Informatique Industrielle - Multi-Tâche et Temps Réel
Partie 2 : FreeRTOS et programmation temps-réel de systèmes embarqués

Romain Delpoux, Arnaud Lelevé, Minh Tu Pham

Maîtres de Conférences.
Email : prenom.nom@insa-lyon.fr.
Laboratoire Ampère UMR CNRS 5005.

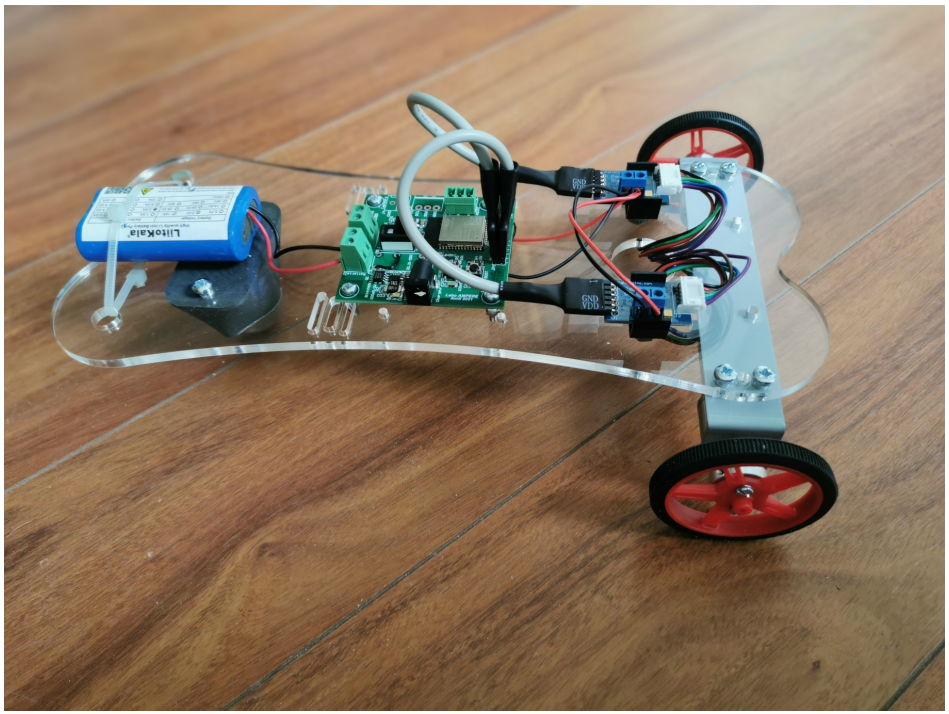
Présentation du cours

Programme

- Cours 1 : introduction générale
- Cours 2 : FreeRTOS et programmation temps-réel de systèmes embarqués
- TP 1 : programmation "basique" d'un robot mobile
- TP 2 : programmation temps réel sur système embarqué
- TP 3 : programmation temps-réel d'un robot mobile

Introduction

Fil conducteur : Pilotage d'un robot mobile



Objectif du robot :

- Asservissement de vitesse et orientation du robot.
- Pilotage via WIFI.

Introduction

Fil conducteur : Pilotage d'un robot mobile

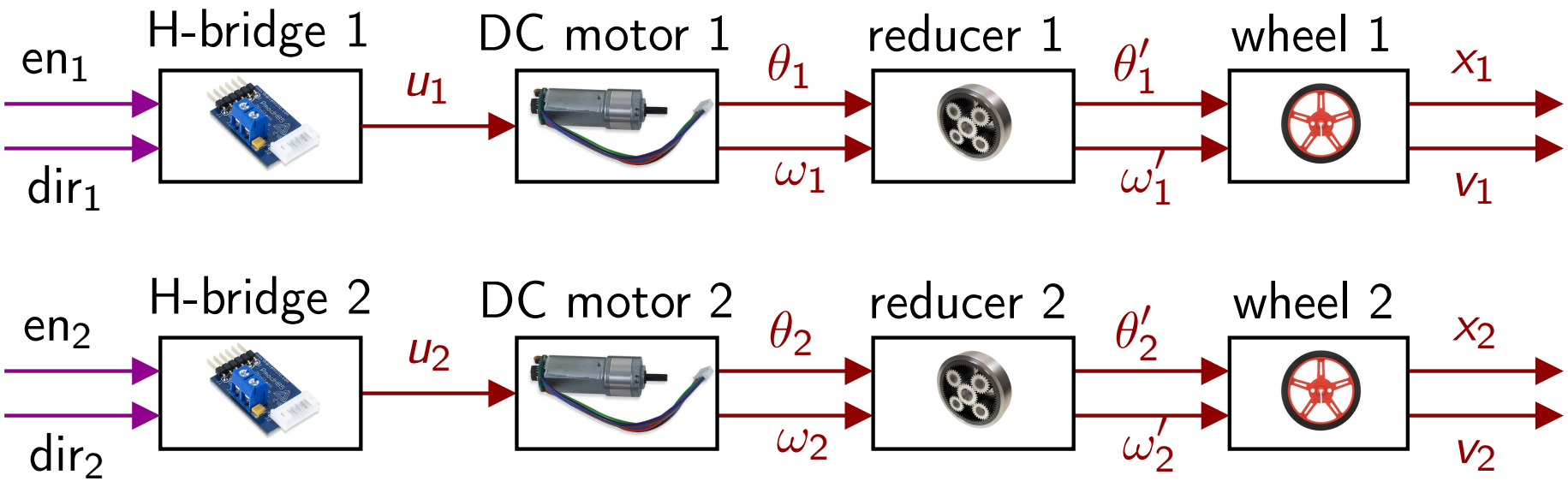


Figure – Schéma bloc du système en boucle ouverte

Alimentation du moteur : Pont en H, MLI

DIR	EN	MOTOR
x	0	STOP
0	1	REV
1	1	FWD

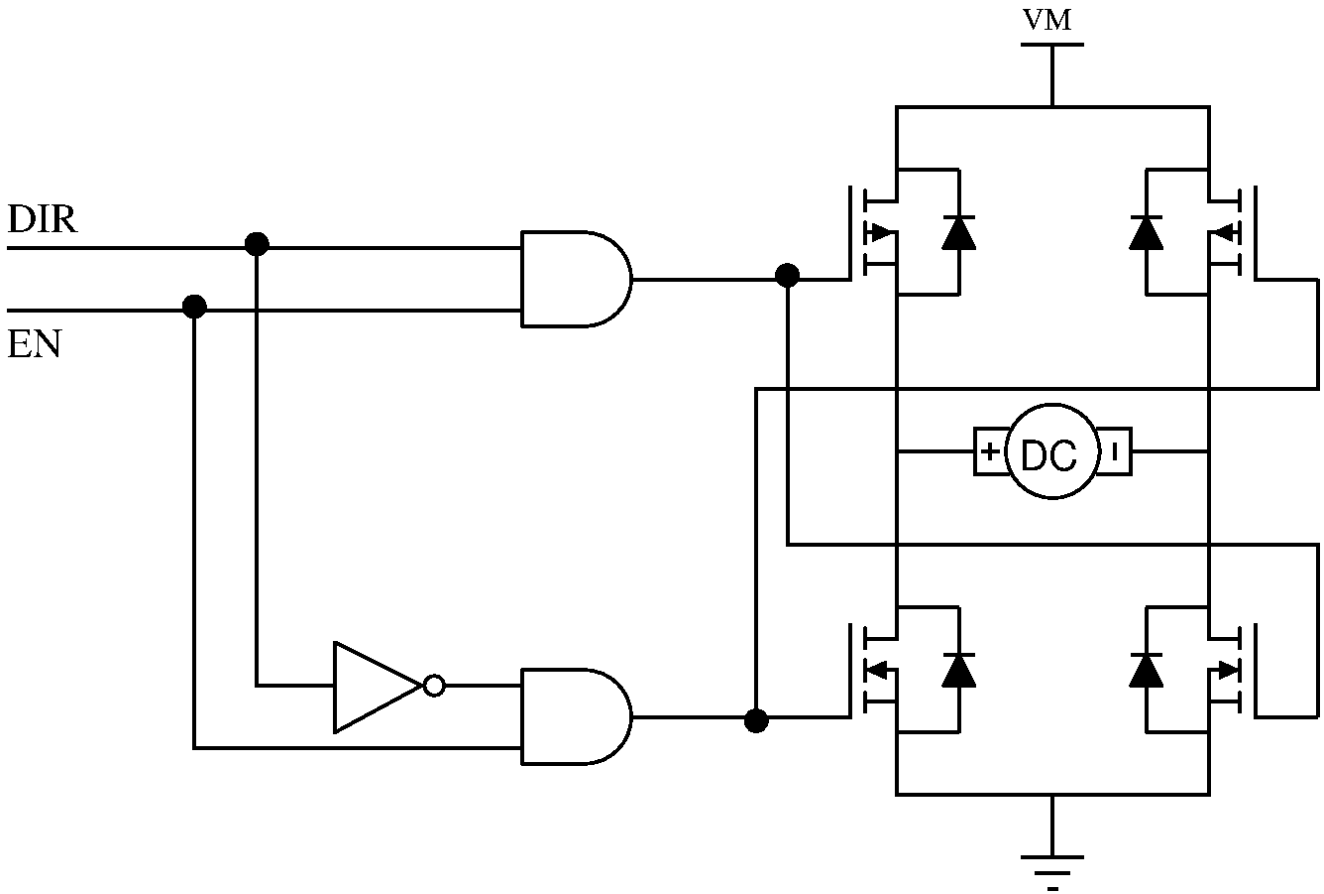


Figure – Fonctionnement d'un pont en H

Mesure de position : Encodeur

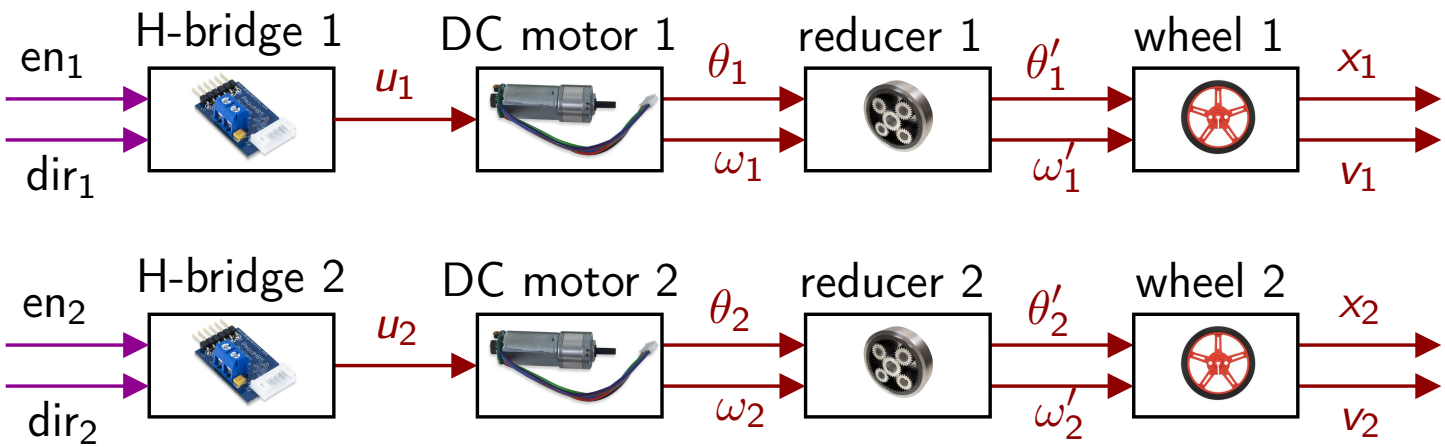


Figure – Mesure de position

Mesure de position : Encodeur

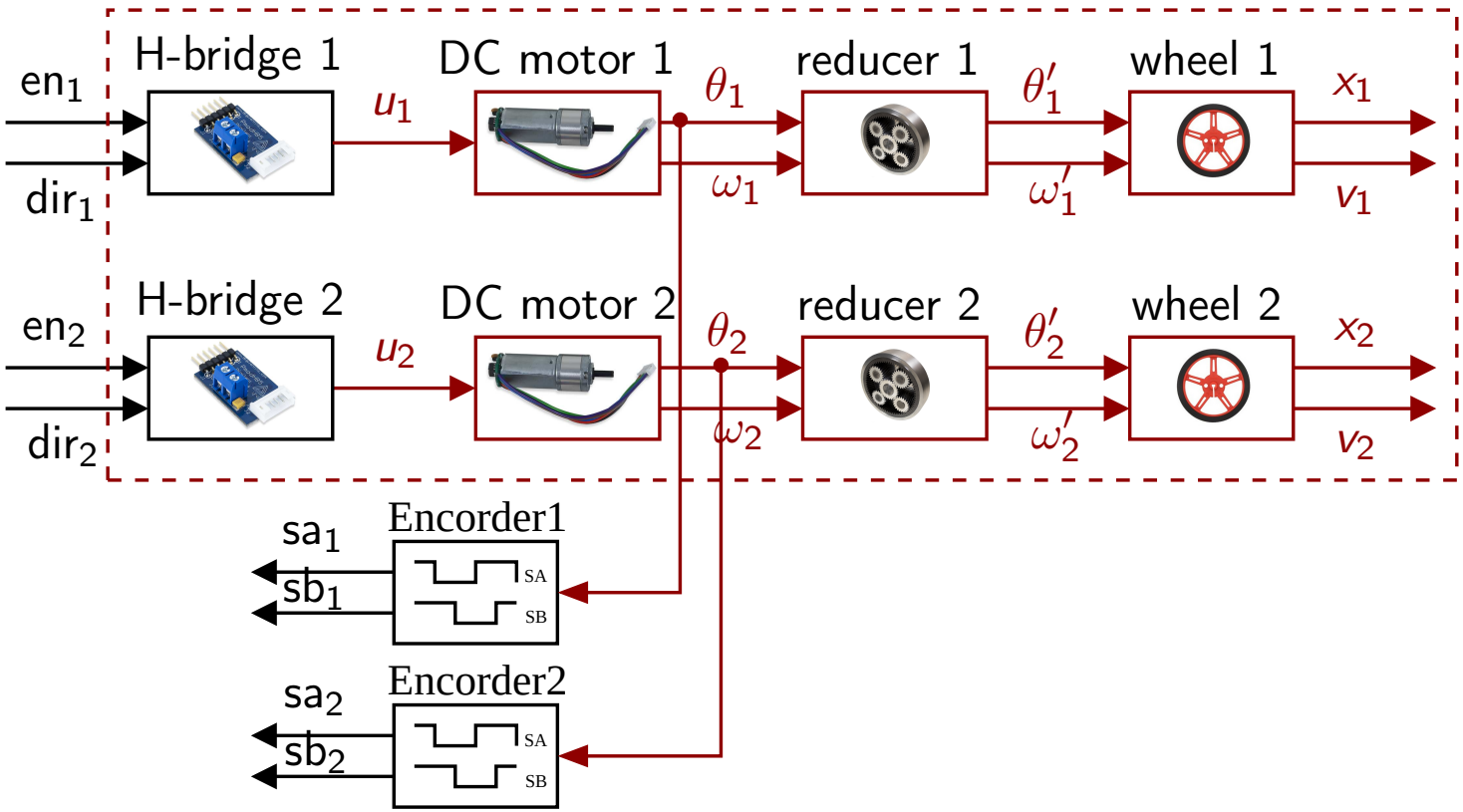


Figure – Mesure de position

Mesure de position : Encodeur

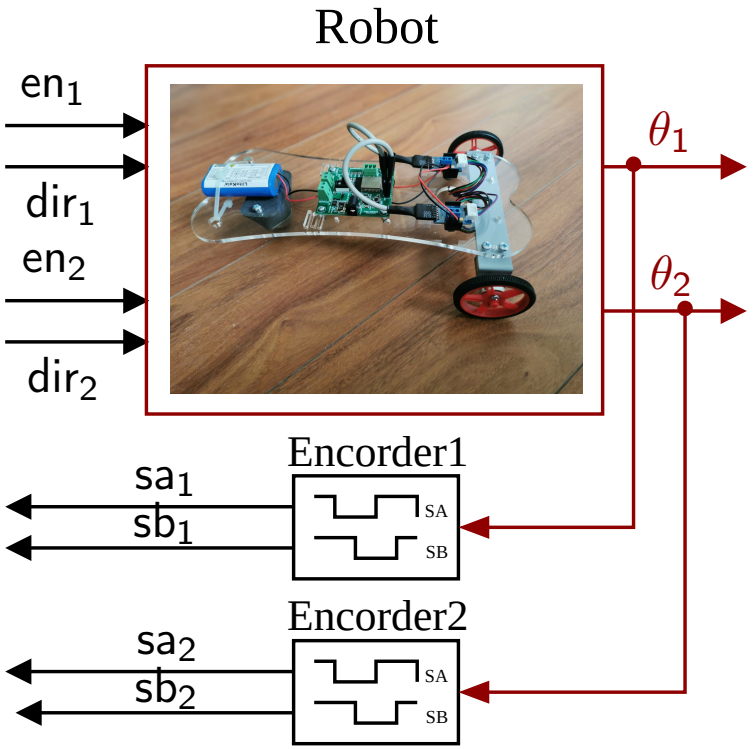


Figure – Mesure de position

Principe de l'encodeur

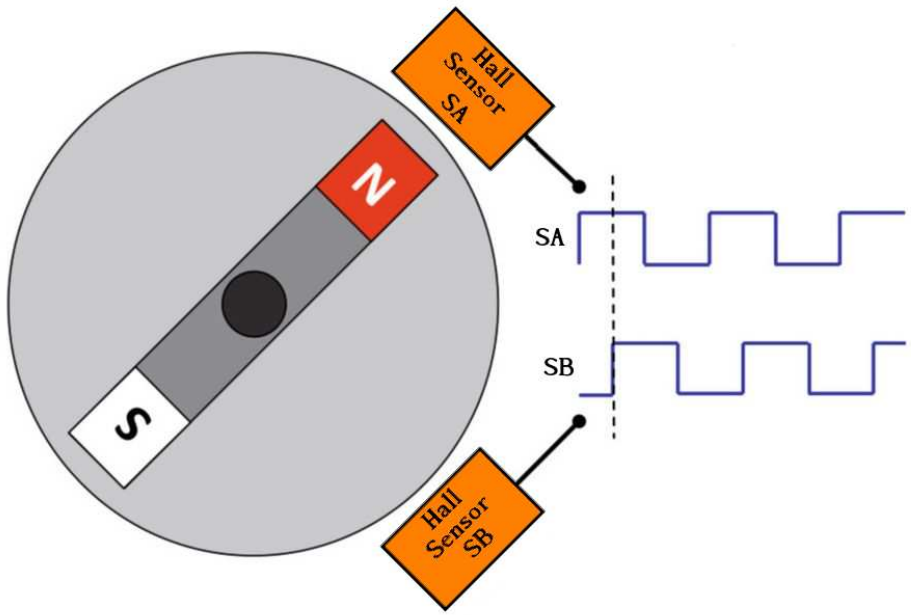


Figure – Principe de l'encodeur

Nécessité d'avoir un comptage rapide : Interruptions

Introduction

Asservissement numérique

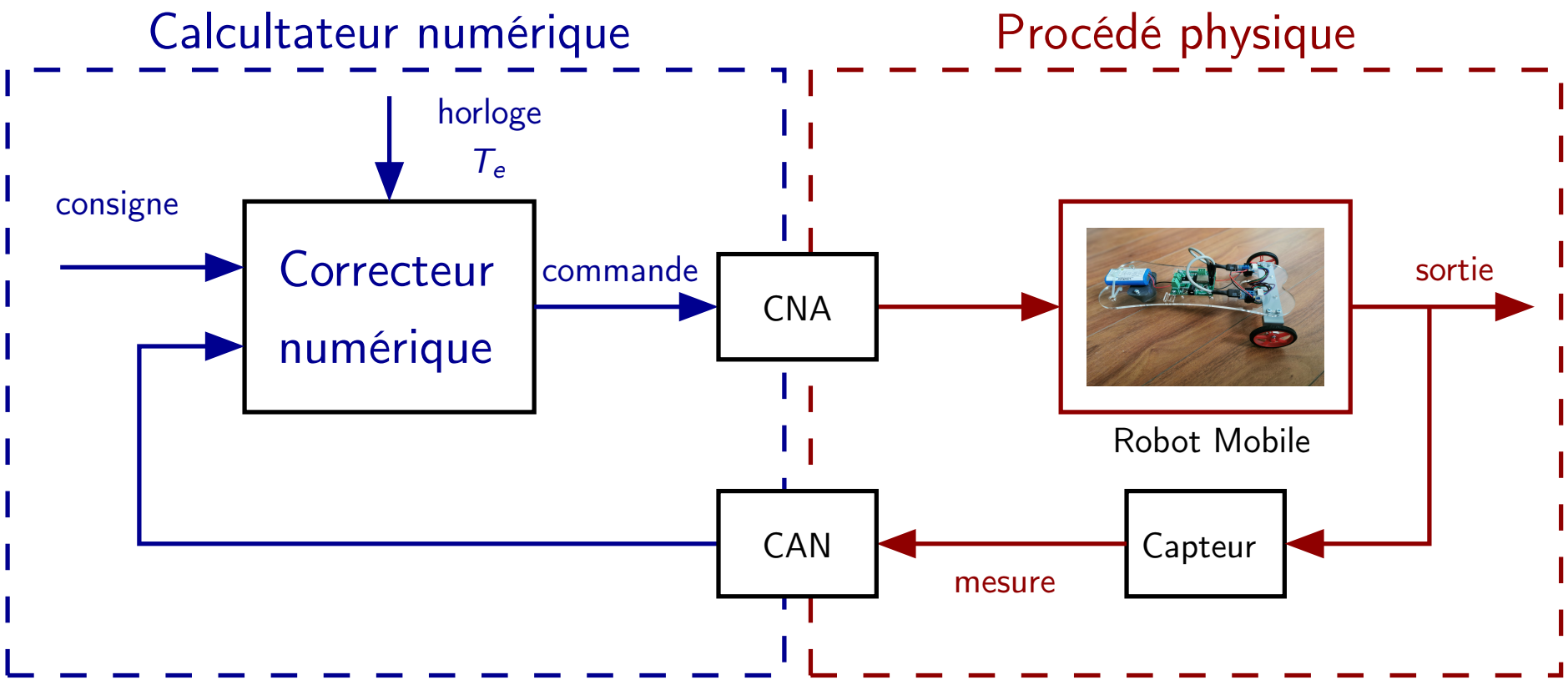


Figure – Principe de le commande numerique

Nécessité d'avoir une horloge régulière : tâche périodique

Introduction

En résumé

- des interruptions pour les codeurs
- tâche périodique rapide pour l'asservissement
- tâche de priorité inférieure pour la communication WIFI

Besoin d'un système temps réel, multi-tâches

En résumé

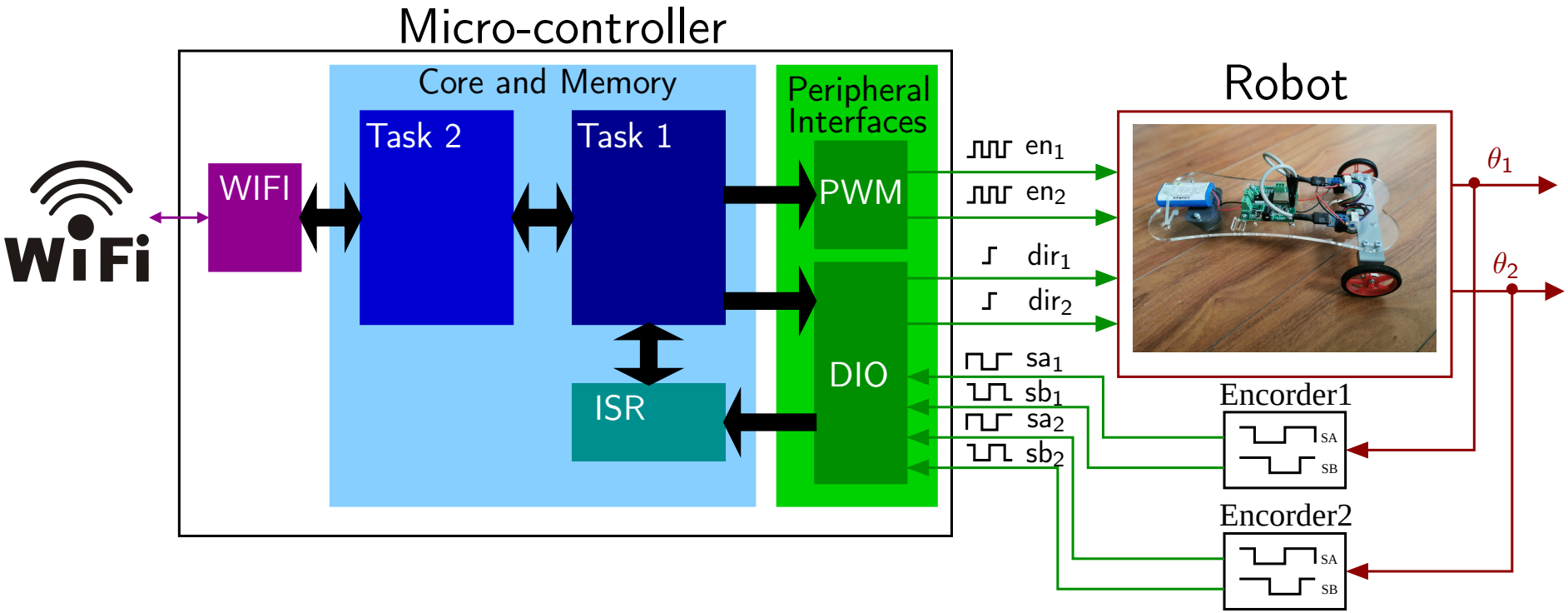


Figure – Architectue globale du système

Sommaire

Sommaire

Microcontrôleur ESP32

Caractéristiques

Asservissement et WIFI oriente le choix vers un microcontrôleur ESP32 :

ESP32-WROOM-32E

Caractéristiques :

- CPU et mémoire
  - Dual Core
  - Fréquence max de l'horloge : 240MHz
  - Flash : 16MB
- Périphériques
  - UART,
  - SPI, SDIO, I2C,
  - LED PWM, Motor PWM,
  - pulse counter,
  - GPIO
  - ...

Sommaire

Real Time Operating System (RTOS)

FreeRTOS

FreeRTOS est un système d'exploitation temps réel (RTOS) faible empreinte, portable, préemptif et open source pour microcontrôleur. Il a été porté sur plus de 40 architectures différentes. Créé en 2003 par Richard Barry et la FreeRTOS Team, il est aujourd'hui parmi les plus utilisés dans le marché des systèmes d'exploitation temps réel. (Source wikipédia)

FreeRTOS a été téléchargé toutes les 170 secondes en 2019 !

Real Time Operating System (RTOS)

Pourquoi FreeRTOS ?

- Fournit une solution unique pour de nombreuses architectures
- Est connu pour être fiable.
- ROM, RAM et surcharge de traitement minimales.

Une image du noyau $\approx$ 6K à 12K octets.

- C'est très simple - le noyau contenu dans seulement 3 fichiers C.
- Est vraiment gratuit pour une utilisation dans des applications commerciales.
- Est bien établi avec une base d'utilisateurs importante et toujours croissante.
- Possède un forum de support gratuit, surveillé et actif.
- Fournit une documentation abondante.
- FreeRTOS offre une alternative de traitement en temps réel plus petite et plus simple.

Real Time Operating System (RTOS)

Definition

FreeRTOS est adapté pour déployer des applications temps réel embarquées qui utilisent des microcontrôleurs ou de petits microprocesseurs.

- Ce type d'application comprend normalement un mélange d'exigences en temps réel matérielles et logicielles ;
  - Les exigences en temps réel souple sont celles qui indiquent une échéance, mais le dépassement de l'échéance ne rendrait pas le système inutile ;
  - Les exigences en temps réel stricte sont celles qui indiquent une échéance, et le non-respect de l'échéance entraînerait une défaillance absolue du système.

Real Time Operating System (RTOS)

Définition

- FreeRTOS est un noyau temps réel.
- Organisation : collection de tâches d'exécution indépendantes.
- Le noyau décide quelle tâche doit être exécuté en examinant sa priorité.
- Dans le cas le plus simple, le concepteur pourrait attribuer
  - des priorités plus élevées aux tâches qui implémentent des exigences en temps réel strictes
  - des priorités inférieures aux threads qui implémentent des exigences en temps réel souples

Notions de tâche

- Les tâches sont de simples fonctions
- Le nombre de tâches exécutés simultanément et leur priorité n'est limités que par le matériel

```
1 void vATaskFunction( void *pvParameters ) // <- une tache
  {
3     for( ;; ) // <- boucle infinie
      {
5         /* Ajouter le code de votre tache ici */
      }
7 }
```

Créer une tâche

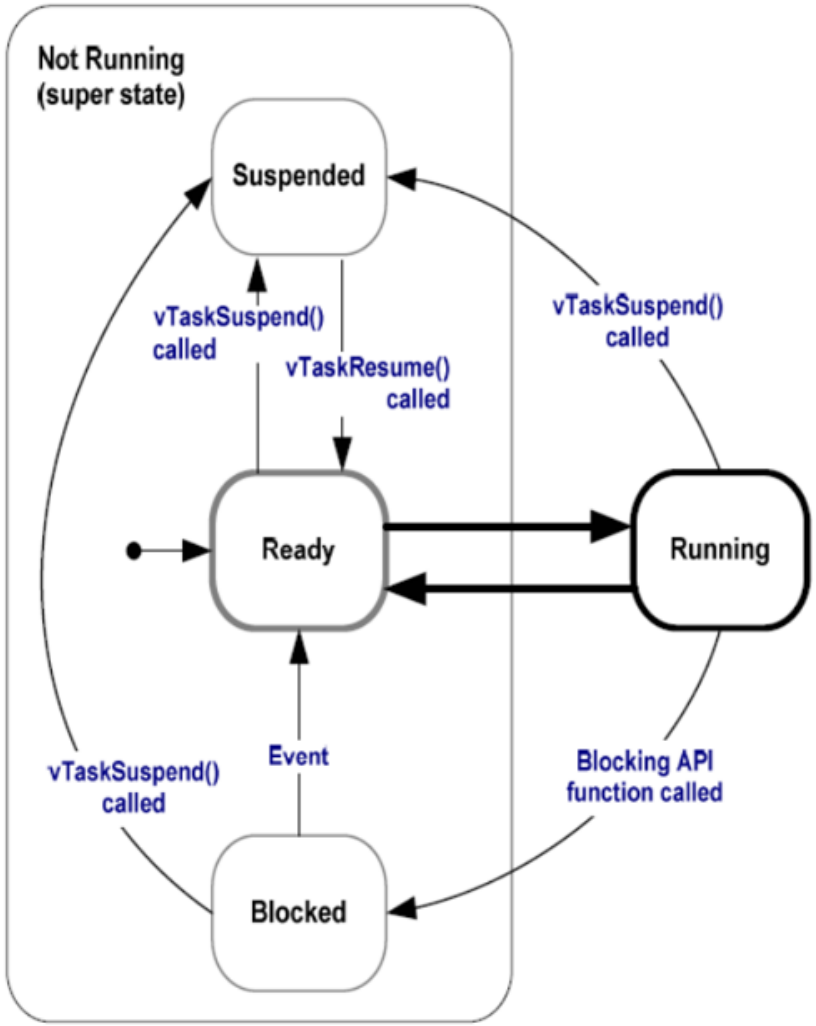
On crée une tâche avec un appel à la fonction `xTaskCreate()`.
Cette fonction accepte les arguments suivants :

- pvTaskCode : pointeur sur la fonction qui implémentera la tâche
- pcName : nom de la tâche
- usStackDepth : taille (en nombre de mots) de la pile
- pvParameters : pointeur vers un paramètre passée à la fonction de tâche
- uxPriority : priorité de la tâche (int)
- pxCreatedTask : descripteur de la tâche

Cette fonction retourne pdPASS en cas de succès ou un code d'erreur .

```
xTaskCreate(
    vATaskFunction, /* Task function. */
    "vATaskFunction", /* name of task. */
    10000, /* Stack size of task */
    NULL, /* parameter of the task */
    1, /* priority of the task */
    NULL); /* Task handle to keep track of created task */
```

État des tâches



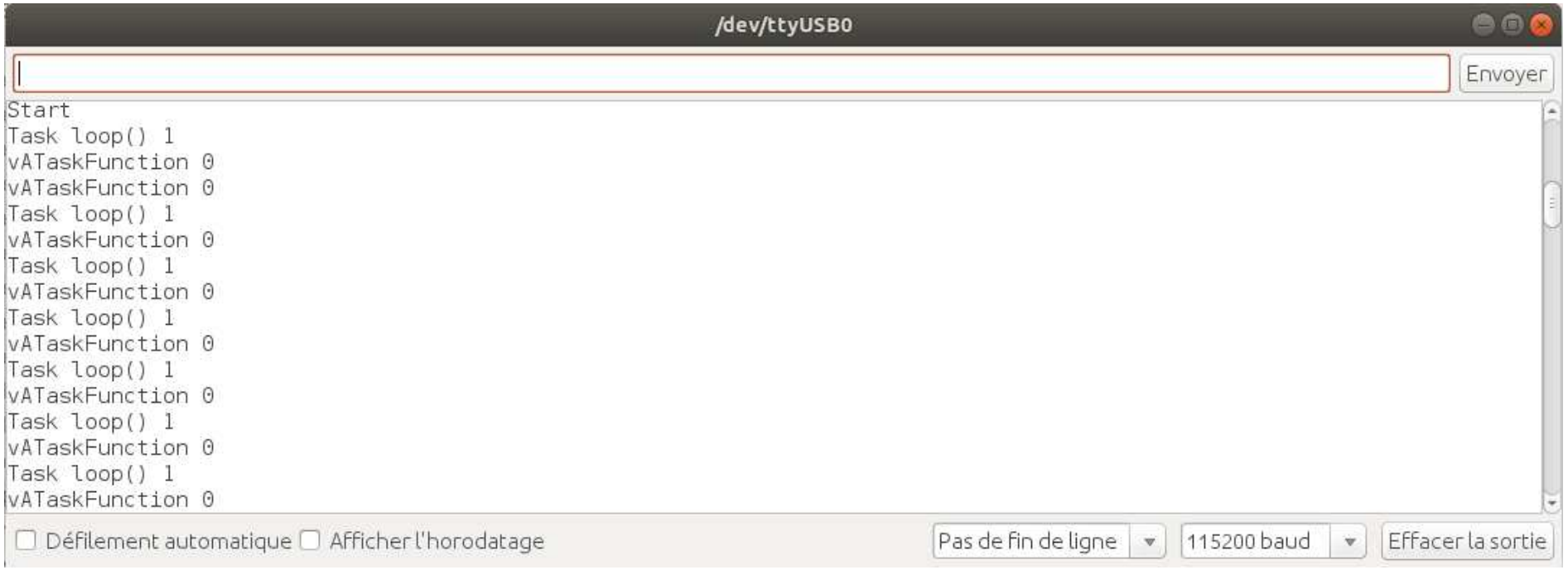
État des tâches

- Running (en cours) : il s'agit de la tâche en cours d'exécution par le CPU.
- Ready (prêt) : les tâches dans cet état sont prêtes à être exécutées.
- Blocked (bloqué) : les tâches dans cet état sont en attentes d'un événement pour se réveiller.
- Suspended (suspendu) : Cet état, peu utilisé, est propre à FreeRTOS et est à manier avec précaution. Une tâche dans cet état n'est plus vue de l'ordonnanceur.

Exemple1.ino

```
1  void vATaskFunction( void *pvParameters ) // <- une tache
   {
3    for ( ;; ) // <- boucle infinie
     {
5      Serial.printf("vATaskFunction %d\n", xPortGetCoreID());
       delay(1000);
7    }
   }
9
   void setup()
11 {
     Serial.begin(115200);
13   while (!Serial);
     Serial.println("Start");
15
     xTaskCreate(
17     vATaskFunction, /* Task function. */
       "vATaskFunction", /* name of task. */
19     1000, /* Stack size of task */
       NULL, /* parameter of the task */
21     1, /* priority of the task */
       NULL); /* Task handle to keep track of created task */
23 }

25 void loop()
   {
27   Serial.printf("Task loop() %d\n", xPortGetCoreID());
     delay(1000);
29 }
```

Exemple1.ino



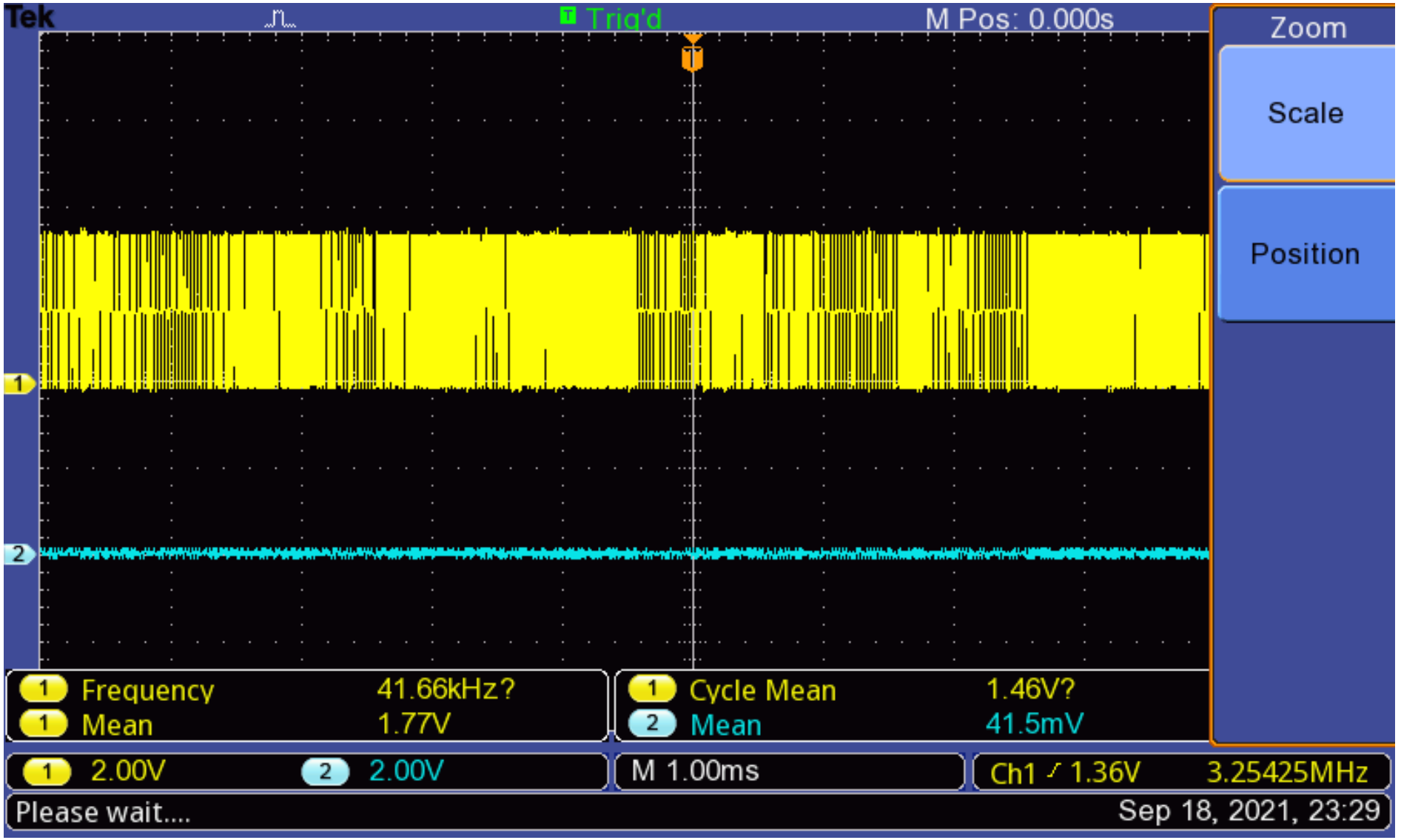
Remarque

2 tâches s'exécutent (`vATaskFunction()` *et* `loop()`*) sur les 2 coeurs de l'ESP32. Il est possible d'empêcher le lancement de* `loop()` *en plaçant une boucle infinie à la fin de setup().*

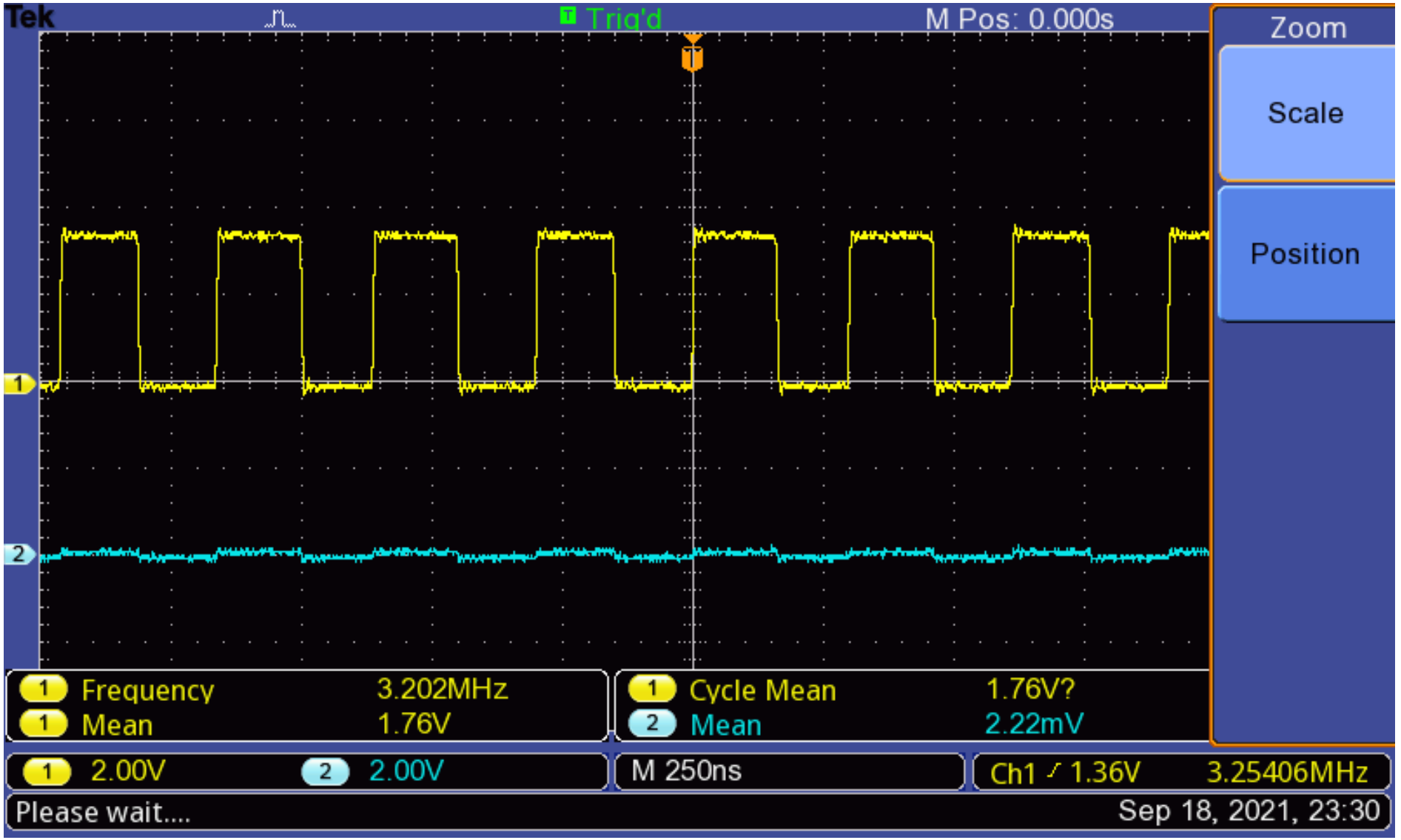
TacheSimple.ino

```
uint8_t oscillo1     = 18;
uint8_t oscillo2     = 19;
int task1_prio = 1;
TaskHandle_t Task1;

void vTask1( void *pvParameters )
{
  bool etat = LOW; int i = 0;
  // boucle infinie
  for ( ;; )
  {
    for ( i = 0; i < 10000; i++ ) {
      if (etat == HIGH)
        etat = LOW;
      else
        etat = HIGH;
      digitalWrite(oscillo1, etat);
    }
  }
}

void setup()
{
  Serial.begin(115200);
  while (!Serial);
  pinMode(oscillo1, OUTPUT);
  xTaskCreate(vTask1, "Task1", 10000, NULL,  task1_prio,  &Task1 );
  for ( ;; ) {
  }
}

// fonction appelee par defaut
void loop() {}
```

TacheSimple.ino



TacheSimple.ino



Real Time Operating System (RTOS)

`vTaskDelay()`

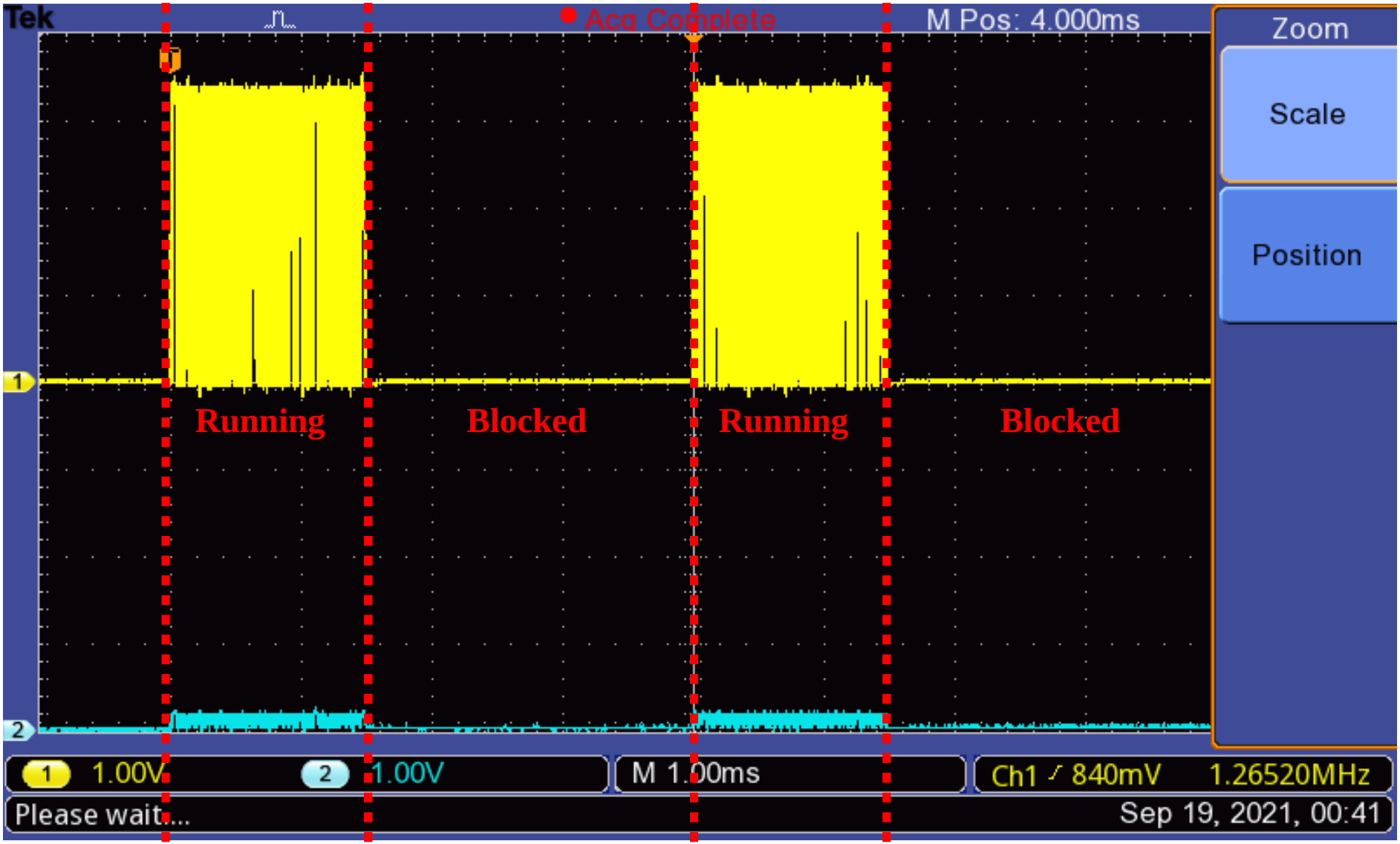
- FreeRTOS fournit une fonction `vTaskDelay()` qui place la tâche appelante dans l'état Bloqué (Blocked) pour un nombre fixe d'interruptions de tick. À la fin de ce délai, la tâche repasse à l'état Prêt (Ready).
- La macro `pdMS_TO_TICKS()` peut être utilisée pour convertir un temps spécifié en millisecondes en un nombre de ticks.

```
vTaskDelay( pdMS_TO_TICKS( 3 ) );
```

Tache Simple Retardée

```
void vTask1( void *pvParameters )
{
  bool etat = LOW;
  int i = 0;
  // boucle infinie
  for( ;; )
  {
    for( i = 0; i<10000; i++ ) {
      if (etat == HIGH)
        etat = LOW;
      else
        etat = HIGH;
      digitalWrite(oscillo1, etat);
    }
     // Delai de 3ms
     vTaskDelay( pdMS_TO_TICKS( 3 ) );
  }
}
```

Tache Simple Retardée



Tâches périodique `vTaskDelayUntil()`

- La fonction `vTaskDelayUntil()` retarde une tâche jusqu'à un nombre de ticks spécifié.
- Cette fonction peut être utilisée par des tâches périodiques pour assurer une fréquence d'exécution constante.

```
vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS( 3 ) );
```

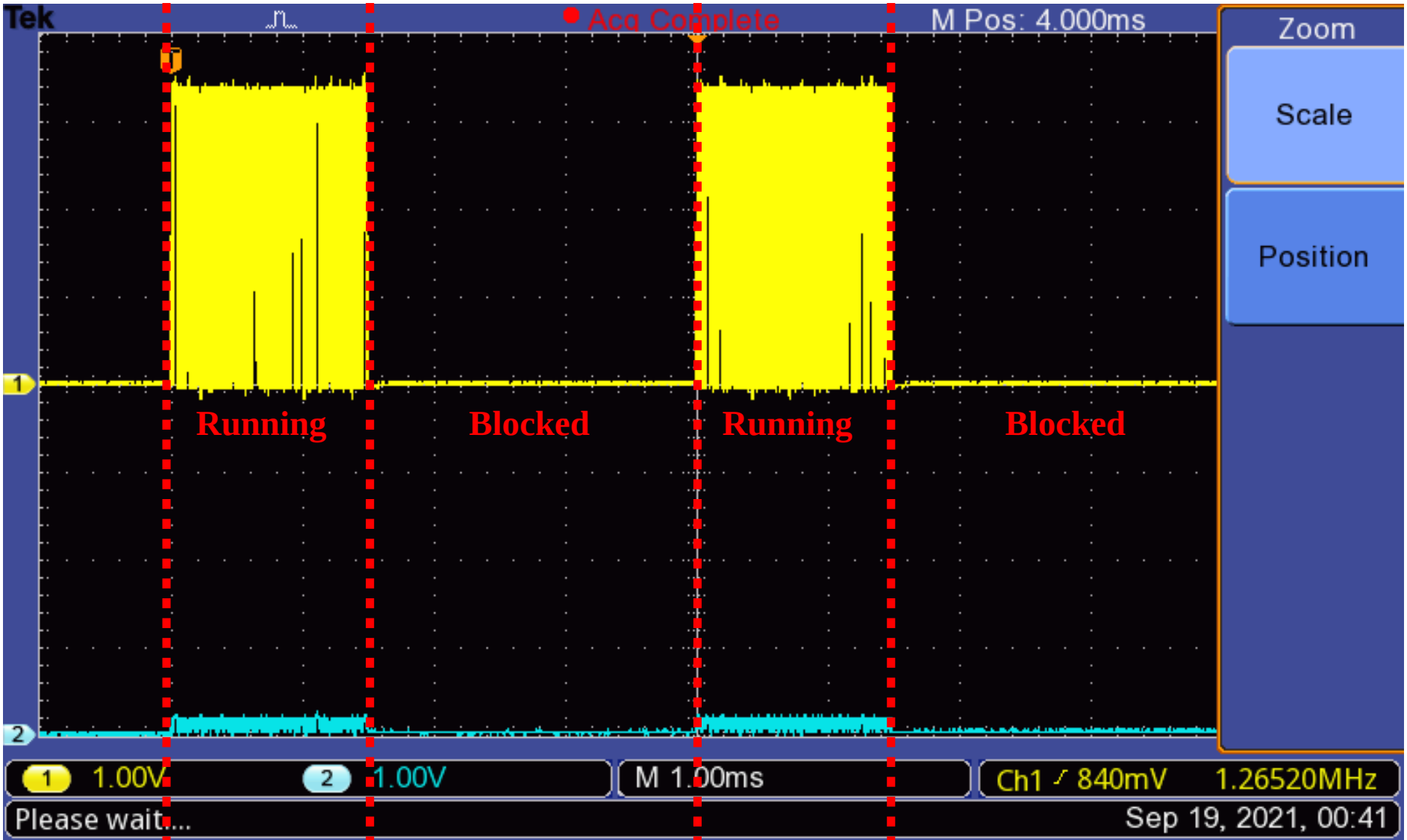
- `vTaskDelay()` spécifie une heure à laquelle la tâche souhaite se débloquer par rapport à l'heure à laquelle `vTaskDelay()` est appelée, tandis que `vTaskDelayUntil()` spécifie une heure absolue à laquelle la tâche souhaite se débloquer.

Tâche Simple Périodique

```
void vTask1( void *pvParameters )
{
  bool etat = LOW;
  int i = 0;
  TickType_t xLastWakeTime;
  xLastWakeTime = xTaskGetTickCount();
  // boucle infinie
  for( ;; )
  {
    for( i = 0; i<10000; i++ ) {
      if (etat == HIGH)
        etat = LOW;
      else
        etat = HIGH;
      digitalWrite(oscillo1, etat);
    }
    vTaskDelayUntil(&xLastWakeTime,pdMS_TO_TICKS(3));
  }
}
```

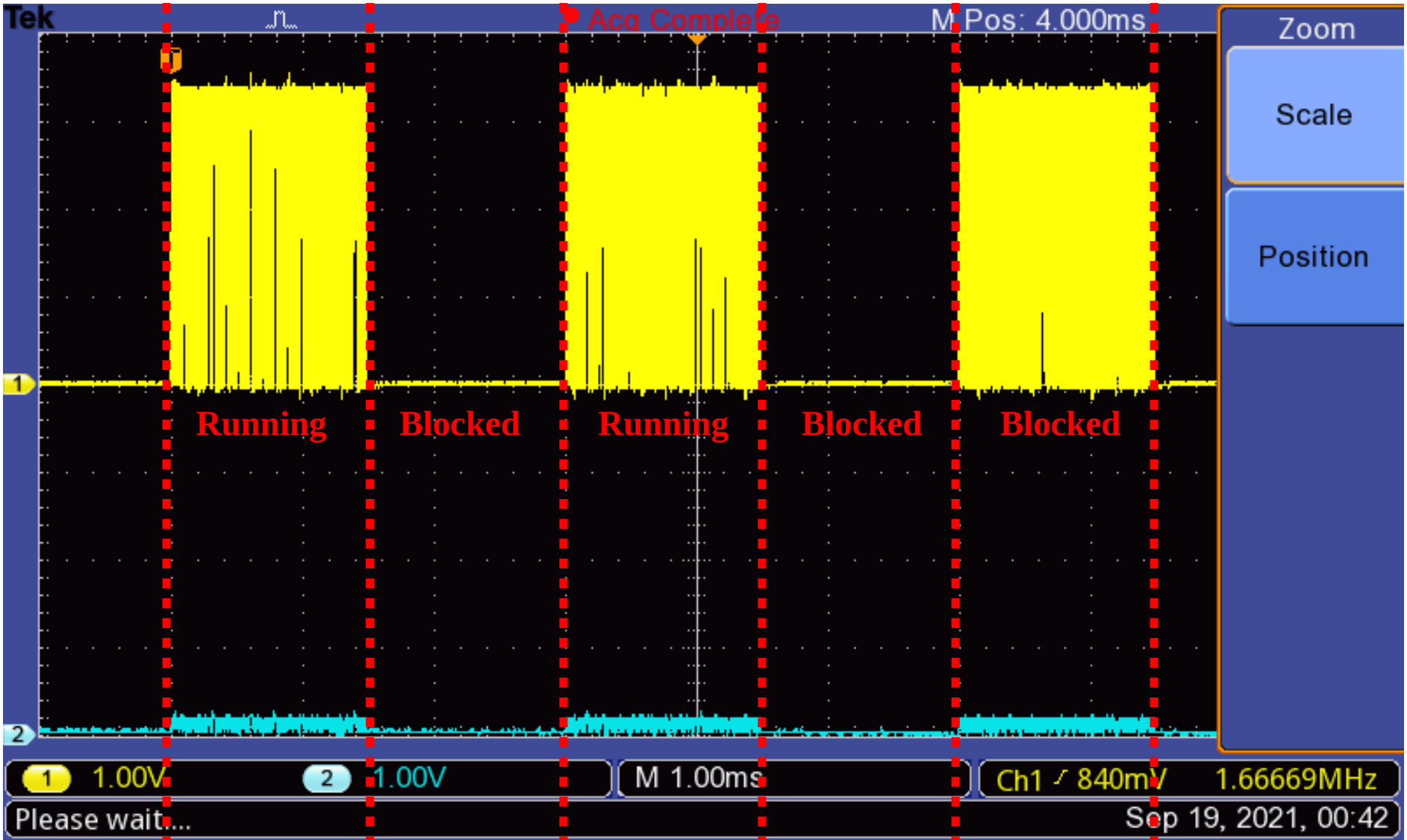
Tâche Simple Périodique

```
vTaskDelay( pdMS_TO_TICKS( 3 ) );
```



Tâche Simple Périodique

```
vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS( 3 ) );
```



Tâche v.s. fonction

- Une différence très importante est qu'à chaque tâche est associée

  un TCB (Task Control BLock).

- Un TCB est une structure de données décrivant une tâche
- L'ordonnanceur utilise ensuite les TCB pour le management de son environnement multitâches.
- Sous FreeRTOS, un TCB comporte par exemple :
  - la priorité de la tâche
  - le ou les événements qu'elle attend
  - le pointeur de base de la pile associée à la tâche
  - le pointeur de sommet de pile
  - ...

Tâche v.s. fonction

- Ce sont les TCB que l'ordonnanceur analyse afin de savoir à quelle tâche prendre ou donner du temps CPU.
- Un programme classique n'est constitué que d'un main() voir d'ISR, aucun OS embarqué.

Algorithme d'ordonnancement

- L'ordonnanceur de FreeRTOS utilise un algorithme de

  Round Robin

- un algorithme Round Robin
  - ne garantit pas que le temps est partagé également entre les tâches de priorité égale,
  - seulement que les tâches à l'état Ready de priorité égale entreront à leur tour dans l'état Running.
  - une unité de temps (time quantum ou time slice) est définie.

Le tourniquet d'un jeu de parcs est l'idée intuitive derrière cet algorithme.

Algorithme d'ordonnancement

- Algorithme d'ordonnancement configuré sur Arduino :

  Pre-emptive Scheduling with Time Slicing

- Pre-emptive

Algorithme d'ordonnancement

- Algorithme d'ordonnancement configuré sur Arduino :

  Pre-emptive Scheduling with Time Slicing

- Pre-emptive :
  - "préempte" immédiatement la tâche d'état *Running* si une tâche qui a une priorité supérieure à la tâche d'état *Running* entre dans l'état *Ready*.
  - être préempté signifie être involontairement hors de l'état *Running* et dans l'état *Ready* permettre à une tâche différente d'entrer dans l'état *Running*.

Algorithme d'ordonnancement

- Algorithme d'ordonnancement configuré sur Arduino :

  Pre-emptive Scheduling with Time Slicing

- Pre-emptive
- Time Slicing

Algorithme d'ordonnancement

- Algorithme d'ordonnancement configuré sur Arduino :

Pre-emptive Scheduling with Time Slicing

- Pre-emptive
- Time Slicing :
  - le découpage temporel est utilisé pour partager le temps de traitement entre des tâches de priorité égale
  - sélectionne une nouvelle tâche pour entrer dans l'état *Running* à la fin de chaque tranche de temps s'il existe d'autres tâches à l'état *Ready* de même priorité
  - Une tranche de temps est égale au temps entre deux interruptions de tick RTOS.

Algorithme d'ordonnancement : Illustration

Deux tâches :

- Tâche 1 : Tâche périodique de 3ms

```
xTaskCreate(vTask1, "Task1", 10000, NULL, 10, &Task1);
```

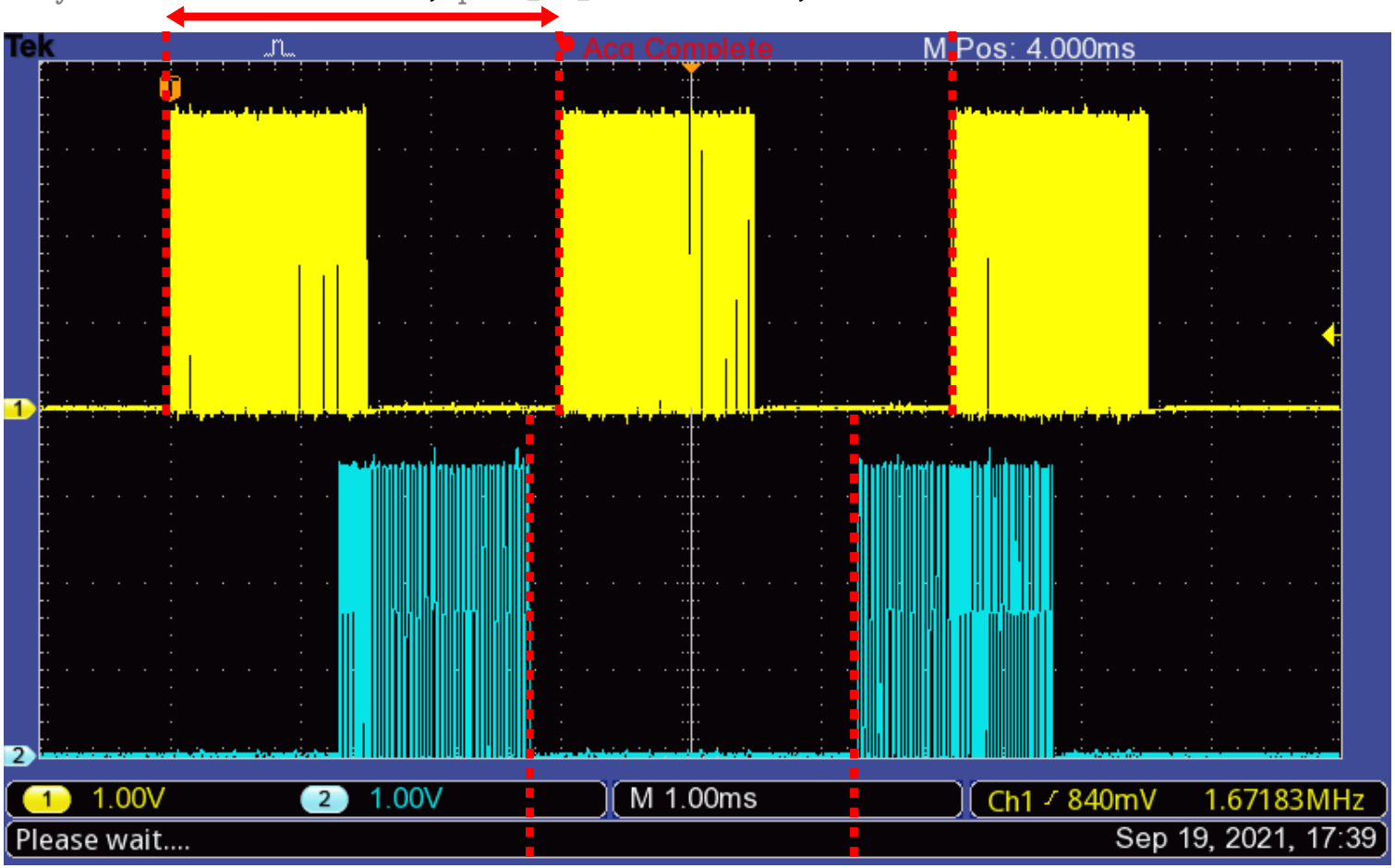
- Tâche 2 : Tache avec un retard de 3ms

```
xTaskCreate(vTask2, "Task2", 10000, NULL, 10, &Task2,1);
```

Les deux tâches ont la même priorité de 10

Algorithme d'ordonnancement : Illustration

```
vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS( 3 ) );
```

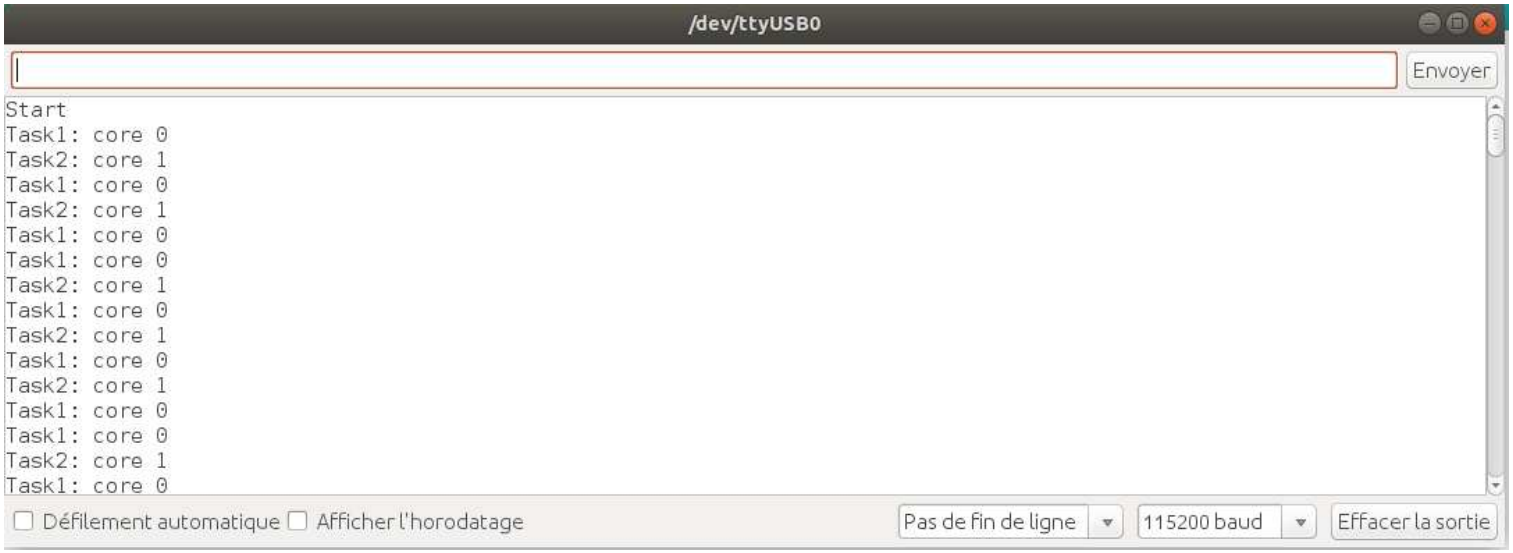



```
vTaskDelay( pdMS_TO_TICKS( 3 ) );
```

Algorithme d'ordonnancement : Illustration

L'ESP32 possède deux cœurs :
Chaque tâche est exécuté simultanément sur les deux coeurs.

```
Serial.printf("Task1: core %d\n", xPortGetCoreID());
```



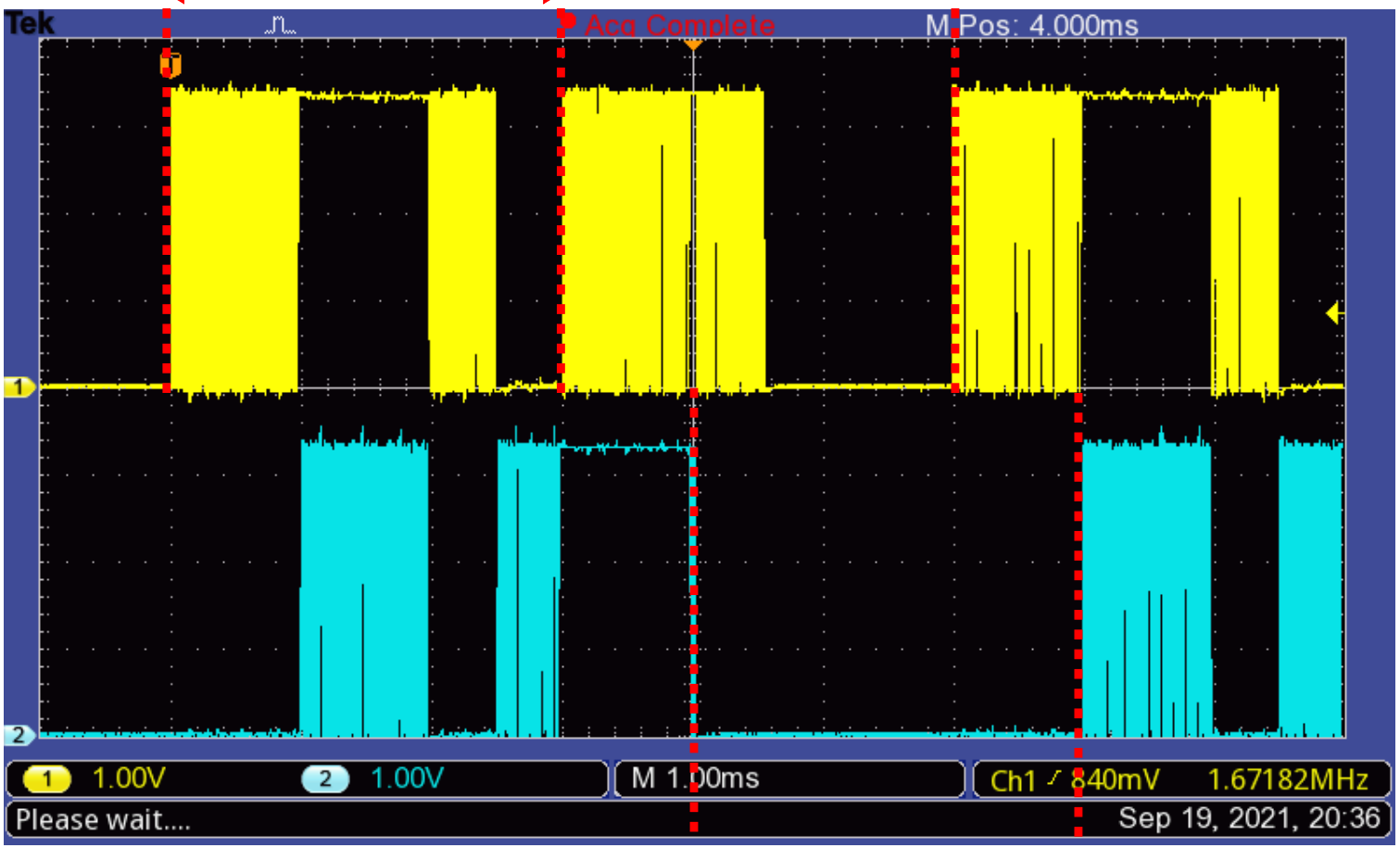
Une tâche seulement peut s'exécuter par cœur

- Affecter une tâche à un cœur

```
xTaskCreatePinnedToCore(vTask1,"Task1", 10000, NULL, 10, &Task1,0);
```

Algorithme d'ordonnancement : Illustration

```
vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS( 3 ) );
```

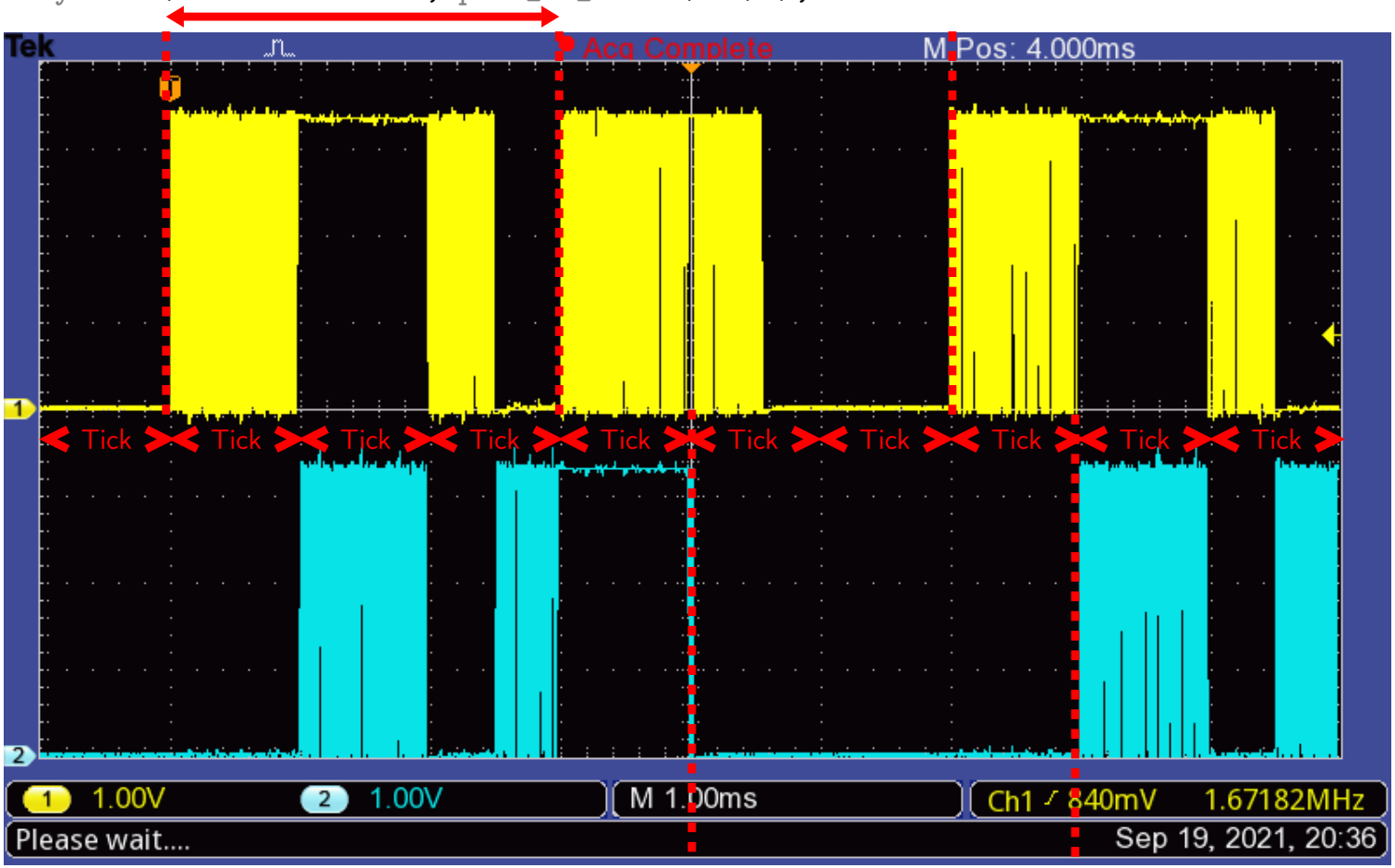



```
vTaskDelay( pdMS_TO_TICKS( 3 ) );
```

Real Time Operating System (RTOS)

Algorithme d'ordonnancement : Illustration

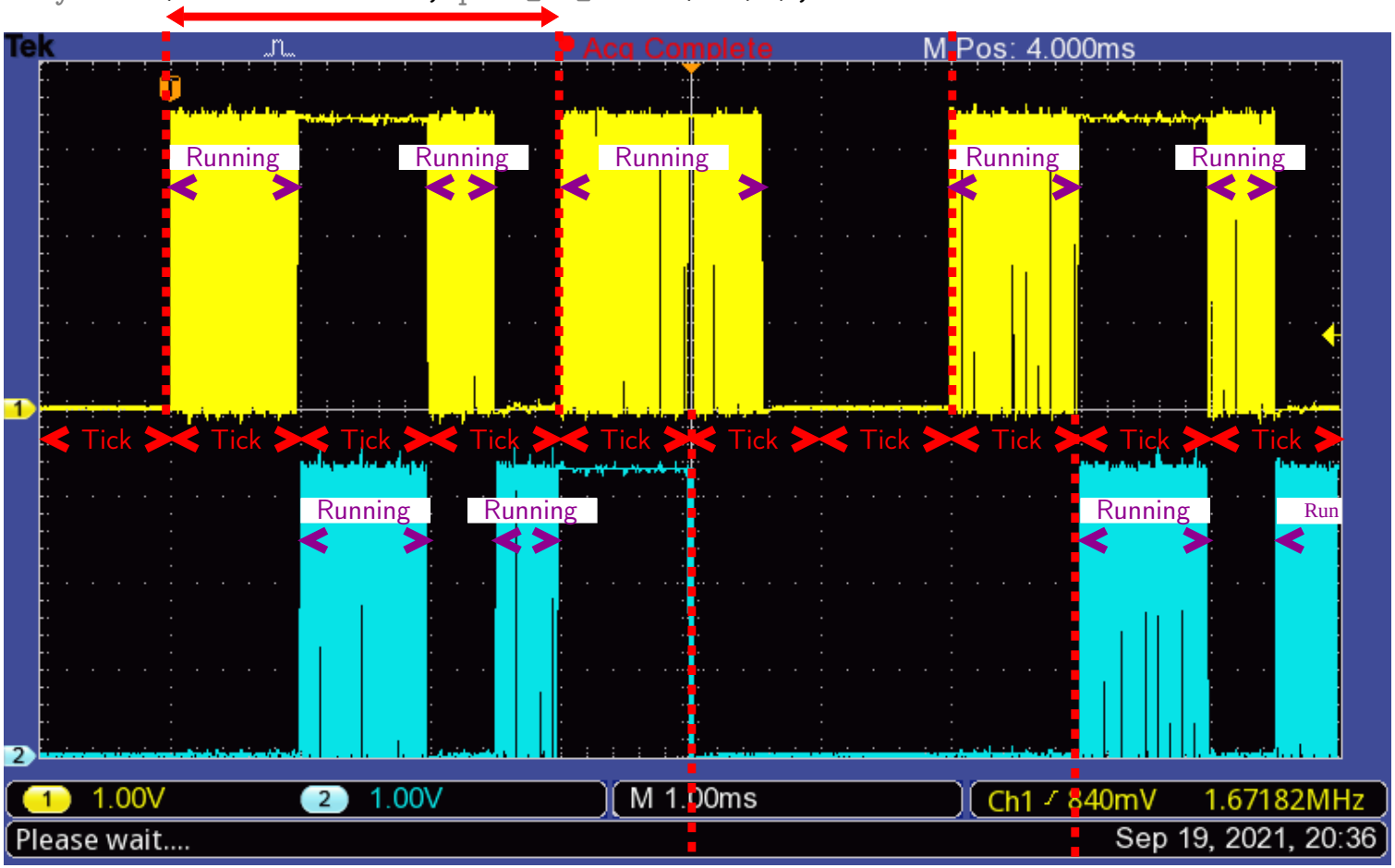
```
vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS( 3 ) );
```




```
vTaskDelay( pdMS_TO_TICKS( 3 ) );
```

Algorithme d'ordonnancement : Illustration

```
vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS( 3 ) );
```

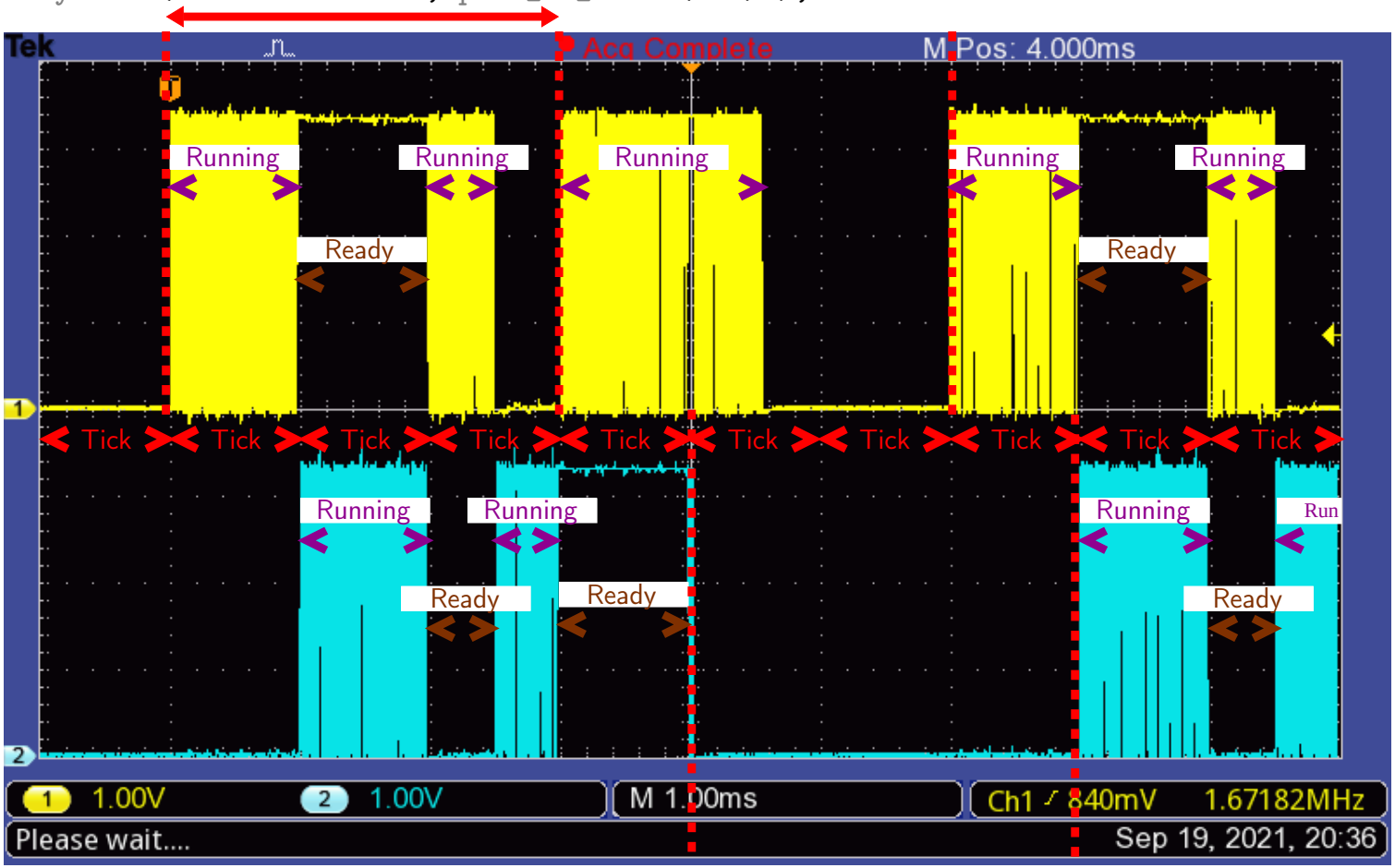



```
vTaskDelay( pdMS_TO_TICKS( 3 ) );
```

Real Time Operating System (RTOS)

Algorithme d'ordonnancement : Illustration

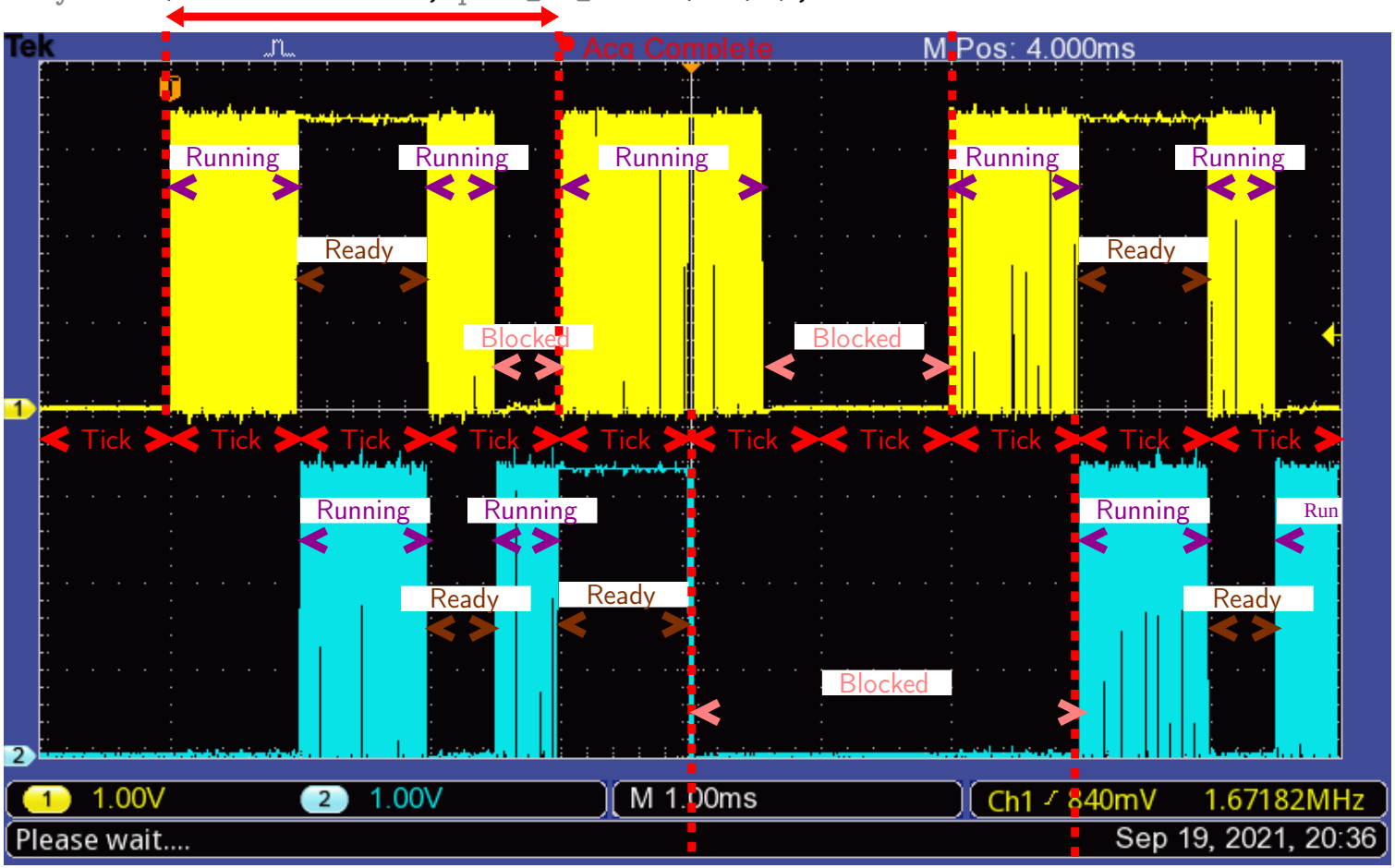
```
vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS( 3 ) );
```




```
vTaskDelay( pdMS_TO_TICKS( 3 ) );
```

Algorithme d'ordonnancement : Illustration

```
vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS( 3 ) );
```




```
vTaskDelay( pdMS_TO_TICKS( 3 ) );
```

Algorithme d'ordonnancement : Illustration

On réduit la priorité de la tâche 2 à 9

- Tâche 1 : Tâche périodique de 3ms

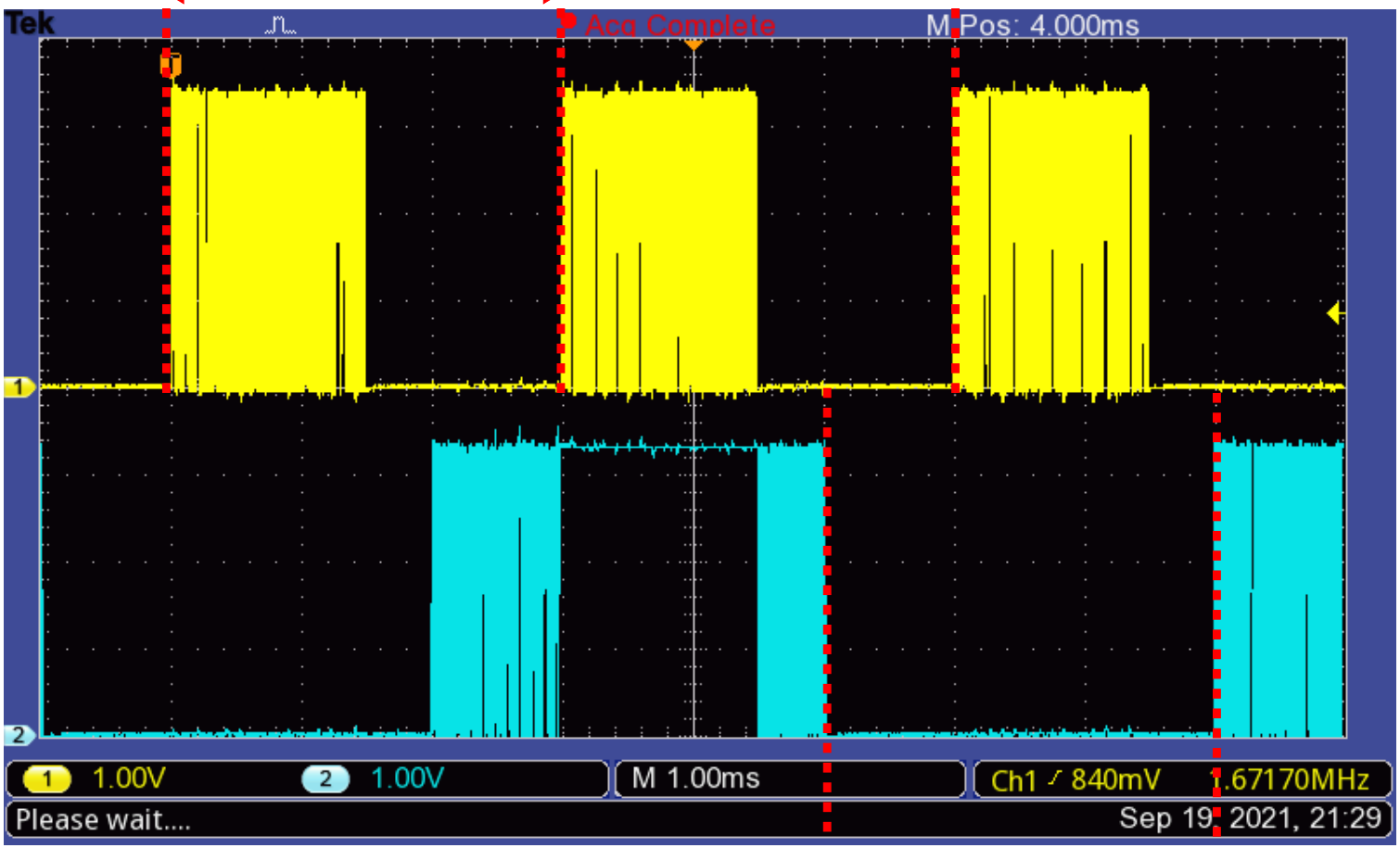
```
xTaskCreate(vTask1, "Task1", 10000, NULL, 10, &Task1);
```

- Tâche 2 : Tache avec un retard de 3ms

```
xTaskCreate(vTask2, "Task2", 10000, NULL, 9, &Task2,1);
```

Algorithme d'ordonnancement : Illustration

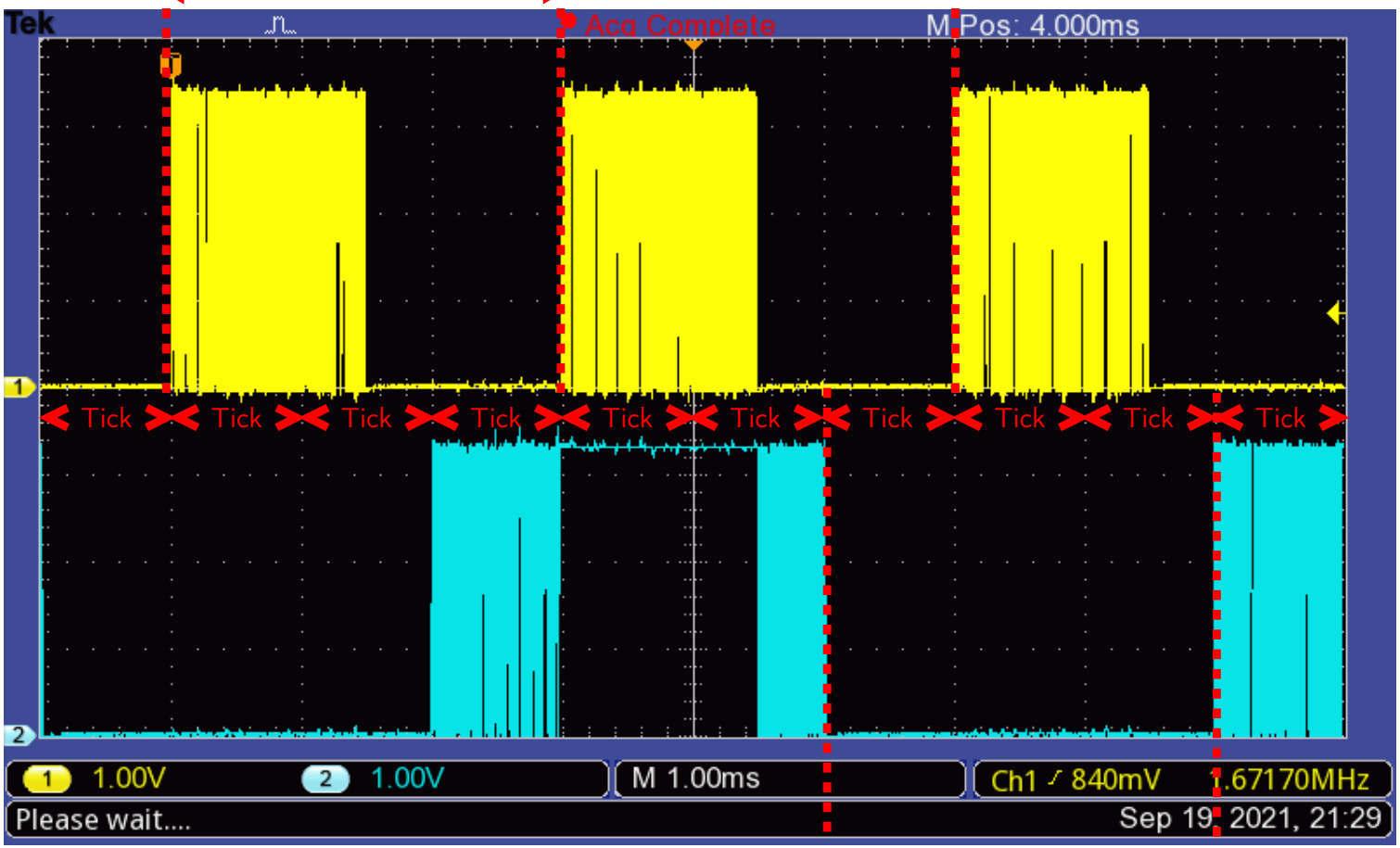
```
vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS( 3 ) );
```




```
vTaskDelay( pdMS_TO_TICKS( 3 ) );
```

Algorithme d'ordonnancement : Illustration

```
vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS( 3 ) );
```

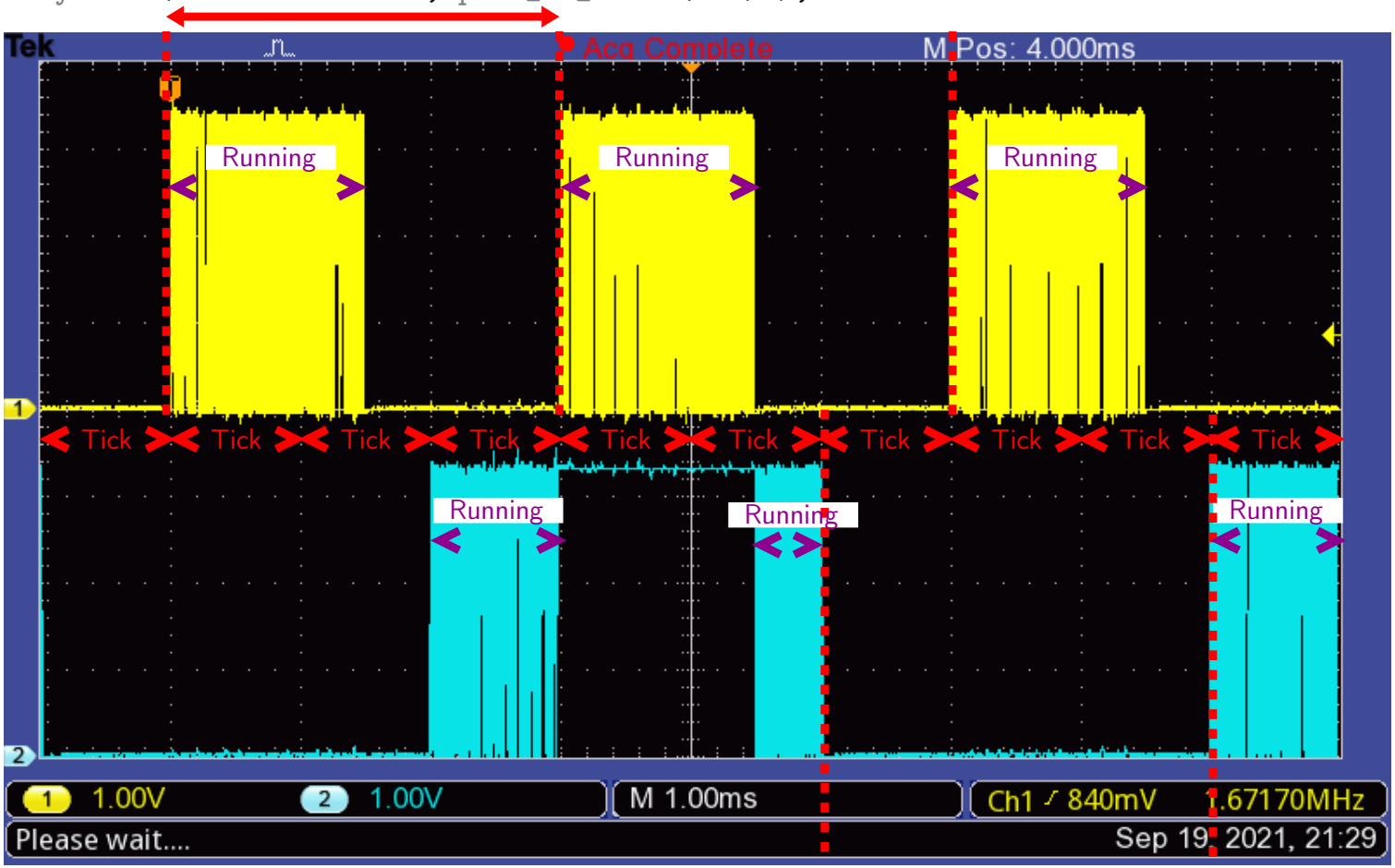



```
vTaskDelay( pdMS_TO_TICKS( 3 ) );
```

Real Time Operating System (RTOS)

Algorithme d'ordonnancement : Illustration

```
vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS( 3 ) );
```

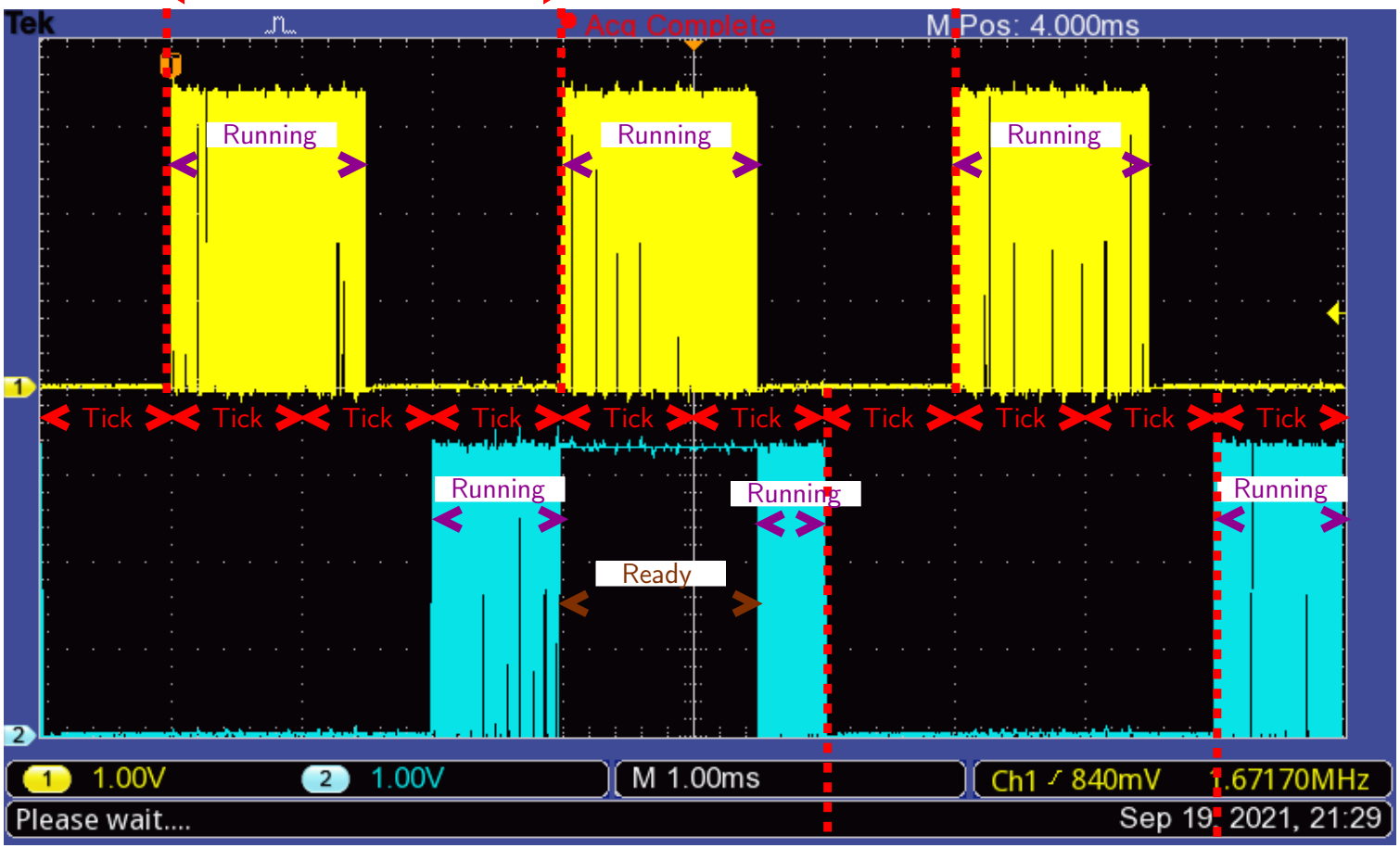



```
vTaskDelay( pdMS_TO_TICKS( 3 ) );
```

Real Time Operating System (RTOS)

Algorithme d'ordonnancement : Illustration

```
vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS( 3 ) );
```

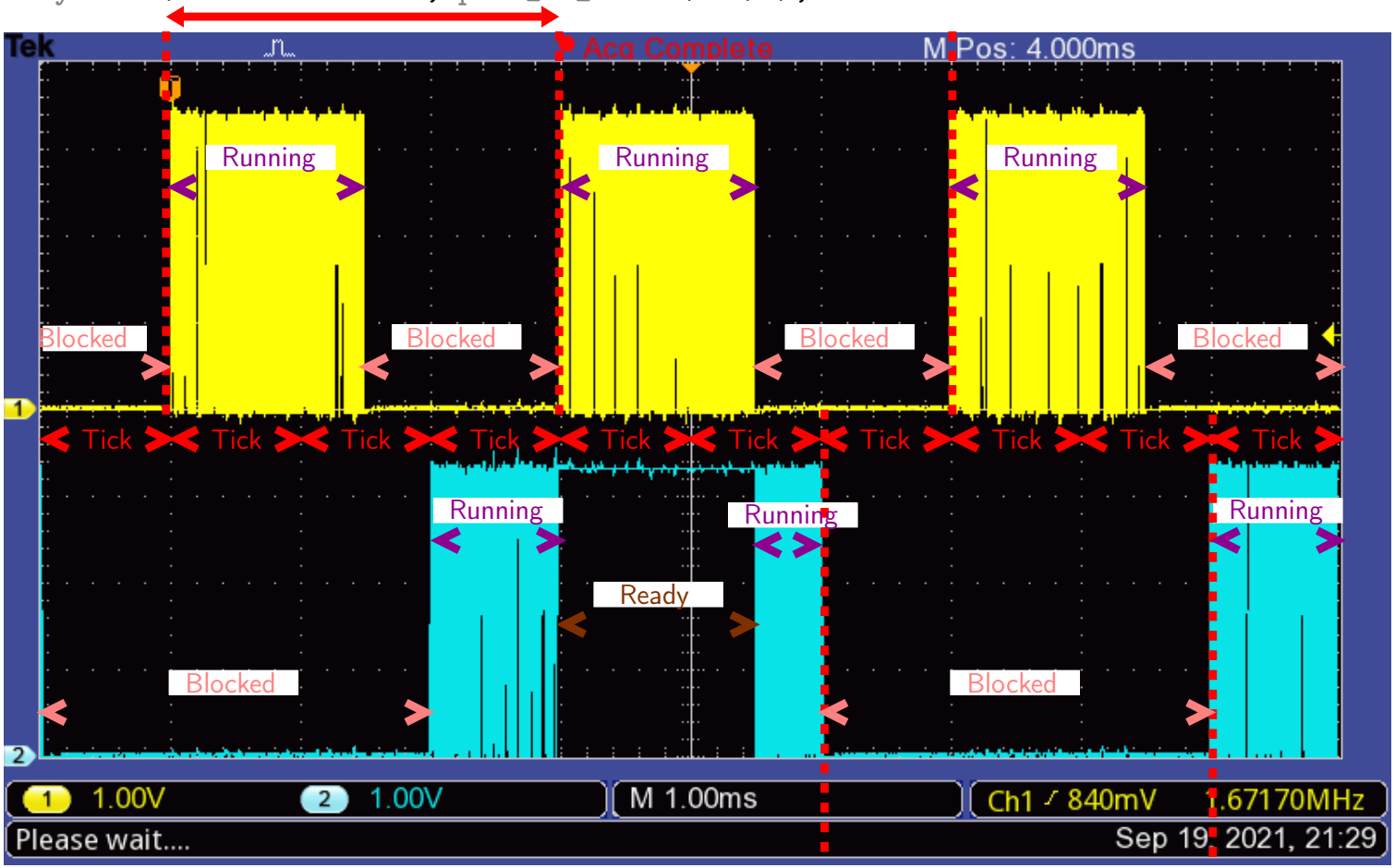



```
vTaskDelay( pdMS_TO_TICKS( 3 ) );
```

Real Time Operating System (RTOS)

Algorithme d'ordonnancement : Illustration

```
vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS( 3 ) );
```




```
vTaskDelay( pdMS_TO_TICKS( 3 ) );
```

Real Time Operating System (RTOS)

Pour aller plus loin ...

- Communication entre tâches
- Synchronisation entre tâches
- Les interruptions

Notions vues en TP

Sommaire

Conclusion

Aperçu du fonctionnement du noyau FreeRTOS

À retenir

- Notion de tâches (Running, Ready, Blocked, Suspended)
- Algorithme de Round Robin
- Pre-emptive Scheduling with Time Slicing
- Priorités

Sources

- FreeRTOS
- Richard Barry - Mastering the FreeRTOS Real Time Kernel
- ESP-IDF Programming Guide
- Warren Gay - FreeRTOS for ESP32-Arduino - Elektor
- Thierry Vaira, ESP32 + FreeRTOS
- FreeRTOS - Alberto Bosio - Université de Montpellier
- Systèmes temps réel - Hugo Descoubes - ENSICAEN