5TC option AUD

Embedded Programming Basics

Romain Michon, Tanguy Risset

Labo CITI, INSA de Lyon, Dpt Télécom

GRAME-CNCM INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

GRAME CENTRE NATIONAL DE CRÉATION MUSICALE, LYON

1er novembre 2022

OS embarqués légers

- Catégories des systèmes
- Modèles de programmation
- Evénements
- Evénements : TinyOS
- Coroutines
- Contiki et protothread
- Direct threading
- Protothread Adam Dunkel
- Protothread Adam Dunkel

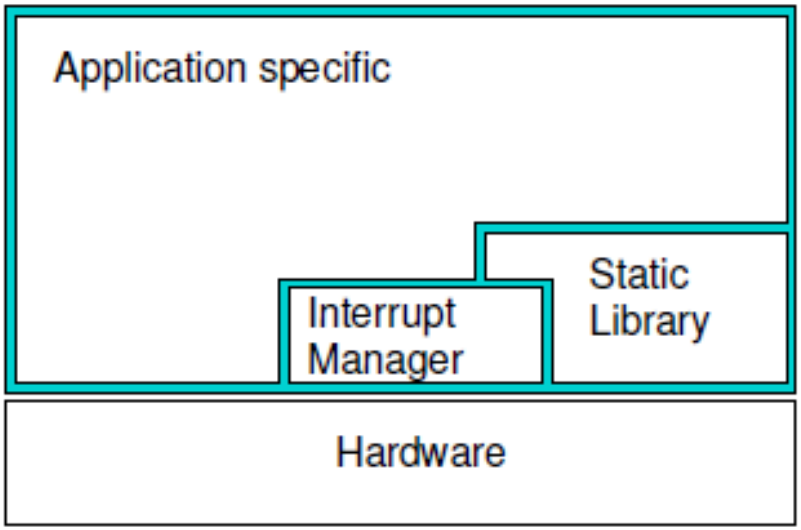
FreeRTOS

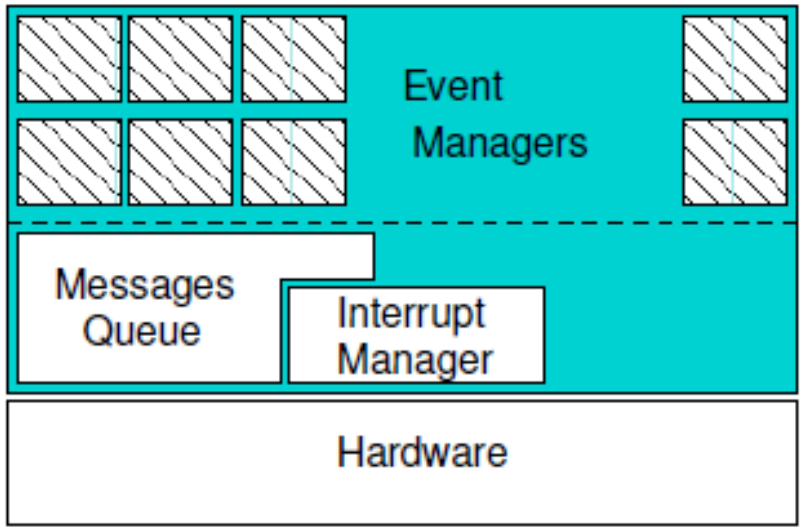
- Modèle à Thread : FreeRTOS
- Free RTOS in IDF
- Communicating between tasks : xQueue

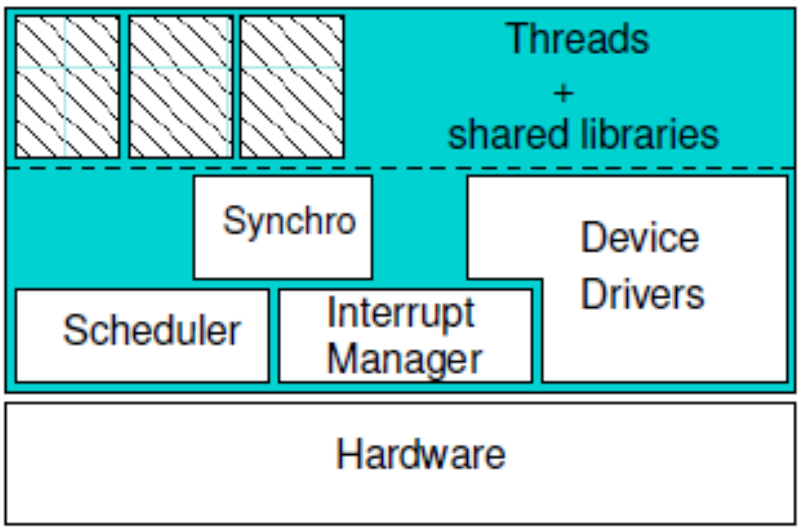
Systèmes d'exploitation légers

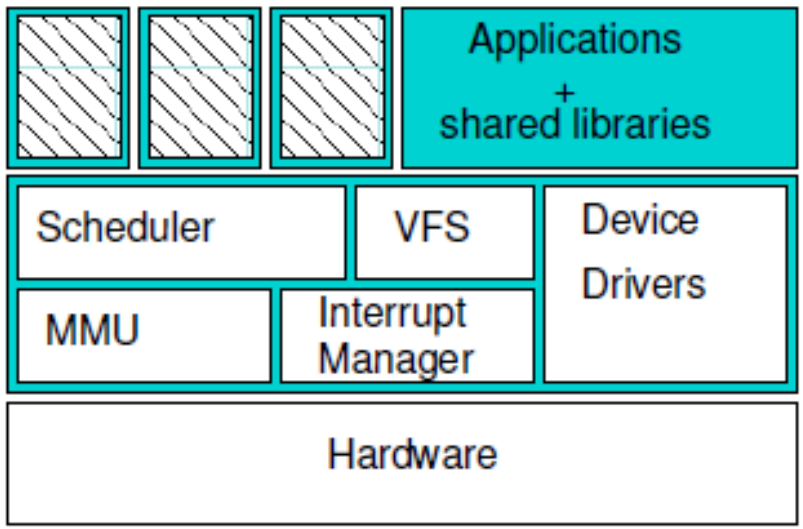
- Les systèmes d'exploitation peuvent aller d'une bibliothèque spécifique pour une application à un système générique type Unix.
- Les applications sans système d'exploitation représentent une part importante des systèmes déployés aujourd'hui.
- Il existe tout de même deux grandes catégories de système
  - modèle "Event driven"
  - modèle "Thread"

Catégories des systèmes









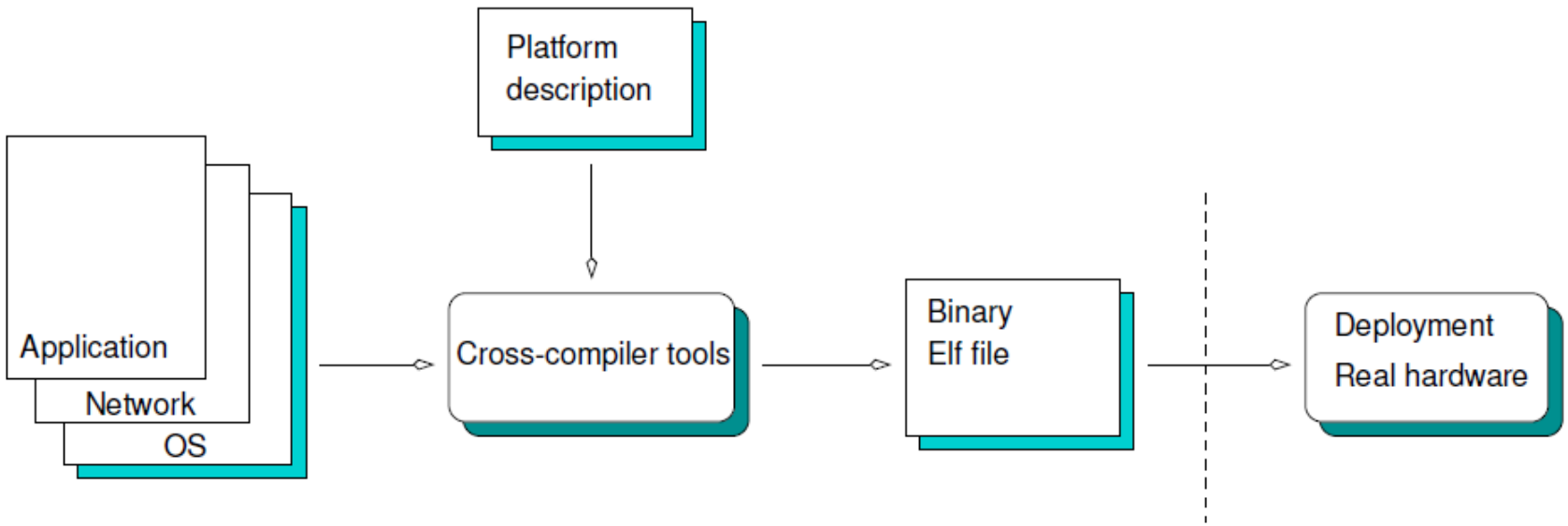
Modèles de programmation et d'exécution

- Evénements :
  - Les événements matériels démarrent des fonctions qui s'exécutent sans interruption (*run to completion*).
  - Les changements de contexte, la gestion de pile, l'ordonnancement et la gestion de priorité sont simplifiés.
  - Exemples : TinyOS 1 & 2

- File de programme / *Thread* :
  - Proche du modèle de programmation classique.
  - Mémoire partagée, piles séparées.
  - Changement de contexte.
  - Exemples : FreeRTOS, Contiki

Environnement logiciel

Les applications sont souvent simples. Les deux modèles sont fait pour être liés statiquement au programme et embarqués dans le système.



Pourquoi utiliser un OS ?

Quels services demander à un OS ?

- Gestion de Tâches/Files = ordonnanceur
- Pilotes de périphériques = interface matériel
  - Gestionnaire d'interruption
  - Gestion du temps et des timers

- Gestion des modes de veille
- Pile réseau intégrée

- Environnement de programmation et outils

Aperçu des systèmes

3 exemples de systèmes utilisés dans les petits objets.

1. TinyOS : modèle à événements
2. Contiki : modèle à protothread (Anciennement Co-routine)
3. FreeRTOS : modèle à thread

Événements : exemple TinyOS

- Gestion des événements
- Fréquences fixes, mode basse conso simple
- Propose des abstractions pour
  - les communications
  - les timers
  - le stockage

- Modèle d'exécution : run to completion
- **Utilisation d'une seule pile d'exécution**

- TinyOS 2.x légère amélioration du système de mise en veille

TinyOS 1.x main loop (1/2)

```
int main(void)
{
  MainM$hardwareInit();
  TOSH_sched_init();
  MainM$StdControl$init();
  MainM$StdControl$start();
  __nesc_enable_interrupt();
  for (; ; ) {
    TOSH_run_task();
  }
}
```

TinyOS 1.x main loop (2/2)

```
bool TOSH_run_task(void)
{
  void (*func)(void );
  __nesc_atomic_t fInterruptFlags = __nesc_atomic_start();
  uint8_t                  old_full = TOSH_sched_full;
  func = TOSH_queue[old_full].tp;
  if (func == NULL) {
      __nesc_atomic_sleep();
      return 0;
  }
  TOSH_queue[old_full].tp = NULL;
  TOSH_sched_full = (old_full + 1) & TOSH_TASK_BITMASK;
  __nesc_atomic_end(fInterruptFlags);
  func();
  return 1;
}
```

Coroutines / Protothread : exemple Contiki

Coroutines

- Multi-tâche coopératif.
- L'application reste maître de l'ordonnancement.

Protothread

- Modèle très proche des coroutines
- Modèle mixte, orienté événements
  - "run to completion"
  - **Changement de fil sur opération bloquante**
  - **Pile d'exécution unique**
  - Les "thread" n'ont pas d'état (variables locales)

Coroutines / Protothread : exemple Contiki

Coroutines

- Multi-tâche coopératif.
- L'application reste maître de l'ordonnancement.

Protothread

- Modèle très proche des coroutines
- Modèle mixte, orienté événements
  - "run to completion"
  - **Changement de fil sur opération bloquante**
  - **Pile d'exécution unique**
  - Les "thread" n'ont pas d'état (variables locales)

Principe des Coroutines utilisées Contiki

- Chaque *thread* va *rendre la main* à interval régulier lors de son execution, pou permettre l'execution des autres thread (on appelle quelquefois ça *yield*)
- En général quand le thread se met en attente sur une condition
- Comme il n'y a qu'une pile, lors de sa reprise le thread aura probablement perdu l'état de ses variables, donc ce sont des thread sans état.
- Problème : on sait comment sortir d'une fonction à n'importe quelle ligne (`return`), mais comment *reprendre* une fonction à n'importe quelle ligne ?

Contiki 2.x protothread example

```
#define PT_WAIT_UNTIL(pt, condition)    \
  do {                                  \
    LC_SET((pt)->lc);                   \
    if(!(condition)) {                  \
      return PT_WAITING;                \
    }                                   \
  } while(0)
static PT_THREAD(thread_periodic_send(struct pt *pt))  {
    PT_BEGIN(pt);
    while(1)  {
        TIMER_RADIO_SEND = 0;
        PT_WAIT_UNTIL(pt, node_id != NODE_ID_UNDEFINED &&
                  timer_reached( TIMER_RADIO_SEND, 1000));
        send_temperature();
    }
    PT_END(pt);
}
```

protothread definition : macro in .h file

```
#define PT_BEGIN(pt) { char PT_YIELD_FLAG = 1; \
        LC_RESUME((pt)->lc)

#define LC_RESUME(s) switch(s) { case 0:

#define LC_SET(s) s = __LINE__; case __LINE__:

#define LC_END(s) }
```

Machine de Duff (1/2)

En 1984 Tom Duff travaillant pour LucasFilm cherche à accélérer le code suivant :

```
send(int *to, int *from, int count)
        {
                do
                        *to = *from++;
                while(--count>0);
        }
```

Machine de Duff (1bis/2)

Technique classique : dérouler la boucle

```
send(int *to, int *from, int count)
{
        register n=(count+7)/8;
        do{
          *to++ = *from++;
          *to++ = *from++;
          *to++ = *from++;
          *to++ = *from++;
          *to++ = *from++;
          *to++ = *from++;
          *to++ = *from++;
          *to++ = *from++;
        }while(--n>0);
    }
}
```

Pb : la taille du tableau doit être un multiple de 8...

Machine de Duff (2/2) : la solution

```
send(int *to, int *from, int count)
{
    register n=(count+7)/8;
    switch(count%8){
        case 0: do{ *to++ = *from++;
        case 7:  *to++ = *from++;
        case 6:  *to++ = *from++;
        case 5:  *to++ = *from++;
        case 4:  *to++ = *from++;
        case 3:  *to++ = *from++;
        case 2:  *to++ = *from++;
        case 1:  *to++ = *from++;
                  }while(--n>0);
    }
}
```

http://www.lysator.liu.se/c/duffs-device.html

Protothread Adam Dunkel : lc-switch.h

```
#define LC_INIT(s) s = 0;

#define LC_RESUME(s) switch(s) { case 0:

#define LC_SET(s) s = __LINE__; case __LINE__:

#define LC_END(s) }
```

Protothread Adam Dunkel : pt.h

```
#include "lc.h"

typedef unsigned short lc_t;
struct pt {
  lc_t lc;
};
#define PT_INIT(pt)   LC_INIT((pt)->lc)
#define PT_THREAD(name_args) char name_args
#define PT_BEGIN(pt) { char PT_YIELD_FLAG = 1;
LC_RESUME((pt)->lc)

#define PT_WAIT_UNTIL(pt, condition)          \
  do {       \
    LC_SET((pt)->lc);     \
    if(!(condition)) {     \
      return PT_WAITING;    \
    }       \
  } while(0)

#define PT_END(pt) LC_END((pt)->lc); PT_YIELD_FLAG = 0; \
                   PT_INIT(pt); return PT_ENDED; }
```

How FreeRTOS help us

- Task Management
- Inter-Task Communication (Message Queues)
- Inter-Task synchronization (Semaphores)

Thread et Préemption : exemple FreeRTOS 4.x

- Opérations de base :
  - Gestion de tâches
  - **Ordonnancement préemptif**
  - Timers & Synchronisation (mutex)

- Utilisation de priorités
- Ordonnancement préemptif
- Primitives de synchronisation
- **Piles séparées par thread**
- Tâche “idle” de plus faible priorité

- Pas de pilote de périphérique

FreeRTOS 4.x main loop

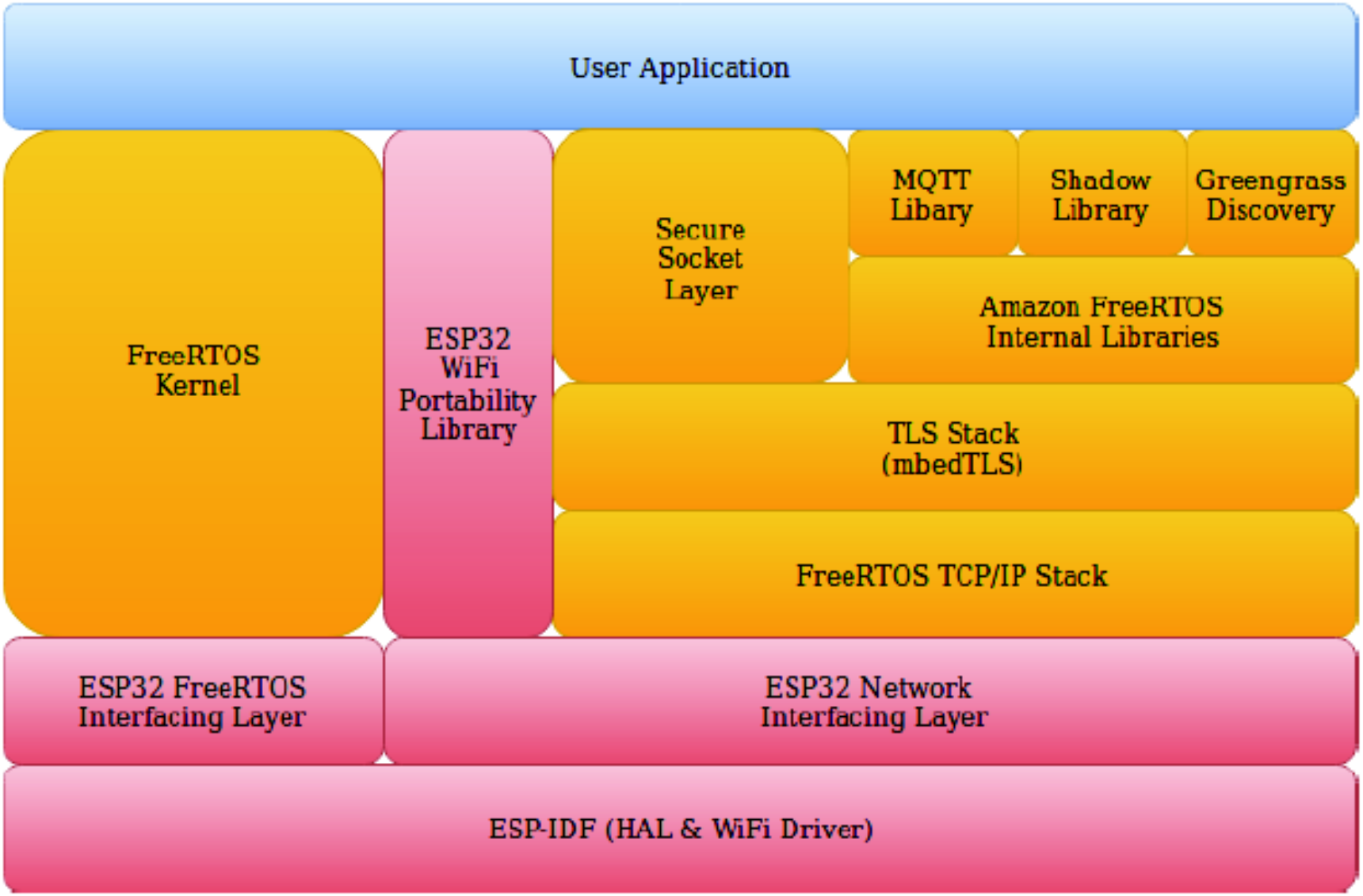
```
portTASK_FUNCTION(task_periodic_send, pvParameters) {
    const portTickType xDelay = 1000 / portTICK_RATE_MS;
    for(;;) {
        send_temperature();
        vTaskDelay(xDelay);
    }
}

int main( void ) {
  prvSetupHardware();
  vParTestInitialise();   // start Idle Task
  xTaskCreate(task_periodic_send, "RADIO", STACK_SIZE,
        & ParameterToPass, TASK_PRIORITY, &task_handle );
  vTaskStartScheduler(); // never returns
  return 0;
}
```

FreeRTOS characteristics

- Scheduler usually configured to allow **preemption**
- Multi-tasking with separate stacks and configurable stack size
- Internal clock management (“Ticks”)
- Task Priority and Round Robin management for identical priority
- timeout on blocking tasks

FreeRTOS integration in IDF



Task in Free RTOS

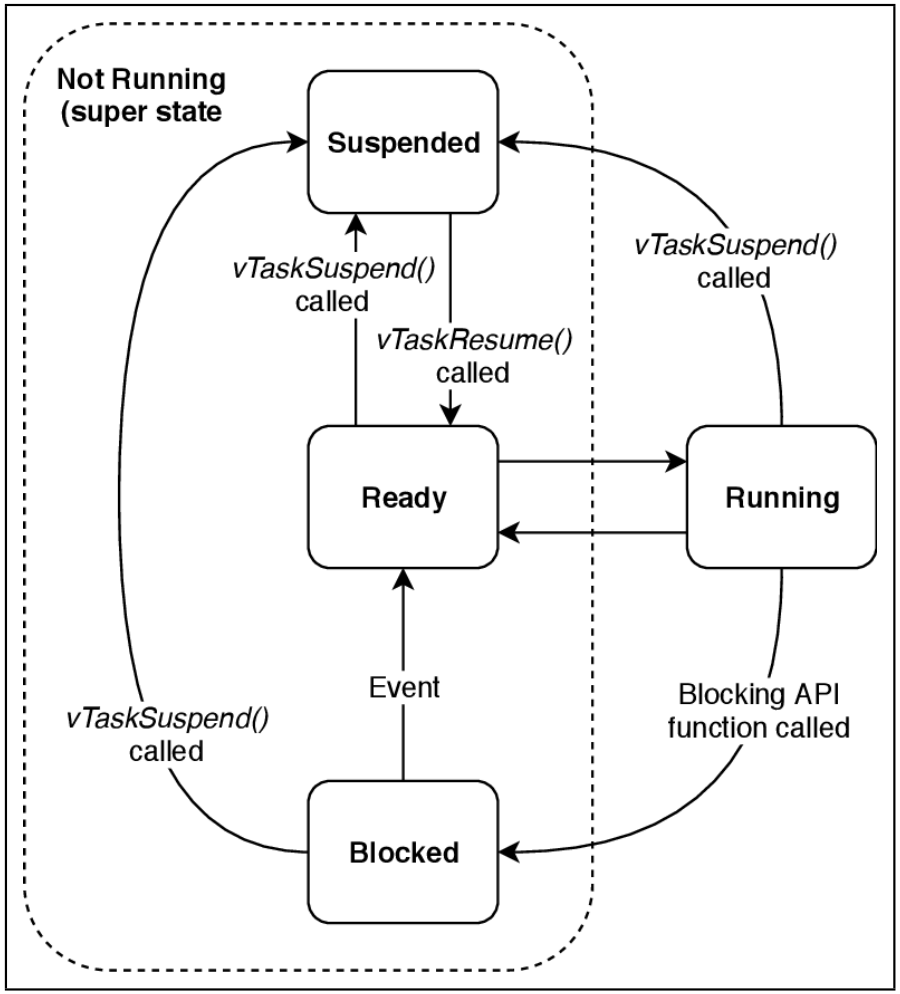
- Tasks are created by the user
- then they are managed (transparently) by the scheduler.
- The scheduler is launched by the used (but on LyraT, IDF is hanfdling the scheduler for you).

FreeRTOS tasks structure

```
void vATaskFunction( void *pvParameters )
  {
   -- Task application initialisation code here. --
   for( ;; )
      { -- Task application code here. -- }
  }
```

- A task “function” returns void and takes a `void *` parameter
- A task should never end (infinite loop)
- Task are created by a call to `xTaskCreate()` and deleted with `vTaskDelete()`
- Task termination is handled by IDF

FreeRTOS tasks states



A task can be

- Running – Actively executing and using the processor
- Ready – Able to execute, but not because a task of equal or higher priority is in the Running state.
- Blocked – Waiting for a temporal or external event. E.g., queue and semaphore events, or calling vTaskDelay() to block until delay period has expired. Always have a “timeout” period, after which the task is unblocked.
- Suspended – Only enter via vTaskSuspend(), depart via xTaskResume() API calls. (handled by IDF)

The Idle Task

- The idle task is created automatically when the scheduler is started (handled by IDF),
- Make sure that it is not starve, i.e. always exists other tasks
- It has the lowest priority : `tskIDLE_PRIORITY` (usually 0).

xTaskCreate

```
BaseType_t xTaskCreate(
    TaskFunction_t pvTaskCode,
    const char * const pcName,
    uint16_t usStackDepth,
    void *pvParameters,
    UBaseType_t uxPriority,
    TaskHandle_t *pxCreatedTask );
```

xTaskCreate() takes the following parameters.

- A pointer to the function that implements the task (type pdTASK_CODE).
- A name for the task.
- The depth of the task's stack.
- The task's priority.
- A pointer to any parameters needed by the task's function.

example :

```
xTaskCreate(myTaskFunction, "MyName", MY_STACK_SIZE,
      & ParameterToPass, MY_PRIORITY, &task_handle );
```

Queues in FreeRTOS

- FreeRTOS proposes queues (`xQueue`) as main form of inter-task communications.
- Queues are used to send messages between tasks **or** between ISR and tasks.
- Queues are created to contain a given type of item (given size actually)
- The size of the Queue and the size of its item is fixed at queue creation
- Item are enqueued by copy (not by reference)

xQueueCreate

```
QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength,
                            UBaseType_t uxItemSize );
```

- Queues are referenced by handles, which are variables of type `QueueHandle_t`.
- FreeRTOS allocates RAM from the FreeRTOS heap when a queue is created.
- `xQueueCreate()` will return NULL if there is insufficient heap RAM available for the queue to be created.

example of xQueue use (1), from F. Jumel

```
struct AMessage {
   portCHAR ucMessageID;
   portCHAR ucData[ 20 ]; };

void vATask( void *pvParameters ) {
  xQueueHandle xQueue1, xQueue2;
  // Create a queue capable of containing 10 unsigned long va
  xQueue1 = xQueueCreate( 10, sizeof( unsigned portLONG ) );

  if( xQueue1 == NULL ) {
    // Queue was not created and must not be used.
  }

  // These seconf queue shows how to pass message "by referen
  xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );
  if( xQueue2 == NULL ) {
    // Queue was not created and must not be used.
  }
```

example of xQueue use (2), from F. Jumel

```
unsigned portLONG uIVar;
if( xQueueSend( xQueue1, ( void * ) &ulVar,
                ( portTickType ) 10 )
    != pdPASS )
{
  // Failed to post the message, even after 10 ticks.
}

// Send a pointer to a struct AMessage object.
// Don't block if the queue is already full.
pxMessage = &xMessage;
xQueueSend( xQueue2, ( void * ) &pxMessage,
            ( portTickType ) 0 );
// ... Rest of task code
```

freeRTOS Naming convention

- Variables
  - Le type void préfixé par v
  - Le type char préfixé par c
  - Le type short est préfixé par s
  - Le type long est préfixé par l
  - Le type float est préfixé par f
  - Le type double est préfixé par d
  - Les variables d'énumération sont préfixées par e
  - Les autres types (par exemple les structures) sont préfixées par x
  - Les Pointeurs have ont un préfixes additionnels p,
  - Les variables non signées ont un préfixe additionnel u.

- Fonctions
  - Les fonctions privées (a priori interne au code du système d'exploitation) sont
  - préfixées par prv
  - Les fonctions de l'API du système d'exploitation sont préfixées par leur type de retour
  - Les fonctions commencent par le nom du fichier qui les contient. Par exemple vTaskDelete est définie dans Task.c et renvoie un void

Sources

Sources for these slides :

- Fabrice Jumel old slides on FreeRTOS for the SETRE course.
- Christopher Kenna slides on FreeRTOS
- FreeRTOS official documentation : “Mastering the FreeRTOS” (`http://www.FreeRTOS.org`)