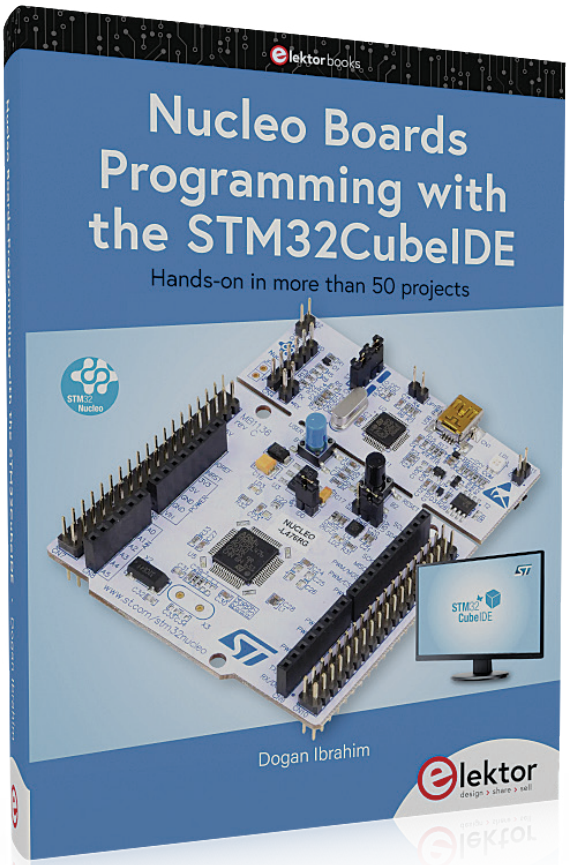
Nucleo-boards programmeren met de STM32CubeIDE

voorbeeldhoofdstuk: FreeRTOS voor de STM32-MCU



Dogan Ibrahim (Verenigd Koninkrijk)

In deze aflevering van Elektor Books presenteren we een appendix van Dogan Ibrahims boek *Nucleo Boards Programming with the STM32CubeIDE*, uitgegeven door Elektor. Het hart van alle in het boek beschreven projecten wordt gevormd door het Nucleo-L476RG-ontwikkelboard, dat alom voor weinig geld verkrijgbaar is, en de gratis STM32CubeIDE-software. Zij vormen een perfecte combinatie voor tamelijk geavanceerde embedded toepassingen. Hier laten we de STM32 microcontroller, de CubeIDE, en Free RTOS hun krachten bundelen in een doen&leren-project.

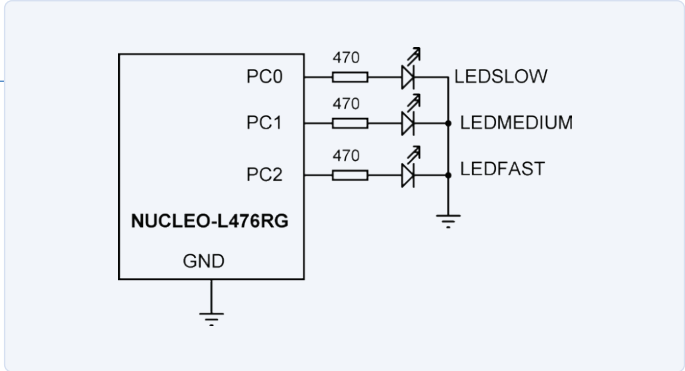
Van de redactie: Dit artikel is een gedeelte uit het boek *Nucleo Boards Programming with the STM32CubeIDE – Hands-on in More Than 50 Projects*, in geringe mate aangepast en met een layout conform de redactionele normen en pagina-indeling van Elektor. Omdat het een deel is uit een grotere publicatie, kunnen sommige termen in dit artikel verwijzen naar besprekingen elders in de oorspronkelijke boekpublicatie. Auteur en redactie hebben hun best gedaan om dit te vermijden, en zijn altijd bereid om te helpen met vragen – contactgegevens staan in het kader **Vragen of opmerkingen?**

De meeste complexe real-time systemen vereisen een aantal taken (of programma's) die bijna gelijktijdig moeten worden verwerkt. Neem bijvoorbeeld een uiterst eenvoudig real-time systeem dat een LED moet laten knipperen met een vereist interval, en tegelijkertijd moet controleren op toetsaanslagen. Een oplossing zou zijn om het toetsenbord in een lus met regelmatige tussenpozen te scannen en tegelijkertijd de LED te laten knipperen. Hoewel deze aanpak in een eenvoudig voorbeeld kan werken, moet in de meeste complexe real-time systemen een multitasking-aanpak worden geïmplementeerd.

De term multitasking betekent dat verschillende taken (of programma') parallel op dezelfde CPU worden verwerkt. Op één CPU kunnen echter niet tegelijkertijd verschillende taken draaien. Daarom wordt *taskswitching* toegepast, zodat taken CPU-tijd delen.

In veel toepassingen kunnen taken bovendien niet onafhankelijk van elkaar draaien en wordt van ze verwacht dat ze op de een of andere manier samenwerken. De uitvoering van een taak kan bijvoorbeeld afhangen van de voltooiing van een andere taak, of een taak kan gegevens van een andere taak nodig hebben. In dergelijke omstandigheden moeten de betrokken taken worden gesynchroniseerd met behulp van een of andere methode voor communicatie tussen taken.

Real-time systemen zijn tijd-responsieve systemen waarbij de CPU nooit overbelast is. In dergelijke systemen hebben taken gewoonlijk strikt in acht te nemen prioriteiten. Een taak met een hogere prioriteit kan de CPU afpakken van een taak met een lagere prioriteit en vervolgens de CPU exclusief gebruiken totdat de taak de CPU weer vrijgeeft. Als de taak met de hogere prioriteit klaar is met



Figuur 1. Schema van de voorbeeldtoepassing om Free RTOS op een STM32-microcontroller te draaien.

verwerken, of als hij wacht tot een systeembron beschikbaar komt, kan de taak met de lagere prioriteit de CPU pakken en verder gaan met verwerken vanaf het punt waar hij onderbroken werd. Van real time-systemen wordt ook verwacht dat ze zo snel mogelijk reageren op gebeurtenissen. Externe gebeurtenissen worden gewoonlijk verwerkt met externe interrupts en de interruptlatentie van dergelijke systemen moet zeer kort zijn, zodat de interrupt service-routine wordt uitgevoerd zodra een interrupt optreedt.
Het is niet moeilijk om een lijst te maken van de voordelen van de multitasking-kernel:

- Zonder een multitasking-kernel kunnen meerdere taken in een lus worden uitgevoerd, maar deze aanpak resulteert in zeer slechte real time-prestaties waarbij de uitvoeringstijden van de taken onvoorspelbaar zijn.
- Het is mogelijk om de verschillende taken te coderen als interrupt service-routines. In de praktijk kán dat werken, maar als de toepassing veel taken heeft, neemt het aantal interrupts toe waardoor de code minder beheersbaar wordt.
- Een multitasking-kernel maakt het mogelijk zonder problemen nieuwe taken toe te voegen of bestaande taken uit het systeem te verwijderen.
- Het testen en debuggen van een multitasking-systeem met een multitasking-kernel is gemakkelijk, vergeleken met een multitasking-systeem zonder kernel.
- Geheugen wordt beter beheerd met een multitasking-kernel.
- Communicatie tussen taken wordt gemakkelijk afgehandeld met een multitasking-kernel.
- Taaksynchronisatie kan eenvoudig worden geregeld met een multitasking-kernel.
- CPU-tijd kan gemakkelijk worden beheerd met een multitasking-kernel.
- De meeste multitasking-kernels bieden geheugenbeveiliging waarbij een taak geen toegang heeft tot de geheugenruimte van een andere taak.
- De meeste multitasking-kernels bieden taakprioriteit waarbij taken met een hogere prioriteit de CPU kunnen opeisen en de uitvoering van taken met een lagere prioriteit kunnen stoppen. Hierdoor kunnen belangrijke taken worden uitgevoerd wanneer dat nodig is.

De noodzaak van een RTOS

Een RTOS (*Real Time Operating System*) is een programma dat systeembronnen beheert, de uitvoering van verschillende taken in een systeem plant, de uitvoering van taken synchroniseert, de toewijzing van systeembronnen beheert, en zorgt voor communicatie en berichtenuitwisseling tussen taken. Elk RTOS bestaat uit een kernel die de low level-functies levert, voornamelijk scheduling, task creation, inter-task communication, resource management enzovoort.
De meeste complexe RTOS'sen bieden ook diensten voor file handling, harddisk lees/schrijf-operaties, interrupt servicing, netwerkbeheer, gebruikersbeheer enzovoort.
Een taak is een onafhankelijke uitvoeringsthread in een multitasking-systeem, gewoonlijk met zijn eigen lokale gegevensverzameling. Een multitasking-systeem bestaat uit verschillende taken, die elk hun eigen code uitvoeren, en die met elkaar communiceren en synchroniseren om toegang te hebben tot gedeelde systeembronnen. Het eenvoudigste voorbeeld van een multitasking-systeem is dat van 3 LED's die elk met een andere snelheid knipperen. Het programmeren van zo'n eenvoudig systeem zonder een multitasking-kernel zou ingewikkeld kunnen zijn. We zullen hier zien hoe een multitasking-besturingssysteem zoals FreeRTOS kan worden gebruikt om de systeembronnen van de MCU te delen.
Het eenvoudigste RTOS bestaat uit een scheduler die de uitvoeringsvolgorde van de taken in het systeem bepaalt. Elke taak heeft zijn eigen context, bestaande uit de toestand van de CPU en de bijbehorende registers. De scheduler schakelt van de ene taak naar de andere door een contextwisseling uit te voeren, waarbij de context van de lopende taak wordt opgeslagen en de context van de volgende taak op de juiste wijze wordt geladen, zodat de uitvoering van de volgende taak op correcte wijze kan worden voortgezet. De tijd die de CPU nodig heeft om van context te wisselen, staat bekend als de contextwisseltijd en is gewoonlijk verwaarloosbaar in vergelijking met de werkelijke uitvoeringstijd van de taken.

Het FreeRTOS

FreeRTOS is een real time operating system dat op veel high-end microcontrollers draait, waaronder de STM32-familie. STM32CubeIDE heeft FreeRTOS-software gebundeld, hoewel we FreeRTOS niet direct zullen aanroepen. ARM heeft de CMSIS-RTOS bibliotheek gemaakt, die ons in staat stelt om aanroepen te doen naar een onderliggend RTOS zoals het FreeRTOS, dat wil zeggen we doen aanroepen naar CMSIS-RTOS (versie 2) om het onderliggende FreeRTOS aan te sturen.
FreeRTOS en multitasking is een zeer veelomvattend onderwerp en er zijn meerdere boeken voor nodig om alle mogelijkheden uit te leggen. Op het internet zijn verschillende boeken, handleidingen en toepassingsnotities voorhanden die nuttig kunnen zijn voor lezers voor wie de concepten van multitasking nieuw zijn. Het boek *ARM-Based Microcontroller Multitasking Projects – Using the FreeRTOS Multitasking Kernel* kan erg nuttig zijn om de concepten van multitasking te begrijpen en FreeRTOS te leren gebruiken in ARM-gebaseerde MCU's [1].

Een FreeRTOS-project met de STM32MCubeIDE

In het vervolg van dit artikel zullen we een eenvoudige toepassing ontwikkelen met 3 LED's die zijn aangesloten op het NUCLEO-L476RG-ontwikkelboard. De LED's worden LEDSLOW, LEDMEDIUM en LEDFAST genoemd. De aansluitingen van deze LED's en hun knippersnelheden zijn als volgt (zie ook **figuur 1**):

LED	Flitssnelheid	Aangesloten op GPIO-pen
LEDSLOW	elke seconde	PC0
LEDMEDIUM	elke 500 ms	PC1
LEDFAST	elke 250 ms	PC2

Een eerste programma

De te volgen stappen zijn:

- Start STM32CubeIDE.
- Kies voor het starten een nieuwe toepassing.
- Maak een nieuwe workspace.
- Selecteer STM32L476RG als processor.
- Noem het programma: *FREE*.
- Configureer *PC0*, *PC1*, en *PC2* als *GPIO_Output*. Klik met de rechtermuisknop op de pinnen en stel de User Labels in op respectievelijk *LEDSLOW*, *LEDMEDIUM* en *LEDFAST* (**figuur 2**).
- Configureer de MCU-klok op 80 MHz.
- Klik op *Middleware* aan de linkerkant en selecteer *FreeRTOS*.
- Stel de interface in op *CMSIS_V2*.
- Er zijn veel parameters die onder het tabblad *Configuration* kunnen worden geconfigureerd. Het gebruik van deze parameters vereist een goede kennis van FreeRTOS. In dit demoproject zullen we alle standaardwaarden accepteren (*figuur 3*).
- Klik op *File*, gevolgd door *Save*, en klik op *YES* om de code te genereren.
- Klik op *Core*, gevolgd door *Src*, en dubbelklik op *main c* om het hoofdprogramma weer te geven.

In dit programma hebben we 3 taken, ook wel threads genoemd. De basisfuncties in een FreeRTOS-programma zijn als volgt:

- definieer de thread-ID's;
- definieer de thread-attributen;
- initialiseer de scheduler;
- maak threads;
- start de scheduler.

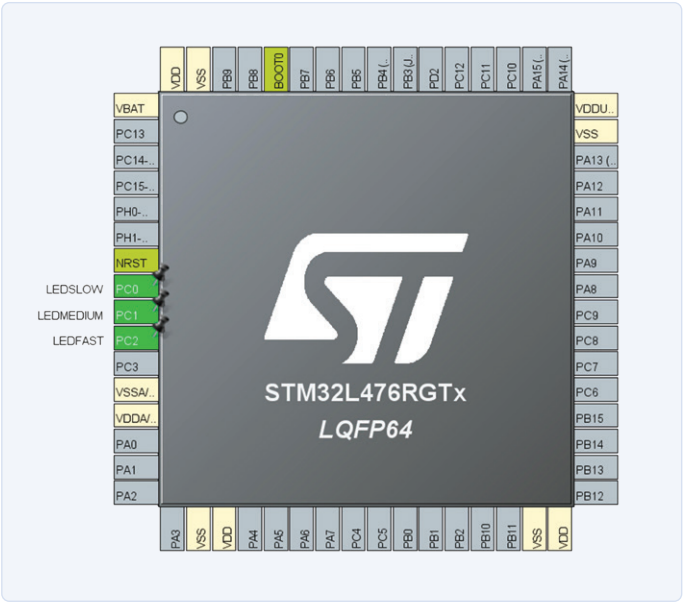
`osThreadId_t` wordt gebruikt om de thread ID's te definiëren. De thread ID's in dit programma zijn:

```
osThreadId_t LEDSTaskHandle; // Slow LED
osThreadId_t LEDMTaskHandle; // Medium LED
osThreadId_t LEDFTaskHandle; // Fast LED
```

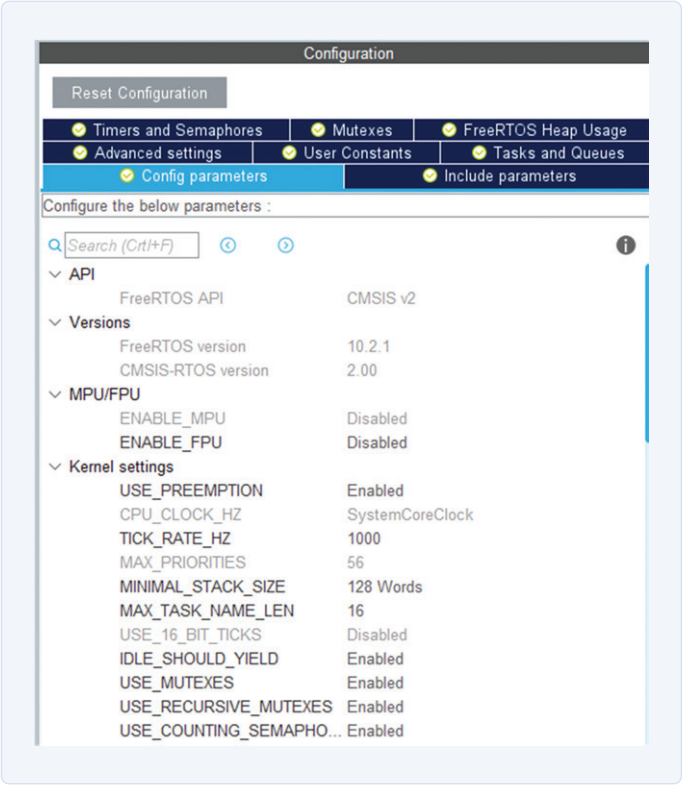
De thread-attributen definiëren verschillende parameters van een thread, zoals de naam, prioriteit, stackgrootte enzovoort. Als voorbeeld de thread-attributen voor de LEDSLOW-taak:

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

De scheduler wordt geïnitialiseerd en gestart met de functie-aanroepen:



Figuur 2. Configuratie van de uitgangen van de STM32-chip in STMCubeIDE.



Figuur 3. Accepteer onder Config parameters de standaardwaarden.

WEBLINK & LITERATUUR

[1] **Dogan Ibrahim: "ARM-Based-Microcontroller-Multitasking-Projects — Using the FreeRTOS." Newness, 2020**

[2] **Cumulatieve programma-download**: www.elektor.com/nucleo-boards-programming-with-the-stm32cubeide

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

Threads worden aangemaakt met de functie `call osThreadNew()`. De thread voor de langzame LED wordt bijvoorbeeld als volgt aangemaakt:

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

Waarbij `StartLEDSTask` de naam is van de functie die door de scheduler zal worden aangeroepen om de langzaam knipperende LED te implementeren. De inhoud ervan is:

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

Merk op dat we `osDelay()` in plaats van `HAL_Delay()` moeten gebruiken. `HAL_Delay()` in een taak met hoge prioriteit zou een context-omschakeling kunnen voorkomen. Maar `osDelay()` vertelt de scheduler om tijdens het wachten over te schakelen naar een andere taak.
Compileer het programma in Release-mode en sleep het binaire bestand *FREE.bin* naar het device *NUCLEO-L476RG*. Op dit punt draaien alle drie de threads tegelijkertijd in hun eigen `forever`-loops. De scheduler schakelt de threads aan en uit, zodat het lijkt alsof er drie threads tegelijk draaien. We zouden de drie LED's tegelijkertijd moeten zien knipperen met verschillende snelheden.

Het programma!

Listing 1 toont het programma met de naam *FREE*. Het programma kan worden opgehaald uit het gratis software-archief (.zip-bestand) op de Elektor-webpagina voor het boek [2]. Scroll op die pagina naar beneden naar Downloads en kies het bestand met de naam *Software_Nucleo Boards Programming with the STM32CubeIDE*. Unzip het op uw lokale schijf, en ga dan naar de map *FREE*.

210236-03

Vragen of opmerkingen?
Hebt u technische vragen of opmerkingen naar aanleiding van dit artikel? Stuur dan een e-mail naar de auteur via d.ibrahim@btinternet.com of naar de redactie van Elektor via redactie@elektor.com.

Een bijdrage van
Tekst: **Dogan Ibrahim**
Redactie: **Jan Buiting**
Vertaling: **Jelle Aarnoudse**
Layout: **Giel Dols**

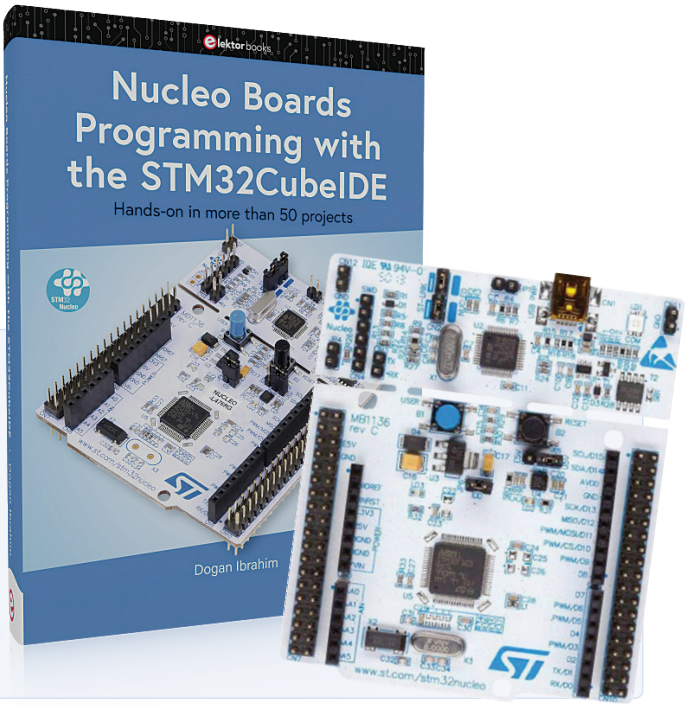
Listing 1. Het programma FREE.

```
/* 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****/
```



GERELATEERDE PRODUCTEN

- **Boek: Nucleo Boards Programming with the STM32CubeIDE**
  www.elektor.nl/nucleo-boards-programming-with-the-stm32cubeide
- **E-boek: Nucleo Boards Programming with the STM32CubeIDE**
  www.elektor.nl/nucleo-boards-programming-with-the-stm32cubeide-e-book
- **STM32 Nucleo L476RG Board**
  www.elektor.nl/stm32-nucleo-l476rg-board