taskYIELD(), est ici inutile : commentez cette instruction et refaites le test, le résultat est le même ; en effet ici chaque tâche sender est préemptée par la tâche Receiver, dès qu'elle envoie un item : elle est renvoyée dans la file de l'état Ready après l'autre Sender. Le blocage de Receiver (à cause de la file vide) donne la chance à l'autre tâche sender de s'exécuter et d'envoyer un item elle aussi : ce ci garantit l'alternance des items reçus (un de chaque tâche).

11. Ajoutez au fichier freertosconfig.h la directive :

    #define configUSE_TIME_SLICING 0

    Reconstruisez votre projet, et faites un test ; cette fois l'alternance est toujours garantie, comme l'indique la capture d'écran suivante :

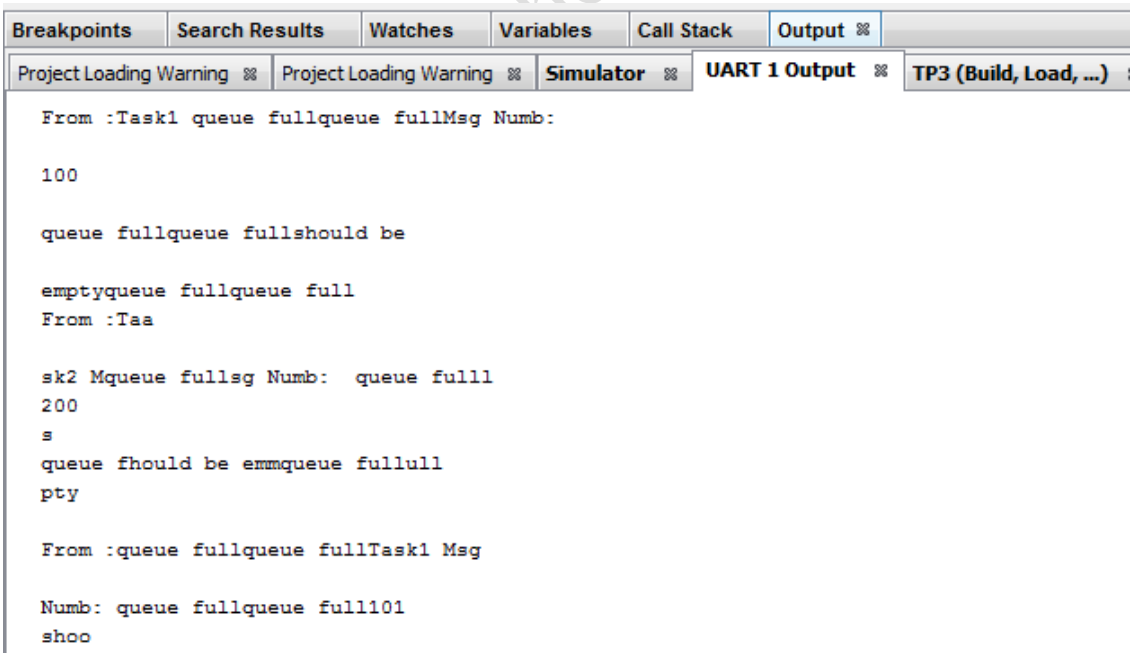


12. Commentez la directive ajoutée à freertosconfig.h:

    // #define configUSE_TIME_SLICING 0

    Donnez maintenant à la tâche Receiver la priorité 1 comme les tâches sender et utilisez l'appel à la fonction taskYIELD( ).

    Le résultat peut ressembler à ce qui suit :



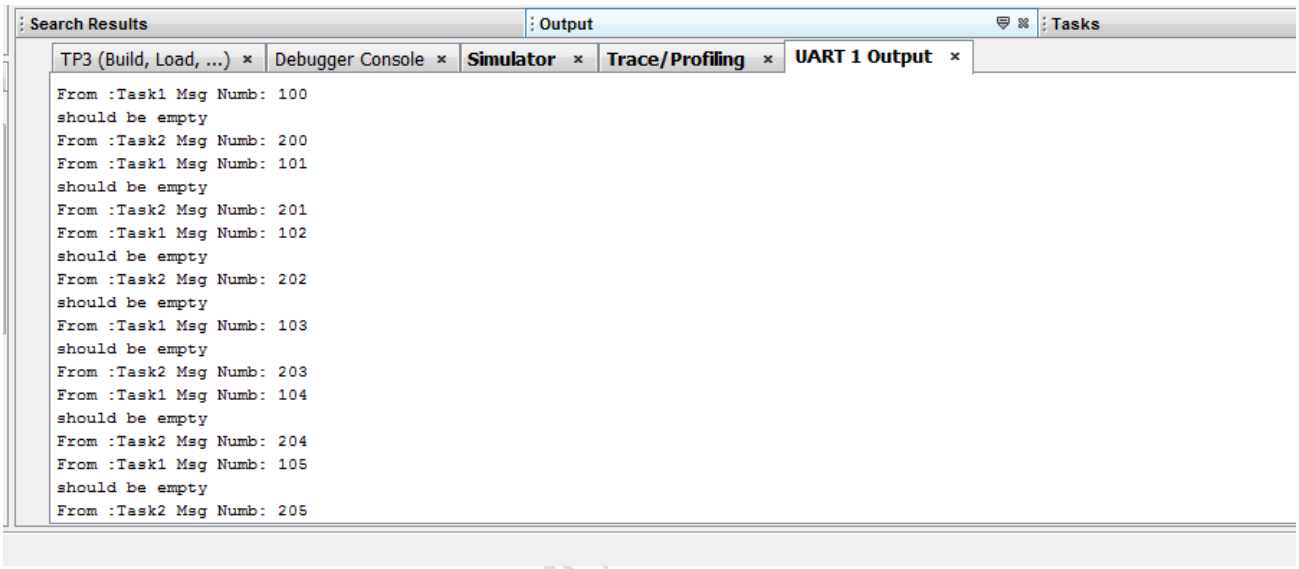
Cette fois ci, la tâche « Receiver » aura une chance sur 3 de s'executer, ce qui ne garantie pas que la file ne sera pas remplie d'au plus un item comme précédemment : cela se justifie par l'apparition du message « Queue should be empty » ; la file même devient par moment complétement remplie et les « Sender » échouent lors des tentatives d'envois d'items : ce qui se justifie par l'apparition du message d'erreur « Queue full » par moment. D'autre part l'UART est une interface série ayant un taux de transfert lent (envoi bit par bit des caractères à un

débit relativement faible) ce qui fait que l'affichage d'un message peut nécessiter plus d'un quantum et sera donc préempté par le scheduler avant son affichage complet ; cela justifie par l'entrelacement des messages issus de différentes tâches !

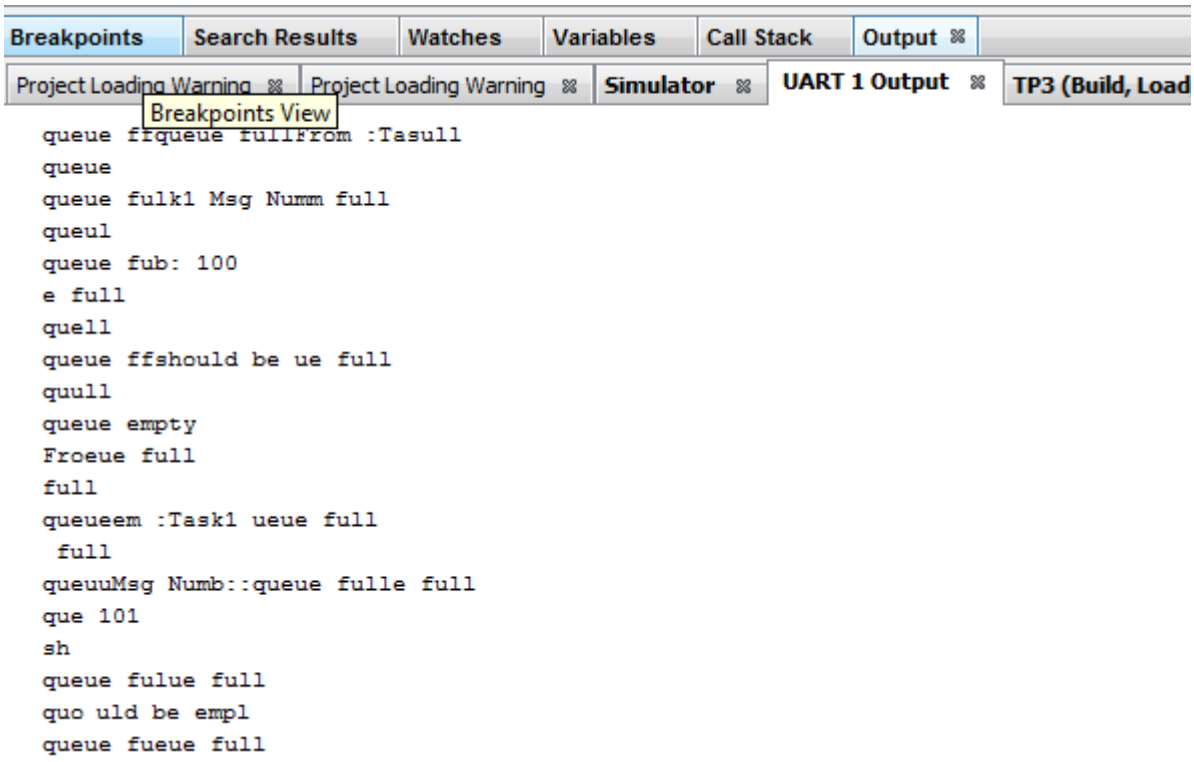
taskYields() force chaque sender à céder le processeur après chaque tentative d'envoi (avec ou sans succès) ; ce qui donne (presque) une alternance des items reçus de la part des deux senders.

Donnez à configTICK_RATE_HZ, dans freertosconfig.h, la valeur 100 au lieu de 1000, pour augmenter le quantum de 1ms à 10ms. On peut alors remarquer la disparition des entrelacements de messages.

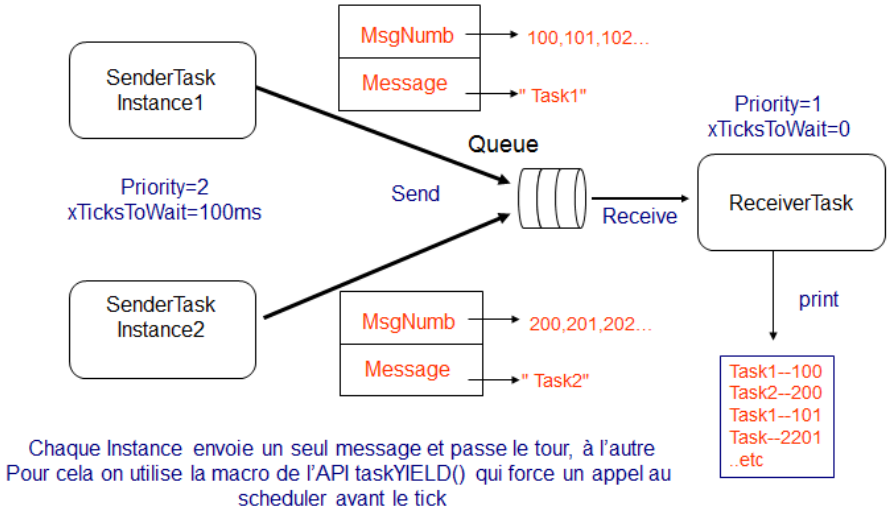
Cette situation n'est pas très déterminsite comme dans la situation précedente, car elle dépend de la largeur des messages à afficher, de la valeur du quantum ;



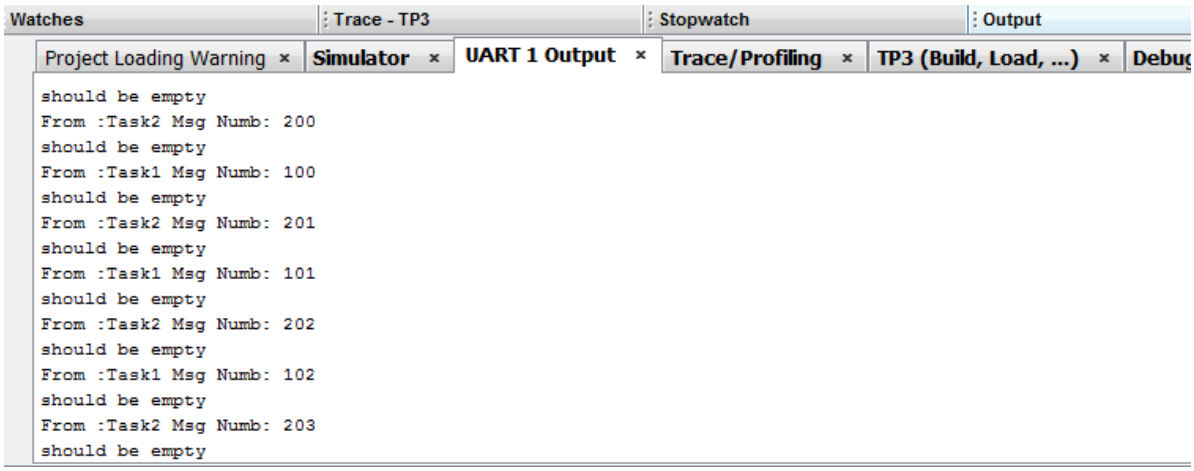
13. Remettez configTICK_RATE_HZ à 1000. Donnez maintenant à la tâche « Receiver » la priorité 1 et commentez l'appel à taskYIELD( ) . Le résultat devient complétement aléatoire, la file est pleine, la plus part de temps, avec beaucoup d'échecs d'envois d'item, et on constate que deux ou plusieurs items issus de la même tâches peuvent être reçus susccessivement. Dans la question précédente, taskYIELD( ) donnait la chance au receiver de s'exécuter après deux tentatives d'envois, ce qui n'est plus le cas ici puisque un sender peut envoyer plusieurs items avant que son quantum ne se termine et que l'autre sender ou au receiver prend le processeur. Là encore la situation n'est pas déterministe.



14. On considère la situation suivante :

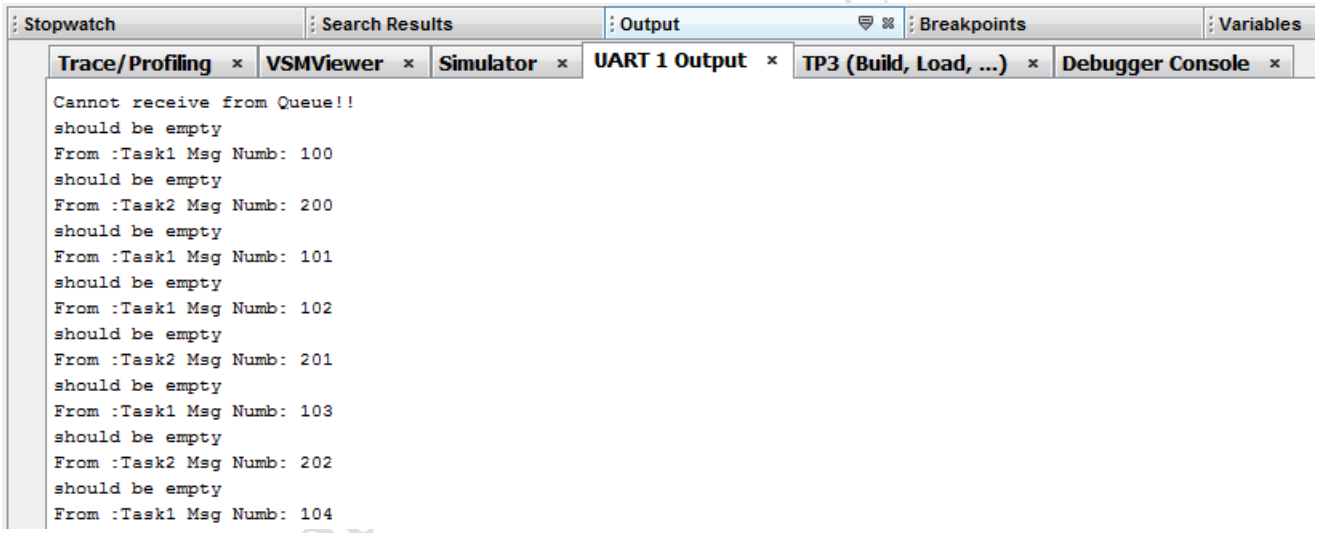


15. Adaptez le code précédent à cette situation, et maintenez l'appel à la fonction taskYIELD( ). Testez le projet, le résultat de cette situation ressemblera à ce qui suit, analysez ce résultat et essayez de le justifier



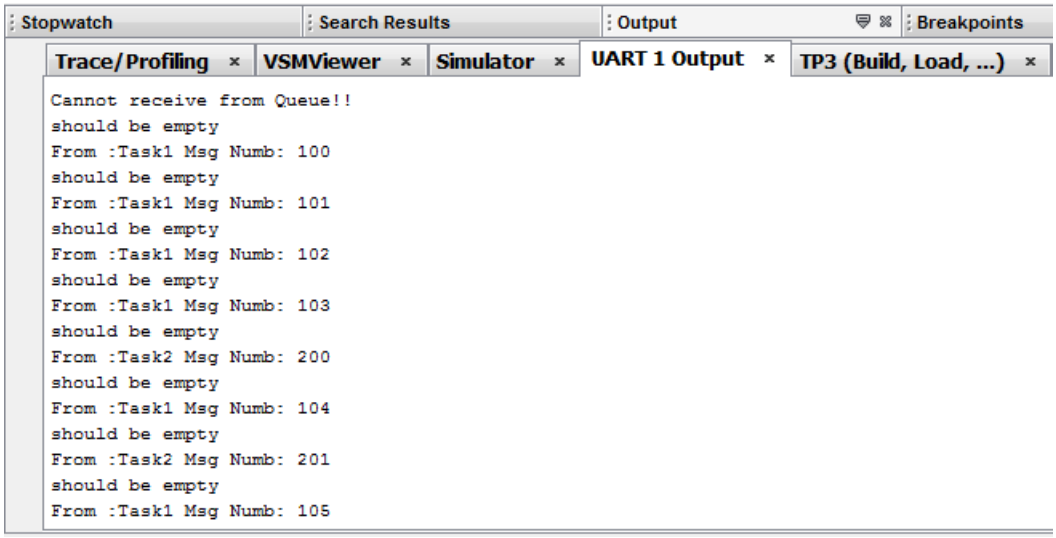
Ici la file est presque toujours pleine, comme en témoigne le message « should be empty » affiché par le receiver. Ceci justifie pourquoi le temps d'attente du receiver est réglé à 0.

16. Donnez maintenant à la tâche Receiver la priorité 2 comme les tâches Senders, le résultat serait comme le suivant :

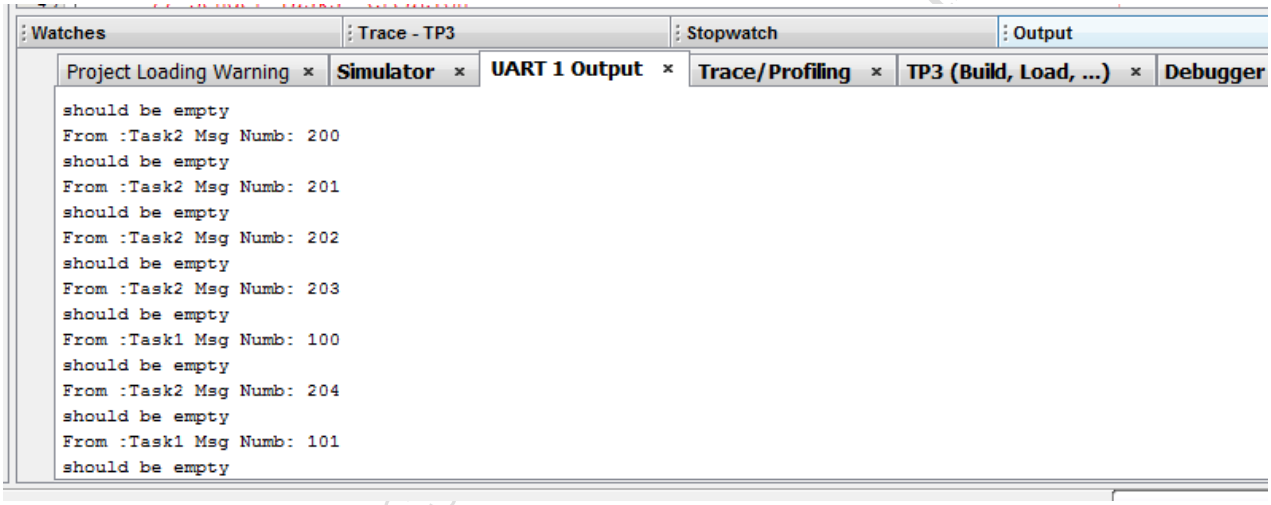


Analysez et commentez ce résultat.

17. Toujours avec une priorité de la tâche Receiver à 2 et commentez taskYIELD( ), le résultat devrait ressemblez à ce qui suit ; analysez et commentez ce résultat :



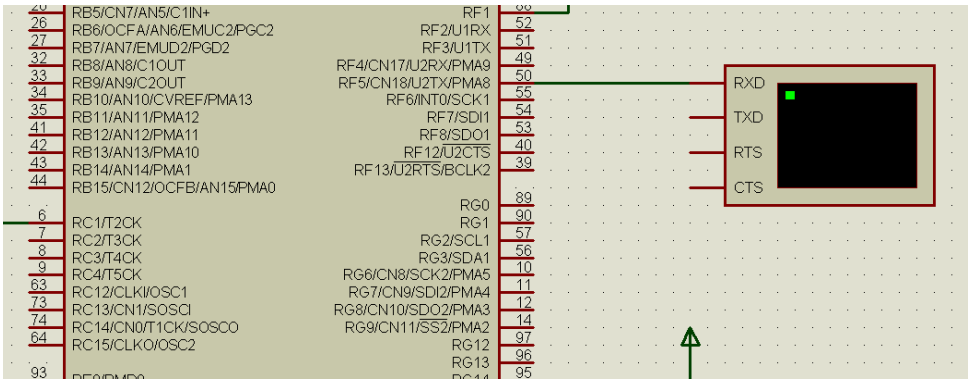
18. Remettez la priorité de la tâche Receiver à 1 et enlevez l'appel à taskYIELD( ) : analysez et commentez le résultat trouvé qui doit en principe ressembler à ce qui suit :



2.3.2.Partage du port série UART2 entre deux tâches pour un affichage vers un terminal :

Le projet sera modifié pour exploiter UART2 du PIC24F pour envoyer les messages de deux Sender vers un terminal Virtuel sur ISIS.

19. Commencez par modifier les propriétés du projet pour utiliser VSM Viewer comme outils de débuggage et concevez un design sur ISIS avec un Terminal Virtuel connecté au PIC:



20. Utilisez la librairie des périphériques fournie avec le compilateur XC16 pour configurer et exploiter UART2 ; pour cela :

20.1. Ajoutez, dans le fichier main.c, la directive permettant d'exploiter les macro-masques utilisés par cette librairie [1], et les deux directives d'inclusions de uart.h [2].

```
#define USE_AND_OR
#include "xc.h"
#include "FreeRTOS.h"
#include "task.h"
#include "list.h"
#include "queue.h"
#include <stdio.h>
#include <string.h>
#include <uart.h>
```

1

2

20.2. L'initialisation du port UART2 se fera au début de la fonction main()

```
int main(int argc, char** argv) {
    //Queue creation
    xQueue = xQueueCreate(3, sizeof ( ItemStruct));
    //Tasks' Items Initialization
    ItemStruct Item1, Item2;
    Item1.usMsgNum = 110;
    strcpy(Item1.Message, "Task1:");
    p1 = &Item1;
    Item2.usMsgNum = 220;
    strcpy(Item2.Message, "Task2:");
    p2 = &Item2;

    /* UART2 Configuration and activation */
    CloseUART2(); //first close UART2 if it was already open
    //Disable UART interrupts
    ConfigIntUART2(UART_RX_INT_DIS | UART_RX_INT_PR6 | UART_TX_INT_DIS | UART_TX_INT_PR6);
    //UART initialized to 9600 baudrate @BRGH=0, 8bit,no parity and 1 stopbit
    OpenUART2(UART_EN, UART_TX_ENABLE, 25);
    // Sender Tasks  Creation
    xTaskCreate(vSenderTask, "Task1", 150, p1, 2, NULL);
    xTaskCreate(vSenderTask, "Task2", 150, p2, 2, NULL);
```

1

20.3. L'affichage qui se faisait par la fonction C standard **printf( )** , sera remplacé par un envoi vers UART2 en utilisant deux fonctions **printfs(unsigned int *p)** qui permet d'envoyer une chaine de caractère pointée par p et se terminant par un '\0', et une autre fonction **printfi(unsigned int i)** qui

permet de convertir un nombre entier sur 16bits non signé en une chaine de caractères puis d'envoyer cette chaine de caractères vers UART2.

```
76  void vSenderTask(void *pv) {
77      portBASE_TYPE xStatus;
78
79      while (1) { // une tâche est une boucle infinie
80          xStatus = xQueueSendToBack(xQueue, pv, 100); //ne se bloque jamais
81          if (xStatus != pdPASS) //Queue is full
82          {
83              //printf("queue full\r");
84              printfs((unsigned int *) "queue full\r");
85          } else {
86              ((ItemStruct *) pv)->usMsgNum++; //increments Message Number
87          }
88          taskYIELD(); //forces yielding
89      }
90  }
```

```
91  void vReceiverTask(void *pv) {
92
93      portBASE_TYPE xStatus;
94      ItemStruct AnItem; //received item
95      portTickType xTicksToWait = 0;
96      for (;;) {
97          if (uxQueueMessagesWaiting(xQueue) != 0)
98              //printf("should be empty\r\n");
99              printfs((unsigned int *) "should be empty\r\n");
100
101         xStatus = xQueueReceive(xQueue, &AnItem, xTicksToWait);
102         if (xStatus == pdPASS) {
103             //printf("From :%s Msg Numb: %d\r\n", AnItem.Message, AnItem.usMsgNum);
104             printfs((unsigned int *) "From :");
105             printfs((unsigned int *) AnItem.Message);
106             printfi((unsigned int) AnItem.usMsgNum);
107             printfs((unsigned int *) "\r\n");
108
109         } else printfs((unsigned int *) "Cannot receive from Queue!!\r\n");
110
111     }
112
113 }
```

20.4. Ajoutez les déclarations et de ces deux fonctions dans le même fichier main.c.

```
//functions declaration
void vSenderTask(void *pv1);
void vReceiverTask(void *pv2);
void vApplicationIdleHook(void);
void printfs(unsigned int *);
void printfi(unsigned int);
```

et leurs définitions

```
void printfs(unsigned int *p) {
    putsUART2(p);
}
void printfi(unsigned int i){
    char chaine[6]; //au max 5 chiffres
    sprintf(chaine,"%d",i); //printf dans chaine le nombre i
    putsUART2(chaine);//envoyer la chaine construite vers le port UART2
}
```

20.5. Vérifiez le bon fonctionnement du port UART2 en construisant le projet et testez quelques situations d'échange entre les Sender et le Receiver, déjà étudiées avec le simulateur MPLABX (en modifiant les priorités des tâches et en utilisant ou non taskYIELD( )).