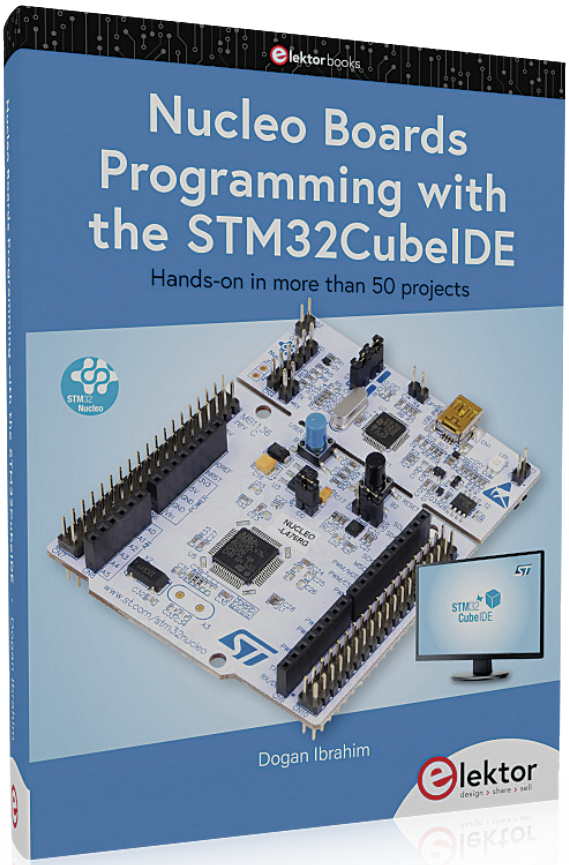
programmation des cartes Nucleo avec STM32CubeIDE

Extrait : FreeRTOS pour le MCU STM32



Dogan Ibrahim (Royaume-Uni)

Cet article présente une annexe extraite du livre *Nucleo Boards Programming with the STM32CubeIDE* de Dogan Ibrahim, publié par Elektor. Au cœur des projets décrits dans le livre se trouvent la carte de développement *Nucleo-L476RG*, bon marché, et le logiciel gratuit *STM32CubeIDE*. Ils forment un couple idéal pour des applications embarquées assez avancées. Pour cette courte initiation, le *STM32*, *CubeIDE* et *FreeRTOS* unissent leurs forces.

Note de l'éditeur : cet article est un extrait du livre *Nucleo Boards Programming with the STM32CubeIDE – Hands-on in More Than 50 Projects* formaté et légèrement modifié pour correspondre aux normes éditoriales et à la mise en page du magazine Elektor. Puisque cet article est extrait d'une publication plus vaste, certains termes peuvent faire référence à des passages du livre d'origine situés ailleurs. L'auteur et l'éditeur ont fait de leur mieux pour l'éviter et seront heureux de répondre aux questions – Pour les contacter, voir l'encadré « **Des questions, des commentaires ?** ».

La plupart des systèmes en temps réel (RT) complexes traitent quasi simultanément un certain nombre de tâches (ou programmes). Prenons l'exemple d'un système RT tout simple qui doit faire clignoter une LED à intervalle fixe, et en même temps, surveiller les touches d'un clavier. Une solution serait d'écrire une tâche qui à la fois scrute le clavier en boucle à intervalle fixe et fait clignoter la LED. Bien que cette approche puisse fonctionner pour un cas simple, dans la plupart des systèmes RT complexes, il faut mettre en œuvre une approche *multitâche*.

Le multitâche consiste à traiter plusieurs tâches (ou programmes) en parallèle avec la même unité centrale (UC). Cependant, il n'est pas possible que plusieurs tâches s'exécutent en même temps sur une seule UC. On procède en fait à une commutation des tâches en partageant le temps d'utilisation de l'UC. Dans une application, les tâches sont souvent interdépendantes : il faut instaurer une forme de coopération. Par ex., l'exécution d'une tâche peut dépendre de l'achèvement d'une autre, ou une tâche peut avoir besoin de données d'une autre. Si c'est le cas, les tâches concernées doivent être synchronisées à l'aide d'une méthode de communication entre tâches.

Les systèmes RT sont sensibles au temps et leur UC n'est jamais surchargée. Dans ces systèmes, en général, les tâches obéissent strictement à des priorités. Une tâche ayant une priorité plus élevée peut s'emparer de l'UC utilisée par une tâche moins prioritaire et en avoir l'usage exclusif jusqu'à ce qu'elle la libère. Si la tâche de priorité supérieure a terminé son traitement ou attend qu'une ressource se libère, la tâche de priorité inférieure peut s'emparer de l'UC et reprendre le traitement au point où il a été interrompu.

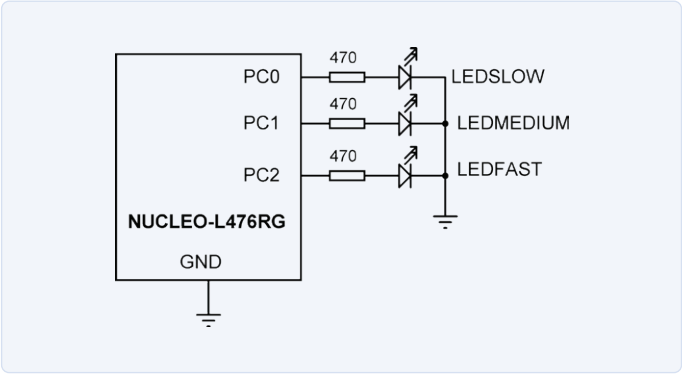


Figure 1. Schéma du circuit d'exemple pour exécuter Free RTOS sur MCU STM32.

Les systèmes en temps réel doivent aussi réagir aux événements le plus vite possible. Les événements externes sont en général traités à l'aide d'interruptions externes et la latence d'interruption doit être très courte afin que la routine de service d'interruption soit exécutée dès qu'une interruption se produit.
Dresser une liste des avantages attribués au noyau multitâche est aisé :

- Sans noyau multitâche, plusieurs tâches peuvent être exécutées en boucle, mais cette approche donne lieu à des performances en temps réel très mal contrôlées où les temps d'exécution des tâches ne peuvent être maîtrisés.
- On peut coder les différentes tâches comme des routines de service d'interruption. En pratique, cela peut fonctionner, mais si l'application comporte de nombreuses tâches, le nombre d'interruptions augmente et le code est moins gérable.
- Un noyau multitâche permet d'ajouter de nouvelles tâches ou de supprimer certaines des tâches existantes sans aucune difficulté.
- Le test et le débogage d'un système multitâche à noyau multitâche sont plus faciles à réaliser qu'avec un système multitâche sans noyau.
- La mémoire est mieux gérée avec un noyau multitâche.
- Un noyau multitâche facilite la gestion de la communication intertâches.
- Un noyau multitâche facilite le contrôle de la synchronisation des tâches.
- Un noyau multitâche facilite la gestion du temps de l'UC.
- En général, un noyau multitâche protège la mémoire en empêchant une tâche d'accéder à l'espace mémoire d'une autre.
- En général, un noyau multitâche offre la possibilité de gérer la priorité des tâches, les plus prioritaires peuvent accaparer l'UC et arrêter l'exécution des moins prioritaires. Les tâches importantes s'exécutent quand c'est nécessaire.

La nécessité d'un RTOS

Un RTOS (*Real-Time Operating System* ou système d'exploitation en temps réel) est un programme de gestion des ressources du système, il ordonnance l'exécution des tâches en son sein et en synchronise l'exécution, gère l'allocation des ressources et assure la communication et la messagerie entre les tâches. Tout RTOS comprend un noyau qui fournit les fonctions de bas niveau, principalement l'ordonnancement et la création de tâches, la communication intertâches, la gestion des ressources, etc.
Les RTOS complexes fournissent aussi d'autres services : gestion de fichiers, opérations de lecture-écriture sur disque, interruptions, gestion de réseau, gestion des utilisateurs, etc.

Dans un RTOS, une tâche est un fil d'exécution indépendant, avec son jeu local de données. Un RTOS comprend plusieurs tâches, chacune exécutant son propre code. Elles communiquent et se synchronisent entre elles pour avoir accès aux ressources partagées. L'exemple le plus simple d'un RTOS est celui où il y a par ex. trois LED et où chaque LED clignote à son propre rythme. La programmation d'un système aussi simple sans noyau multitâche pourrait être ardue. Nous verrons ici comment un RTOS tel que FreeRTOS peut être utilisé pour partager les ressources de l'UC.
Le RTOS le plus simple est un ordonnanceur qui détermine l'ordre d'exécution des tâches en son sein. Chaque tâche possède son contexte propre, constitué de l'état de l'UC et de ses registres associés. L'ordonnanceur passe d'une tâche à une autre en commutant le contexte : celui de la tâche en cours d'exécution est stocké et le contexte de la tâche suivante est chargé de sorte que l'exécution de celle-ci puisse reprendre correctement là où elle en était. Le temps que met l'UC pour changer de contexte s'appelle *temps de commutation de contexte* et est en général négligeable par rapport au temps d'exécution réel des tâches.

FreeRTOS

FreeRTOS est un RTOS qui fonctionne sur de nombreux microcontrôleurs haut de gamme, dont la famille STM32. STM32CubeIDE inclut le logiciel FreeRTOS, mais nous ne ferons pas d'appel à FreeRTOS directement. ARM a créé la bibliothèque CMSIS-RTOS, qui permet de faire des appels à un RTOS sous-jacent tel que FreeRTOS, c.-à-d. que nous ferons des appels à CMSIS-RTOS (v. 2) pour commander le FreeRTOS sous-jacent.
Vaste sujet que FreeRTOS et le multitâche : il faudrait plusieurs livres pour en expliquer toutes les fonctions. Les lecteurs qui ne connaissent pas les principes du multitâche trouveront sur l'internet des livres, tutoriels et notes d'application. Le livre *ARM-Based Microcontroller Multitasking Projects - Using the FreeRTOS Multitasking Kernel* facilitera l'assimilation des notions de multitâche et l'apprentissage de FreeRTOS dans les MCU basés sur ARM [1].

Projet FreeRTOS avec STM32MCubeIDE

Nous allons maintenant créer une application simple avec trois LED connectées à la carte de développement NUCLEO-L476RG. Les LED sont baptisées *LEDSLOW*, *LEDMEDIUM* et *LEDFAST*. Voici les connexions de ces LED et leur cadence de clignotement (voir aussi la fig. 1) :

LED	Cadence	Broche GPIO
LEDSLOW	1 s	PC0
LEDMEDIUM	500 ms	PC1
LEDFAST	250 ms	PC2

Un premier programme

Voici les étapes :

- Démarrer STM32CubeIDE.
- Choisir de démarrer une nouvelle application.
- Créer un nouvel espace de travail.
- Sélectionner le STM32L476RG comme processeur.
- Nommer le programme : *FREE*.
- Configurer *PC0*, *PC1*, et *PC2* comme sorties *GPIO_Output*. Faire un clic droit sur les broches et définir trois étiquettes utilisateur *User Labels* : *LEDSLOW*, *LEDMEDIUM* et *LEDFAST* (**fig. 2**).
- Configurer l'horloge du MCU à 80 MHz.
- Sur le côté gauche, cliquer sur *Middleware* et sélectionner *FreeRTOS*.
- Définir l'interface : *CMSIS_V2*.
- De nombreux paramètres peuvent être configurés sous l'onglet *Configuration*. Leur utilisation nécessite une bonne connaissance de FreeRTOS. Dans ce projet de démonstration, toutes les valeurs par défaut conviennent (voir **fig. 3**).
- Cliquer sur *File*, sur *Save* et enfin sur *YES* pour générer le code.
- Cliquer sur *Core*, *Src*, et double-cliquer sur *main c* pour afficher le programme principal.

Ce programme comporte trois tâches, aussi appelées fils (*threads*). Les fonctions de base du logiciel FreeRTOS sont les suivantes :

- définir les ID des fils ;
- définir les attributs des fils ;
- initialiser l'ordonnanceur (*scheduler*) ;
- créer les fils ;
- démarrer l'ordonnanceur.

`osThreadId_t` est utilisé pour définir les ID des fils. Les ID des fils de ce programme sont :

```
osThreadId_t LEDSTaskHandle; // LED lente
osThreadId_t LEDMTaskHandle; // LED moyenne
osThreadId_t LEDFTaskHandle; // LED rapide
```

Les attributs de fil définissent divers paramètres d'un fil, tels que nom, priorité, taille de pile, etc. Par ex., les attributs de fil pour la tâche lente sont :

```
const osThreadAttr_t LEDSTask_attributes =
{
.name = "LEDSTask",
.priority = (osPriority_t) osPriorityNormal,
.stack_size = 128 * 4
};
```

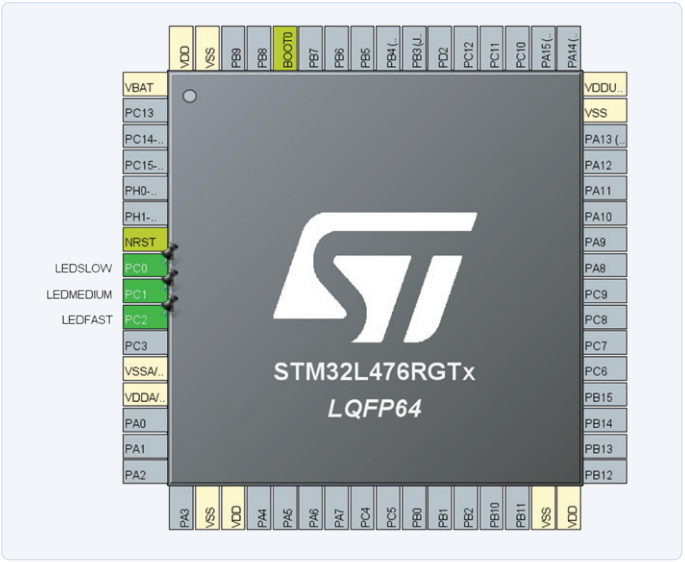


Figure 2. Configuration des sorties de l'UC STM32 dans STMCubeIDE.

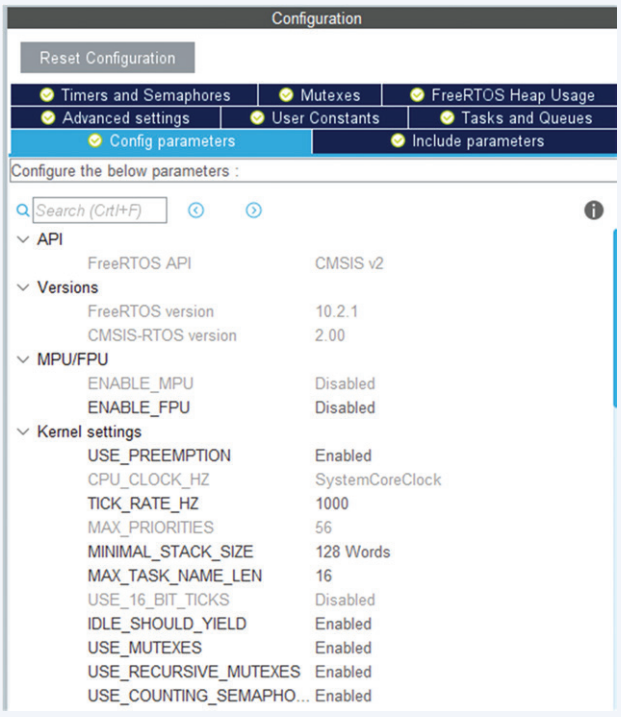


Figure 3. Pour les paramètres de configuration, accepter les valeurs par défaut.

LIENS

[1] **Livre en anglais de Dogan Ibrahim, « ARM-Based-Microcontroller-Multitasking-Projects - Using the FreeRTOS », Elektor, 2020**

[2] **Page du livre** : www.elektor.fr/nucleo-boards-programming-with-the-stm32cubeide

L'ordonnanceur est initialisé et commence par les appels de fonctions :

```
osKernelInitialize();
osKernelStart();
```

Les fils sont créés par la fonction `call osThreadNew()`. Par exemple, le fil pour la LED lente est créé comme suit :

```
LEDSTaskHandle = osThreadNew(StartLEDSTask, NULL,
   &LEDSTask_attributes);
```

Où `StartLEDSTask` est le nom de la fonction appelée par l'ordonnanceur pour réaliser le clignotement lent de la LED. Son contenu est :

```
void StartLEDSTask(void *argument)
{
for(;;)
{
HAL_GPIO_TogglePin(GPIOC, LEDSLOW_Pin);
osDelay(1000);
}
}
```

Noter qu'au lieu de `HAL_Delay()`, il faut utiliser `osDelay()`. Dans une tâche à haute priorité, `HAL_Delay()` pourrait empêcher un changement de contexte. Mais `osDelay()` indique à l'ordonnanceur de passer à une autre tâche pendant l'attente.
Compiler le programme en mode *Release* et glisser-déposer le fichier binaire *FREE.bin* sur le périphérique *NUCLEO-L476RG*. À ce stade, les trois fils sont exécutés simultanément dans leur propre boucle `sans fin`. L'ordonnanceur va commuter les fils pour donner l'impression qu'ils s'exécutent tous trois en même temps. Les trois LED doivent clignoter en même temps à des rythmes différents.

Le programme !

Le listage 1 montre le programme appelé *FREE*. Le programme peut être extrait de l'archive du logiciel libre (.zip) disponible sur la page web du livre [2]. Sur cette page, faire défiler vers le bas jusqu'à *Téléchargements* et prendre le fichier appelé *Software_Nucleo Boards Programming with the STM32CubeIDE*. Le décompresser sur un disque local, puis aller dans le dossier *FREE*.

(210236-04)

Des questions, des commentaires ?
Envoyez un courriel à l'auteur (d.ibrahim@btinternet.com) ou contactez Elektor (redaction@elektor.fr).

Contributions
Texte : **Dogan Ibrahim**
Rédaction : **Jan Buiting**
Mise en page : **Giel Dols**
Traduction : **Yves Georges**

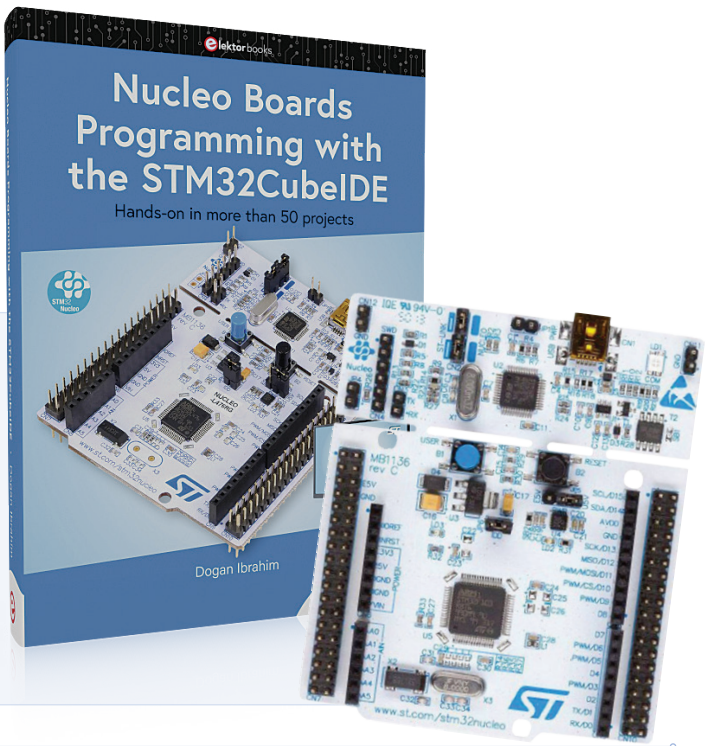
Listage 1 : programme FREE.

```
/* USER CODE BEGIN Header */
/**
******************************************************************************
* @file : main.c
* @brief : Main program body
******************************************************************************
* @attention
*
* Copyright (c) 2020 STMicroelectronics.
* All rights reserved.
*
* This software component is licensed by ST under Ultimate Liberty license
* SLA0044, the «License»; You may not use this file except in compliance with
* the License. You may obtain a copy of the License at:
* www.st.com/SLA0044
*
******************************************************************************
*/
#include «main.h»
#include «cmsis_os.h»
//
// Define Thread IDs
//
osThreadId_t LEDSTaskHandle;          // Slow LED
osThreadId_t LEDMTaskHandle;          // Medium LED
osThreadId_t LEDFTaskHandle;          // Fast LED
```

```
//
// Slow LED task. Flash every second
//
void StartLEDSTask(void *argument)
{
   for(;;)
   {
     HAL_GPIO_TogglePin(GPIOC, LEDSLOW_Pin);
     osDelay(1000);
   }
}
//
// Medium LED task. Flash every 500ms
//
void StartLEDMTask(void *argument)
{
   for(;;)
   {
     HAL_GPIO_TogglePin(GPIOC, LEDMEDIUM_Pin);
     osDelay(500);
   }
}

//
// Fast LED task. Flash every 250ms
//
void StartLEDFTask(void *argument)
{
   for(;;)
   {
     HAL_GPIO_TogglePin(GPIOC, LEDFAST_Pin);
     osDelay(250);
   }
}

void SystemClock_Config(void);
static void MX_GPIO_Init(void);
void StartDefaultTask(void *argument);
//
// Start of main program
//
int main(void)
{
   HAL_Init();
   SystemClock_Config();
   MX_GPIO_Init();
   //
   // Slow LED Task attributes
   //
   const osThreadAttr_t LEDSTask_attributes =
   {
      .name = «LEDSTask»,
      .priority = (osPriority_t) osPriorityNormal,
      .stack_size = 128 * 4
   };
   //
   // Medium LED Task attributes
   //
   const osThreadAttr_t LEDMTask_attributes =
   {
      .name = «LEDMTask»,
```

```
        .priority = (osPriority_t) osPriorityNormal,
        .stack_size = 128 * 4
    };
    //
    // Fast LED Task attributes
    //
    const osThreadAttr_t LEDFTask_attributes =
    {
        .name = «LEDFTask»,
        .priority = (osPriority_t) osPriorityNormal,
        .stack_size = 128 * 4
    };

    /* Init scheduler */
    osKernelInitialize();

    /* creation of Tasks */
    LEDSTaskHandle = osThreadNew(StartLEDSTask, NULL, &LEDSTask_attributes);
    LEDMTaskHandle = osThreadNew(StartLEDMTask, NULL, &LEDMTask_attributes);
    LEDFTaskHandle = osThreadNew(StartLEDFTask, NULL, &LEDFTask_attributes);

    /* Start scheduler */
    osKernelStart();

    while (1)
    {
    }
}

void SystemClock_Config(void)
{
    RCC_OscInitTypeDef RCC_OscInitStruct = ;
    RCC_ClkInitTypeDef RCC_ClkInitStruct = ;

    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
    RCC_OscInitStruct.HSIState = RCC_HSI_ON;
    RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
    RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
    RCC_OscInitStruct.PLL.PLLM = 2;
    RCC_OscInitStruct.PLL.PLLN = 20;
    RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
    RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
    RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
    if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
    {
      Error_Handler();
    }

    RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                                  |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
    RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
    RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
    RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
    RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
    {
      Error_Handler();
    }
```

```
  if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
   {
     Error_Handler();
   }
}

static void MX_GPIO_Init(void)
{
   GPIO_InitTypeDef GPIO_InitStruct = ;

  /* GPIO Ports Clock Enable */
   __HAL_RCC_GPIOC_CLK_ENABLE();
   /*Configure GPIO pin Output Level */
   HAL_GPIO_WritePin(GPIOC, LEDSLOW_Pin|LEDMEDIUM_Pin|LEDFAST_Pin, GPIO_PIN_RESET);

  /*Configure GPIO pins : LEDSLOW_Pin LEDMEDIUM_Pin LEDFAST_Pin */
   GPIO_InitStruct.Pin = LEDSLOW_Pin|LEDMEDIUM_Pin|LEDFAST_Pin;
   GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
   GPIO_InitStruct.Pull = GPIO_NOPULL;
   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
   HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
}

void Error_Handler(void)
{

}

#ifdef USE_FULL_ASSERT

void assert_failed(uint8_t *file, uint32_t line)
{

}
#endif
/************************ (C) COPYRIGHT STMicroelectronics *****END OF FILE****/
```



PRODUITS

- **Livre en anglais**
  « Nucleo Boards Programming with the STM32CubeIDE »

Version papier :
www.elektor.fr/nucleo-boards-programming-with-the-stm32cubeide
Version numérique :
www.elektor.fr/nucleo-boards-programming-with-the-stm32cubeide-e-book

- **Carte STM32 Nucleo L476RG**

www.elektor.fr/stm32-nucleo-l476rg-board