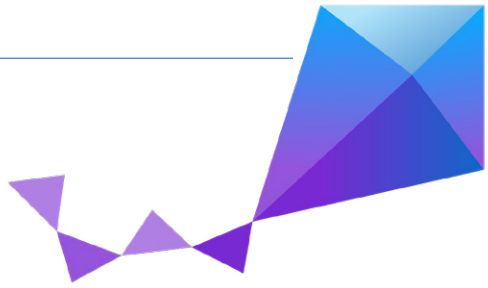


commencer avec le RTOS Zephyr

aussi puissant que difficile à maîtriser

Clemens Valens (Elektor)

Zephyr est un système d'exploitation en temps réel (RTOS) petit et évolutif, optimisé pour les appareils à ressources limitées, sur plusieurs architectures. Hébergé par la Fondation Linux (voir [1]), le projet est un effort de collaboration open-source dont l'objectif est de fabriquer un RTOS de premier ordre. Le système d'exploitation (SE) Zephyr a gagné en popularité ces dernières années dans l'informatique embarquée et aujourd'hui, de nouveaux microcontrôleurs et cartes de développement arborent fièrement le support de Zephyr. Il est temps d'y regarder de plus près.

Zephyr est évolutif, ce qui lui permet de s'adapter à une large gamme d'appareils ayant des contraintes de ressources différentes. L'évolutivité est obtenue grâce à une architecture modulaire qui permet aux développeurs de n'inclure que les composants dont ils ont besoin, minimisant ainsi l'empreinte du système. Le site web de Zephyr indique qu'il fonctionne sur des systèmes dotés d'une mémoire de 8 KO jusqu'à plusieurs gigaoctets.

Prise en charge d'un grand nombre de supports

Zephyr supporte une large gamme d'architectures, y compris ARM, x86, RISC-V, Xtensa, MIPS et plus encore. Les FPGA sont également pris en charge avec les noyaux logiciels Nios2 et MicroBlaze. À l'heure où nous écrivons ces lignes, Zephyr répertorie plus de 600 cartes utilisables, dont les Arduino UNO R4 Minima, GIGA R1 WIFI et Portenta H7, de multiples saveurs de l'ESP32, les deux versions du BBC micro:bit, le Raspberry Pi Pico (et même le Raspberry Pi 4B+), les cartes nRF51 et nRF52, la famille NXP MIMXRT1010-EVK, et les familles STM32 Nucleo et Discovery. Je n'ai cité que les cartes couramment rencontrées dans Elektor, mais il en existe bien d'autres.
Outre les cartes processeurs, Zephyr prend également en charge de nombreuses cartes d'extension (connues sous le nom de *shields*) et est livré avec des pilotes pour toutes sortes d'interfaces et plus de 150 capteurs.

Multitâche, mise en réseau et gestion de l'énergie

En tant que système d'exploitation en temps réel (RTOS), Zephyr offre des fonctionnalités telles que le multitâche préemptif, la communication inter-thread et la prise en charge de l'horloge en temps réel. Le système d'exploitation est également doté de technologies et de protocoles de réseau tels que TCP/IP, Bluetooth, IEEE 802.15.4 (utilisé par exemple dans Zigbee), MQTT, NFS et LoRaWAN. Associées à ses capacités de mise en réseau, les fonctionnalités de gestion de l'énergie intégrées à Zephyr le rendent adapté aux applications IdO économes en énergie et aux appareils fonctionnant sur batterie.
Un ensemble de bibliothèques et d'intergiciels simplifient les tâches courantes, telles que les protocoles de communication, les systèmes de fichiers et les pilotes de périphériques.
Sachez que Zephyr est également conçu pour être compatible avec les certifications de sécurité telles qu'ISO 262, ce qui le rend adapté aux applications critiques en matière de sécurité.

Inspiré par Linux

Zephyr n'est pas Linux, mais il utilise des concepts, des techniques et des outils utilisés par ce dernier. Par exemple, Kconfig est utilisé pour configurer le système d'exploitation, et les propriétés et configurations matérielles sont décrites à l'aide de la spécification de l'arbre des périphériques (DTS) [2]. Les développeurs Linux se sentiront donc rapidement à l'aise lorsqu'ils coderont pour Zephyr.

Open Source

Enfin, Zephyr est publié sous la licence Apache 2.0, qui permet une utilisation commerciale et non commerciale. Sa communauté d'utilisateurs fournit de l'aide et de la documentation. Vous pouvez également la rejoindre.

Essai de Zephyr

Essayer Zephyr est sur ma liste de choses à faire depuis plusieurs années, mais mes premières expériences avec cet SE n'étaient pas très encourageantes, et je n'ai donc jamais approfondi. À l'époque,

Figure 1. La minuscule carte BBC micro:bit est une excellente cible pour essayer le RTOS Zephyr. Qui aurait pensé que cette petite carte destinée à enseigner la programmation à des enfants de 10 ans à l'aide de MakeCode, un langage graphique de type Scratch, serait aussi un super outil pour les développeurs chevronnés de logiciels embarqués désireux d'apprendre un système d'exploitation en temps réel de qualité industrielle ?

l'un de ses principaux problèmes (outre le fait de le compiler sans erreur) était qu'il nécessitait un adaptateur de programmation pour programmer le contrôleur cible, ce qui le rendait moins adapté aux *makers*. Grâce à Arduino et son chargeur d'amorçage (*bootloader*), nous nous sommes habitués à ne pas avoir besoin d'outils de programmation spéciaux, et en exiger un me semblait un pas en arrière.

Choisir une carte

Les choses ont évolué depuis. Comme mentionné plus haut, Zephyr supporte aujourd'hui plus de 600 cartes de microcontrôleurs. Il y a de fortes chances que vous possédiez déjà une ou plusieurs compatibles. En regardant la liste, j'ai découvert que j'en avais plus d'une douzaine différentes à ma disposition.

Vive la BBC micro:bit !

J'ai essayé la plupart d'entre elles pour finalement me fixer sur la BBC micro:bit pour mes expériences (**figure 1**, connu par Zephyr sous le nom de `bbc_microbit` ou `bbc_microbit_v2`, selon la version de la carte). Comparé à mes autres options, en plus d'être facilement disponible, elle a probablement le meilleur support Zephyr, ce qui signifie que tous ses périphériques sont accessibles et supportés par quelques exemples. Mieux encore, il peut être programmé *et* débogué sans avoir besoin d'outils supplémentaires.

La populaire ESP-WROOM-32 (connu par Zephyr sous le nom `esp32_devkitc_wroom`) est également une candidate appropriée, mais le débogage nécessite un outil externe.

L'Arduino GIGA R1 WIFI est une autre bonne option, mais son *bootloader* est détruit lors de l'utilisation de Zephyr. On peut le restaurer, bien sûr, mais c'est un effet collatéral non souhaité.

Officiellement, l'Arduino UNO R4 Minima nécessite une sonde de programmation compatible SWD (comme beaucoup d'autres cartes, dont la Raspberry Pi Pico), mais j'ai trouvé un moyen de contourner ce problème en utilisant *dfu-util* (voir ci-dessous). Cependant, comme la GIGA R1, son chargeur d'amorçage Arduino est piétiné par Zephyr.

Utilisez plutôt un émulateur

Si vous n'avez pas de carte appropriée, mais que vous voulez vraiment essayer Zephyr, sachez qu'il dispose d'un support d'émulateur intégré pour QEMU (sur Linux/macOS uniquement). Cela vous permet d'exécuter et de tester des applications virtuellement. Renode d'Antmicro est capable de réaliser des prouesses similaires, bien que je ne l'aie pas essayé.

Installer Zephyr

J'ai installé Zephyr sur un ordinateur fonctionnant sous Windows 11, je n'ai pas essayé Linux ou macOS. Des instructions d'installation détaillées et fonctionnelles sont disponibles en ligne [3]. Les étapes à suivre sont clairement indiquées et ne nécessitent pas d'explications supplémentaires. J'ai utilisé un environnement Python virtuel, comme suggéré. Cela signifie que vous devez noter la commande d'activation de l'environnement virtuel quelque part, car vous en aurez besoin à chaque fois que vous commencerez à travailler. Si vous utilisez Windows PowerShell, vous devez lancer le *script activate.ps1* ; dans l'invite de commandes, il s'agit du fichier *batch activate.bat*. Windows PowerShell est plus performant pour traiter les messages émit par le compilateur et l'éditeur de liens (**figure 2**).

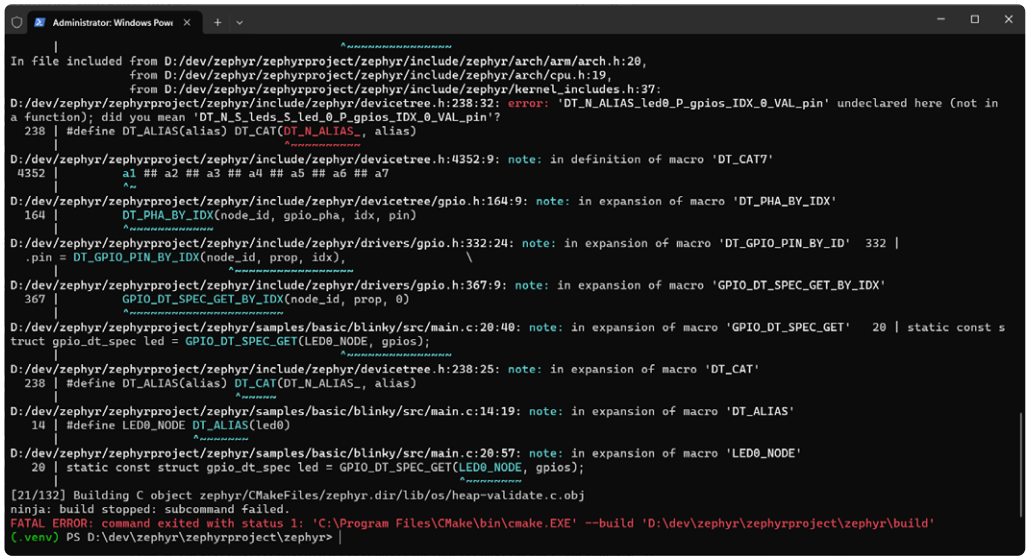


Figure 2. L'outil de construction west de Zephyr est destiné à fonctionner dans un terminal. Le cmd.exe de Windows peut être utilisé, mais ce n'est pas un terminal. Windows PowerShell, alias Terminal, est donc plus adapté.

Figure 3. Dans de nombreuses situations (mais pas toutes), un programmateur/débogueur JTAG ou SWD est nécessaire pour vos expériences Zephyr.

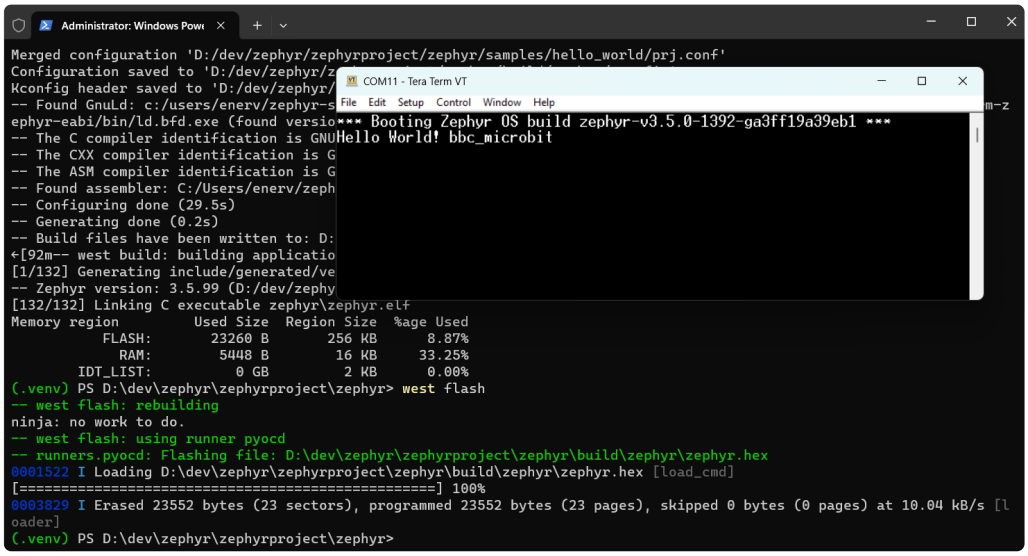


Figure 4. L'exemple de Hello World n'est pas très étendu. Si vous êtes trop lent à ouvrir un terminal série, vous ne verrez pas le message de bienvenue.

Zephyr se compose de deux parties, le SE lui-même et un SDK contenant une collection de chaînes d'outils microcontrôleur (21 au moment de la rédaction…). Le SE et le SDK n'ont pas besoin d'être installés au même endroit. Au total, pour moi, cela a consommé environ 12 GB d'espace précieux sur le disque dur. Pour récupérer de l'espace, vous pouvez supprimer les chaînes d'outils dont vous n'avez pas besoin.

Après l'installation, vérifiez qu'il fonctionne en compilant un exemple (*sample*) et en le téléchargeant sur la carte avec la commande ci-dessous. Remplacez `<ma_carte>` par le nom de votre carte, par exemple `arduino_uno_r4_minima` :

```
west build -p always -b <my_board> samples/basic/blinky
```

Si vous ne souhaitez pas modifier le chemin d'accès à l'exemple, vous devez exécuter cette commande à l'intérieur du dossier (où `(.venv)` indique que vous vous trouvez dans un environnement virtuel) :

```
(.venv) <my_path_to_zephyr>\zephyrproject\zephyr
```

Si l'exemple se construit sans erreur, vous pouvez le télécharger sur la carte à l'aide de la commande :

```
west flash
```

et la LED « default » de la carte commence à clignoter à une fréquence de 0,5 Hz.

Comme mentionné précédemment, le clignotement peut nécessiter un outil de programmation externe, tel qu'un adaptateur J-Link ou un autre programmateur (compatible JTAG ou SWD) (**figure 3**) et le logiciel pilote pour celui-ci doit être accessible (c'est-à-dire dans le chemin de recherche, *%PATH%* sous Windows). Si ce n'est pas le cas, vous en serez informé par un message (souvent long et énigmatique).

Sur la BBC micro:bit V2, la première fois, j'ai dû copier le fichier HEX manuellement sur la carte en utilisant la procédure de programmation standard du micro:bit. Ensuite, la commande *flash* a fonctionné correctement. L'exécutable *zephyr.hex* se trouve dans *zephyrproject\zephyr\build\zephyr*.

La commande *flash* par défaut pour les cartes Arduino, UNO R4 Minima et GIGA R1 WIFI, nécessite que l'utilitaire de programmation *dfu-util* soit dans le chemin de recherche du système d'exploitation (avant d'activer l'environnement virtuel, si vous en utilisez un). Cet utilitaire est inclus dans l'EDI Arduino. Cependant, c'est à vous de déterminer où il se trouve exactement sur votre ordinateur (par défaut dans *%HOMEPATH%\AppData\Local\Arduino15\packages\arduino\tools\dfu-util\<votre version Arduino la plus récente>*). La carte doit également être mise en mode DFU. Pour ce faire, appuyez deux fois sur le bouton de réinitialisation. Lorsque la LED commence à « pulser » ou à « respirer », vous pouvez flasher le programme.

Compatibilité avec Blinky

Vous avez peut-être remarqué que j'ai suggéré l'Arduino UNO R4 Minima comme carte à utiliser pour l'exemple Blinky et non la BBC micro:bit dont j'étais si enthousiaste plus tôt. La raison est que, bien qu'elle ait 25 LED (sans compter l'indicateur de mise sous tension), elle n'a pas de LED compatible avec l'exemple Blinky. L'ESP Wroom 32 n'en a pas non plus, mais la R4 Minima en a une.

La GIGA R1 est également compatible avec Blinky. Le microcontrôleur de cette carte possède deux cœurs (Cortex-M7 et -M4) et Zephyr vous permet de choisir lequel utiliser en sélectionnant soit `arduino_giga_r1_m4` soit `arduino_giga_r1_m7` pour la commande de construction. Vous pouvez montrer que les cœurs sont effectivement indépendants en flashant l'exemple Blinky deux fois, une fois pour le -M4 et une autre pour le -M7. La GIGA possède une LED RGB et Blinky utilisera des couleurs différentes pour chaque cœur : bleu pour le M4 et rouge pour le M7. Pour rendre les deux Blinky plus distincts, vous pouvez changer le taux de clignotement de l'un d'entre eux (dans *samples\basic\blinky\src\main.c*, changez la valeur de `SLEEP_TIME_MS`).

Hello World

Pour les cartes sans Blinky LED, il y a l'exemple *hello_world* qui émet une chaîne de caractères sur le port série.

```
west build -p always -b <my_board> samples/hello_world
west flash
```

Cet exemple fonctionne avec le BBC micro:bit et le module ESP-WROOM-32. Pour voir la chaîne de sortie, ouvrez un

programme de terminal série sur votre ordinateur. Le débit est généralement de 115 200 bauds (115200,n,8,1). Il se peut que vous deviez d'abord réinitialiser la carte, car le message n'est envoyé qu'une seule fois, et vous l'avez peut-être manqué (**figure 4**).
Sur la R4 Minima et la GIGA R1, la sortie série est sur la pin 1 et non, comme vous pouvez naïvement vous y attendre, sur le port USB-C comme ce serait le cas sur l'EDI Arduino. Ceci est dû au fait que le port USB est un périphérique du microcontrôleur et non une puce séparée. Zephyr étant un système d'exploitation modulaire et évolutif, le support USB, comme tout autre module ou périphérique, doit être explicitement activé pour le projet avant de pouvoir être utilisé. Cela se fait dans les fichiers de configuration du projet. Nous y reviendrons plus tard.
Pour les cartes sans convertisseur série-USB intégré, vous devez trouver le port série (généralement le port 0 au cas où le microcontrôleur en aurait plusieurs) et le connecter à votre ordinateur par le biais d'un convertisseur série-USB externe.

Pour aller un peu plus loin

Si vous avez réussi à faire fonctionner les exemples Blinky et *hello_world* sur votre carte, vous êtes en bonne position pour créer une application fonctionnant sur Zephyr. Si un seul est fonctionnel, et que vous souhaitez que l'autre le soit aussi, les choses sont un peu plus compliquées.
J'ai choisi la BBC micro:bit pour les expériences Zephyr, même si elle n'est pas compatible avec l'exemple Blinky. Ce n'est pas vraiment un problème, car la carte est livrée avec quelques exemples (dans le sous-dossier *bbc_microbit* du dossier *samples\boards*), dont l'un (*display*) est beaucoup plus joli qu'un Blinky à une seule diode électroluminescente. Il y a aussi des exemples pour d'autres cartes, mais très peu par rapport aux plus de 600 cartes supportées (même pas 5%). De plus, la plupart de ces exemples concernent des cas d'utilisation avancés ou obscurs.
Lorsque vous essayez de construire Blinky pour la BBC micro:bit (ou l'ESP-WROOM-32 ou toute autre carte incompatible), vous obtenez un message d'erreur incompréhensible. Ce qu'il essaie de vous dire, c'est que `led0` est une entité inconnue. C'est la LED Blinky par défaut (un peu comme `LED_BUILTIN` sur Arduino). Comme la micro:bit possède un port d'extension sur lequel vous pouvez connecter des LED et d'autres choses, essayons de définir l'une des broches de ce port comme `led0`.
Avant cela, faites une copie de sauvegarde du dossier *samples/basic/blinky* ou créez une copie avec un nouveau nom et utilisez-la à la place. Dans ce qui suit, le dossier *samples/basic/blinky* est utilisé.

L'arbre des périphériques

La définition d'une `led0` nous amène à l'arbre des périphériques, déjà brièvement mentionné. Il est contenu dans un ou plusieurs fichiers texte et liste les périphériques et la mémoire disponibles sur une carte ou dans un contrôleur. Dans Zephyr, ces fichiers ont le format *.dts* ou *.dtsi* (« I » pour *include*) et les fichiers peuvent inclure d'autres fichiers. Les fichiers *.dtsi* du processeur se trouvent dans le dossier *dts*, les fichiers *.dts* et *.dtsi* de la carte se trouvent dans le dossier *boards*. Les deux dossiers sont organisés par architecture de processeur.
Pour mieux visualiser les fichiers DTS(I), vous pouvez utiliser le plugin DeviceTree pour Visual Studio Code [4]. Il permet la mise en

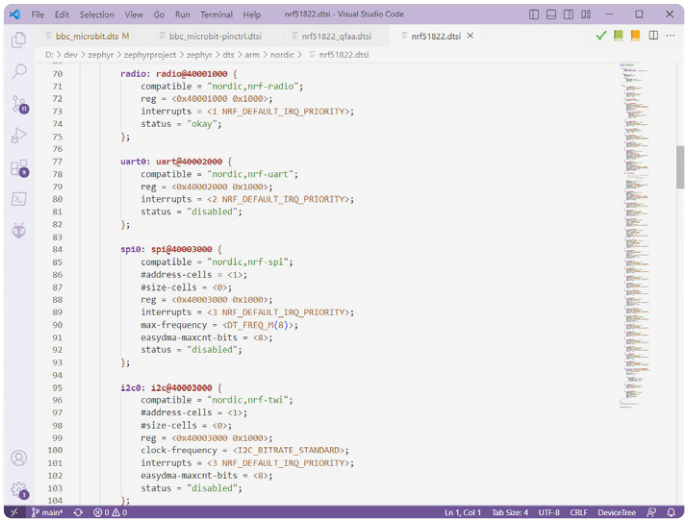


Figure 5. Un extrait du fichier nrf51822.dtsi. La vue zoomée à droite montre la longueur de ce fichier.

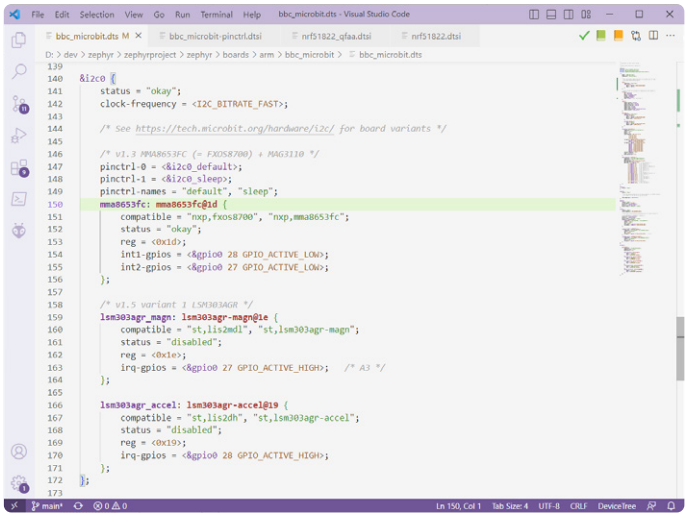


Figure 6. Cette section de l'arbre des périphériques représente le bus I²C de la carte BBC micro:bit, et non de son processeur. Tiré du fichier bbc_microbit.dts.

évidence de la syntaxe et le pliage du code, ce qui rend les fichiers plus faciles à lire (les fichiers DTS utilisent une syntaxe de type langage C). La **figure 5** montre un extrait du fichier *.dtsi* pour le nRF51822 qui est au cœur de la carte BBC micro:bit V1. Ce fichier est inclus dans le fichier DTS de la carte. À titre d'exemple, notez que le statut de l'`uart0` est réglé sur `disabled`. Ce statut est surchargé dans le fichier DTS de la carte, où il est réglé sur `okay`, ce qui signifie qu'il peut être utilisé. Il en va de même pour `gpio0` et `i2c0`.

I²C dans l'arborescence

Un autre extrait du fichier .dts pour la BBC micro:bit est visible dans la **figure 6**. Il montre l'arbre des périphériques pour le bus I²C. La micro:bit a un ou deux capteurs attachés au bus (selon la variante de la carte micro:bit V1) et ils sont représentés dans l'arbre par `mma8653fc` et `lsm303agr` (ce dernier comprend deux capteurs, c'est pourquoi il apparaît deux fois). Le premier a l'état `okay`, tandis que les deux autres sont `disabled`. Ceci est correct pour ma variante de carte, qui est un échantillon de la toute première génération de micro:bit V1.
Comme le montre l'extrait, ce capteur est compatible avec le FXOS8700 et le MMA8653FC, son adresse sur le bus I²C est 0x1d, et deux signaux `int` (*interrupt*) sont déclarés qui sont connectés à

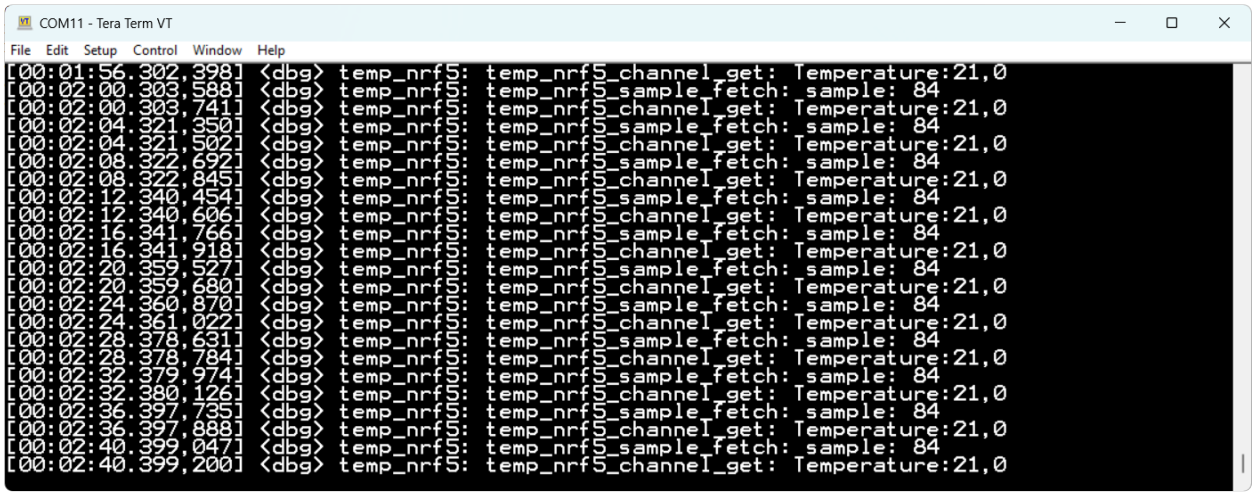


Figure 7. La sortie de la démo samples/sensor/fxos8700 fonctionnant sur la carte BBC micro:bit.

deux broches GPIO : 27 et 28. Si vous voulez l'essayer, un programme de démonstration est disponible :

```
west build -p always -b bbc_microbit samples/sensor/
fxos8700
west flash
```

Notez que cela ne fonctionnera pas pour la BBC micro:bit V2, car elle a un capteur différent dans son arbre de périphériques.
La sortie de la démo est montrée dans la **figure 7**, mais nous nous éloignons du sujet.

Superposition de l'arbre des périphériques

Revenons à notre LED, `led0`. L'arbre des périphériques de la carte ne mentionne pas `led0`, comme prévu, et nous devons donc l'ajouter. Nous pourrions l'insérer directement dans le fichier de l'arbre des périphériques de la carte, mais ce serait incorrect, car elle n'a pas de `led0`. La bonne façon d'étendre l'arborescence d'un périphérique est d'ajouter un fichier de superposition (*overlay* en anglais). Le contenu d'un fichier de superposition y est ainsi ajouté. Les sections qui existent dans l'arborescence sont étendues (dans le cas d'un nouvel élément) ou écrasées (dans le cas où l'élément existe déjà dans l'arborescence) ; de nouvelles sections sont ajoutées.
Les fichiers de superposition doivent être placées à l'intérieur du dossier du projet, dans un sous-dossier nommé boards. Lorsque ce sous-dossier est présent, le processus de construction y recherchera un fichier portant le nom *<ma_carte>.overlay*. Dans mon cas, le nom du fichier est *bbc_microbit.overlay* (*bbc_microbit_v2.overlay* pour les utilisateurs de la V2). La **figure 8** montre le contenu du fichier.

Ajouter une LED clignotante

Zephyr a un mot-clé spécial pour l'arbre des périphériques pour les LED, qui est `leds`, donc nous créons un nœud (branche) pour cela (vous pouvez l'appeler comme vous voulez, mais tenez-vous en à `leds` s'il est supposé être superposé à un nœud `leds` existant). Il doit être ajouté à la racine de l'arbre ; c'est pourquoi une barre oblique '/' est placée devant, car elle signifie racine en langage DT. La ligne suivante indique que cette branche est compatible avec le pilote `gpio-leds` intégré à Zephyr (vous pouvez trouver l'interface de ce pilote dans *zephyr\include\zephyr\drivers\led.h*).

Nœuds fils

Vient ensuite la liste des nœuds fils des LED. Comme je n'ai qu'une LED, il n'y a qu'un seul nœud fils, `led_0`, que j'ai étiqueté `led0`. L'étiquetage d'un nœud est facultatif, mais il permet de le référencer ailleurs dans l'arbre, ce que nous ferons quelques lignes plus bas. De plus, ils peuvent être utilisés par (le développeur de) l'application pour accéder aux nœuds et à leurs propriétés.
Un nœud fils doit spécifier les propriétés de l'appareil. Dans le cas d'une LED, seule la broche GPIO est une propriété obligatoire, mais la propriété optionnelle nommée `label` peut être ajoutée. Ces étiquettes peuvent être utilisées pour fournir de la documentation ou des informations lisibles par l'utilisateur. Elles n'ont pas d'autre utilité.
Comme broche GPIO, j'ai choisi `1`, qui correspond au grand trou `2` sur le connecteur d'extension de la micro:bit. Si vous avez une BBC micro:bit V2, utilisez `4` comme broche GPIO (au lieu de `1`).

Créer un alias

L'étape suivante est nécessaire, car l'exemple Blinky l'attend. Elle consiste à créer l'alias `led0` pour notre LED. On pourrait penser qu'étiqueter le nœud fils aurait été suffisant, mais ce n'est pas le cas, car Blinky utilise la macro `DT_ALIAS` pour y accéder. Par conséquent, nous devons fournir quelque chose que cette macro peut assimiler, ce qui se trouve être un alias. Il se trouve à l'intérieur du bloc alias. Si Blinky avait utilisé la macro `DT_NODELABEL` à la place,

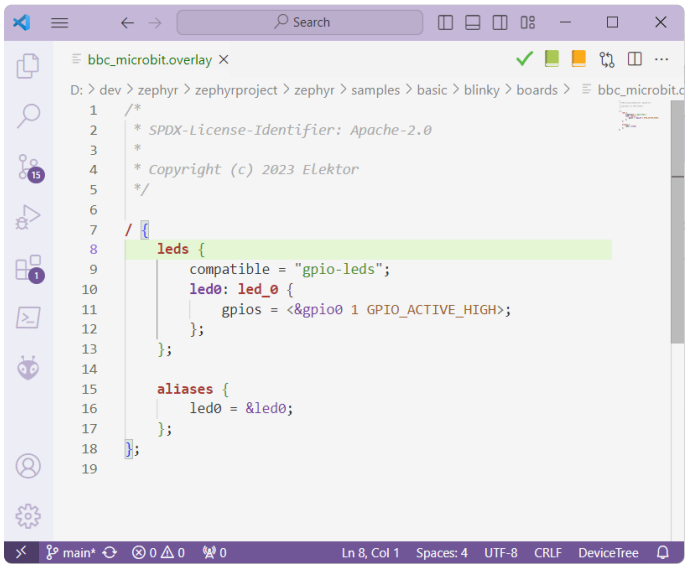


Figure 8. Ce fichier de superposition d'abre de périphériques rend la BBC micro:bit V1 compatible avec l'exemple Blinky.

un alias aurait été superflu, car `DT_NODELABEL` récupère directement l'étiquette du nœud fils `led0`. Je sais que le fait que les étiquettes et les alias portent le même nom est un peu déroutant, mais c'est nécessaire pour mon explication.

Macros Zephyr

Bien que les macros soient mal vues dans la programmation C/C++, Zephyr les utilise beaucoup. Celles telles que `DT_ALIAS` et `DT_NODELABEL` permettent à l'application et aux outils de configuration du projet d'extraire des informations de l'arbre des périphériques, et elles sont nombreuses. Vous trouverez leur description dans le manuel Zephyr, au chapitre « Devicetree API » [5].

Fait amusant : de nombreuses (toutes ?) macros Zephyr s'attendent à ce que leurs arguments soient en minuscules, les caractères qui ne sont pas des lettres ('a' à 'z') ou des chiffres ('0' à '9') étant remplacés par des tirets bas ('_'). C'est ce qu'on appelle la compatibilité *lowercase-and-underscores* (minuscules et tirets bas). Par exemple, imaginons que j'ai étiqueté le nœud fils LED d'avant `LED-0` au lieu de `led0`. L'argument à utiliser avec `DT_NODELABEL` aurait alors été led_0, c'est-à-dire `DT_NODELABEL`(led_0), car '-' n'est ni une lettre ni un chiffre, et les lettres doivent être en minuscules. En d'autres termes, pour le développeur d'applications utilisant des macros d'arborescence, le caractère de soulignement est un joker. Ainsi, `led_0` dans l'application peut faire référence à `led_0`, `led-0`, `Led_0`, `LED-0` et `ledé0` (et toutes les autres variantes possibles) dans l'arbre des périphériques. En gardant cela à l'esprit, il est fortement recommandé de lire attentivement la documentation des macros Zephyr.

Tu ne feras pas d'erreurs

Notez que faire des erreurs dans l'arbre des périphériques est puni par le compilateur qui vous crie *FATAL ERROR* sans fournir d'autres informations.

Constructions parfaites

Lorsque vous jouez avec l'arbre des périphériques ou avec votre application, il est probable que vous recompilez souvent votre projet. Pour accélérer un peu les choses, supprimez l'option `-p always`('p' de *pristine*) de la commande de compilation. Cela l'empêchera de tout reconstruire à partir de zéro. Si, en revanche, vous essayez de nombreux exemples différents les uns après les autres, gardez-le, car cela vous évitera une erreur ennuyeuse concernant le dossier de compilation qui n'est pas destiné à votre projet.

Notez que la commande *flash* déclenche également la dernière commande *build*, il suffit donc d'exécuter la commande *flash* à chaque fois que vous modifiez quelque chose.

Utiliser un pilote de périphérique

L'exemple Blinky appelle la fonction `gpio_pin_toggle_dt()` pour basculer l'état de la LED. Il s'agit d'une fonction du pilote GPIO. Bien sûr, c'est tout à fait possible, mais Zephyr inclut également une collection de pilotes de LED. Leur utilisation ne rend pas seulement le programme plus lisible, mais aussi plus flexible et portable, car un pilote de LED peut être remplacé par un autre sans modifier l'application elle-même. C'est là que l'évolutivité et la modularité de Zephyr entrent en jeu.

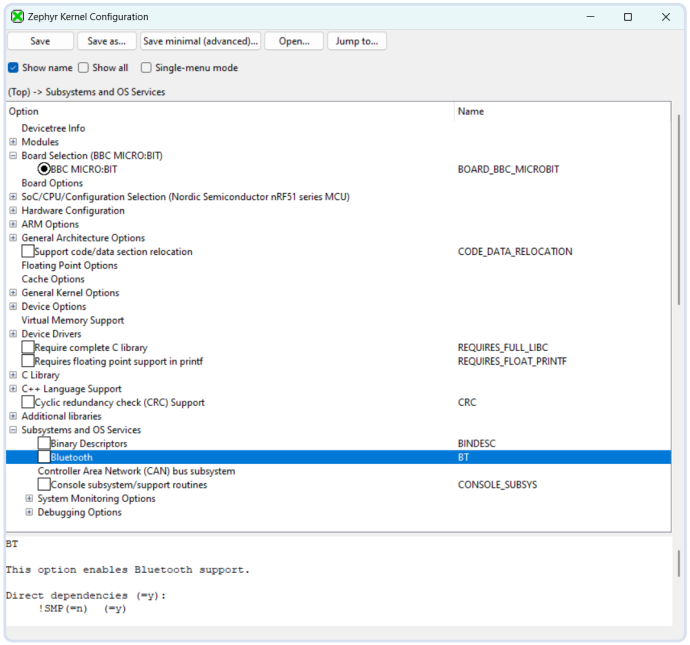


Figure 9. La GUI de l'outil de configuration de projet Kconfig. Les options sont nombreuses.

Kconfig a une interface graphique

L'intégration d'un pilote de périphérique LED dans notre programme nécessite quelques étapes. Tout d'abord, le projet doit être reconfiguré afin de l'inclure. La configuration du projet est gérée par Kconfig, le système de configuration du noyau au moment de la compilation, également utilisé par Linux. Il y a plusieurs façons d'interagir avec lui, et l'une d'entre elles est une interface utilisateur graphique (GUI). Dans Zephyr, vous l'ouvrez comme suit :

```
west build -t guiconfig
```

La GUI met un certain temps à s'ouvrir, mais lorsqu'elle le fait, elle ressemble à la **figure 9**. Elle affiche de nombreuses informations sur le projet en cours de développement. Notez la partie projet en cours de développement. Pour s'assurer que Kconfig travaille bien sur votre projet, faites une construction vierge (avec l'option `-p always`) de votre projet avant de lancer la GUI de Kconfig.

Tant d'options de configuration…

Prenez le temps d'explorer l'arbre de configuration. Déployez les branches en cliquant sur les symboles '+'. Mettez les options en surbrillance en cliquant dessus. Cela affichera des informations dans le volet inférieur. Notez que la prise en charge de la virgule flottante pour `printf()` est une option de configuration, tout comme la prise en charge du langage C++. De même, sous *Build and Link Features*, vous pouvez trouver des options d'optimisation du compilateur.

Il existe des tonnes d'options de configuration. Celle qui nous intéresse se trouve dans la branche *Device Drivers*. Déployez-la et faites défiler vers le bas tout en regardant tout ce qui est disponible. Le pilote de LED se trouve à peu près à mi-chemin : *Light-Emitting Diode* (LED) Drivers (pilotes de diodes électroluminescentes). Cochez la case. Laissez les options de sa sous-branche sur leurs valeurs par

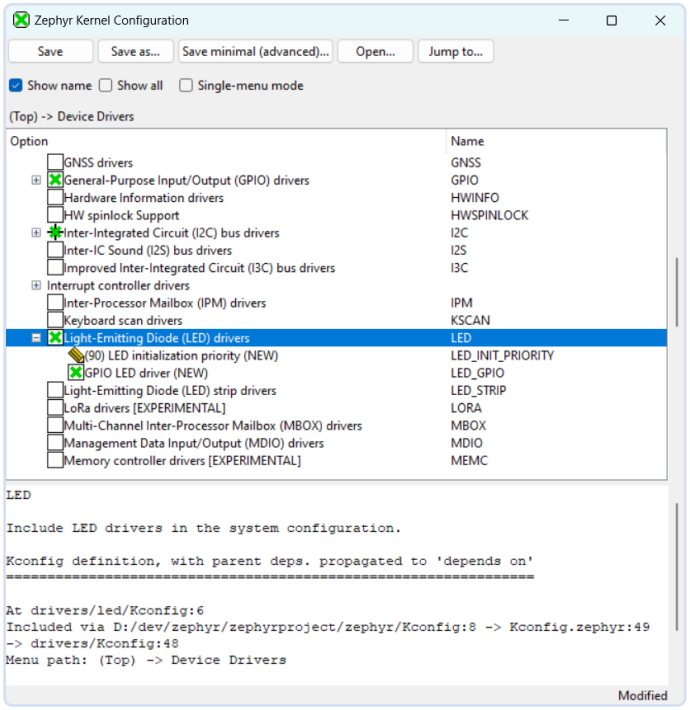


Figure 10. Sélectionnez Light-Emitting Diode (LED) Drivers et laissez les valeurs enfant telles quelles.

```c
#include <zephyr/kernel.h>
#include <zephyr/device.h>
#include <zephyr/drivers/led.h>

#define SLEEP_TIME_MS   (1000) /* 1000 msec = 1 sec */

int main(void)
{
    printf("\nBlinky with LED device driver\n");

    const struct device *const led_device = DEVICE_DT_GET_ANY(gpio_leds);
    if (!led_device)
    {
        printf("No device with compatible 'gpio-leds' found\n");
        return 0;
    }
    else if (!device_is_ready(led_device))
    {
        printf("LED device %s not ready\n", led_device->name);
        return 0;
    }

    while (1)
    {
        int result = led_on(led_device,0);
        if (result<0)
        {
            printf("led_on failed\n");
            return 0;
        }
        k_msleep(SLEEP_TIME_MS);

        result = led_off(led_device,0);
        if (result<0)
        {
            printf("led_off failed\n");
            return 0;
        }
        k_msleep(SLEEP_TIME_MS);
    }

    return 0;
}
```

Figure 11. Le programme Blinky réécrit pour être utilisé avec un pilote de périphérique LED. Notez qu'il n'y a pas de code spécifique à la carte. Ce programme devrait fonctionner sur n'importe quelle carte ayant un pilote `gpio_leds` *(ou* `gpio-leds`*) listé dans son arbre de périphériques.*

défaut (**figure 10**). Cliquez sur le bouton *Save* et notez le chemin du fichier de configuration imprimé en bas de la fenêtre. Il est instructif d'inspecter ce fichier, juste pour voir ce qu'il contient (beaucoup de choses). Fermer l'interface graphique.
A partir de maintenant, ne spécifiez plus le `-p always` dans les commandes de compilation, car cela annulerait les changements que vous avez faits ci-dessus. Je montrerai plus tard comment rendre le changement de configuration permanent.

Blinky avec le pilote de périphérique LED

Nous pouvons maintenant écrire le nouveau programme Blinky, voir **figure 11**. Il commence par inclure les fichiers d'en-tête du périphérique et du pilote de LED. Ensuite, dans le programme principal, nous employons la macro `DEVICE_DT_GET_ANY` pour obtenir une référence de périphérique pour la LED à partir de l'arbre des périphériques. Notez que l'argument `gpio_leds` de la macro est compatible avec les minuscules et les tirets bas, de sorte qu'il correspondra à la valeur `gpio-leds` de la propriété compatible du nœud leds dans l'arbre des périphériques (comme expliqué ci-dessus). S'il n'en trouve pas parce que vous avez fait une faute de frappe ou autre, le nouveau Blinky affichera le message `No device with compatible gpio-leds found`. Ce message sera également déclenché lorsque la propriété d'état d'un périphérique est réglée sur `disabled`.
Il faut un peu de temps pour s'habituer à l'utilisation du mot `compatible` comme substantif dans Zephyr. Ainsi, le message d'erreur ci-dessus ne signifie pas qu'il n'y a pas de périphérique compatible, mais qu'il n'y a pas de périphérique ayant une propriété nommée `compatible` avec la valeur `gpio-leds` (ni d'ailleurs avec `gpio_leds`, rappelons que le tiret bas '_' remplace n'importe quel caractère sauf 'a' à 'z' et '0' à '9').
Une deuxième vérification permet de s'assurer que le périphérique s'est initialisé correctement. En supposant que c'est le cas, nous continuons.
Dans la boucle infinie `while()`, la LED est allumée et éteinte à l'aide des commandes `led_on` et `led_off` fournies par le pilote [6]. L'argument `0` signifie le premier (et seul) périphérique trouvé par la macro `DEVICE_DT_GET_ANY`, qui est `led0`.

Vérifier les valeurs de retour

Comme nous utilisons un pilote de périphérique au lieu de basculer directement une broche GPIO au niveau du registre matériel, il est bon de vérifier les valeurs de retour de tous les appels de fonction du pilote, car ils peuvent échouer pour une raison ou une autre. Un pilote doit fournir certaines fonctions et certains rappels, mais il peut aussi avoir des caractéristiques optionnelles. Un pilote de LED, par exemple, doit implémenter `led_on` et `led_off`, mais `led_blink` est optionnel. Si vous appelez `led_blink` dans notre projet, il ne se passera rien parce qu'il n'est pas implémenté. Elle existe, mais elle est vide. La valeur de retour vous le montrera. En général, c'est une bonne pratique de programmation que de vérifier la valeur de retour de chaque appel de fonction.
Construisez et téléchargez le programme comme suit (notez l'absence du drapeau `-p always`).

```
west build -b bbc_microbit samples/basic/blinky
west flash
```

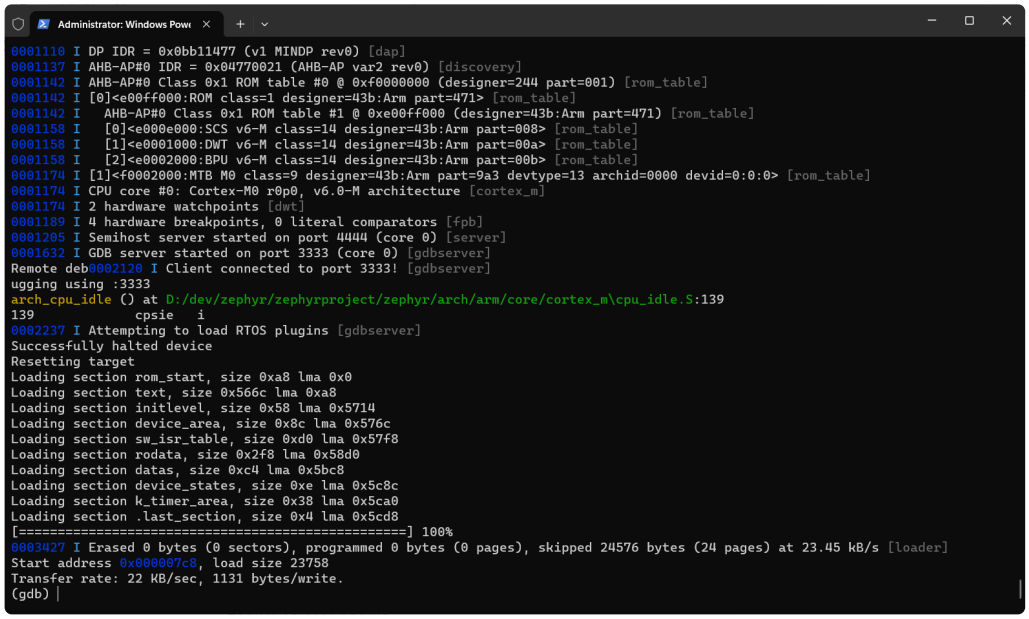


Figure 12. La BBC micro:bit supporte le débogage avec gdb sans avoir besoin d'outils supplémentaires ou quoi que ce soit.

Configurer le projet

Si la LED a commencé à clignoter à 0,5 Hz, nous avons un programme qui fonctionne. Pour qu'il le reste, nous devons rendre la configuration actuelle permanente. Pour ce faire, ouvrez le fichier *prj.conf* dans le dossier de notre Blinky et ajoutez la ligne (contrairement aux fichiers d'arbre de périphériques qui utilisent la syntaxe du langage C, les fichiers de configuration Kconfig utilisent la syntaxe du langage Python) :

```
CONFIG_LED=y
```

Pour vérifier qu'il fonctionne, exécutez une compilation *pristine* de votre projet et téléchargez l'exécutable sur la carte.

Débogage

Si votre carte le permet (comme la BBC micro:bit) ou si vous disposez d'un outil de débogage approprié, vous pouvez déboguer l'application avec

```
west debug
```

Ceci lancera un serveur gdb et ouvrira un terminal (**figure 12**). Consultez internet pour apprendre à travailler avec gdb car cela sort du cadre de cet article.

Zephyr contre Arduino

Maintenant que vous avez vu comment démarrer avec le système d'exploitation Zephyr, vous vous demandez peut-être pourquoi vous l'utiliseriez. N'est-il pas plus simple d'utiliser Arduino ? Comme Zephyr, Arduino supporte plusieurs architectures de processeurs et des centaines de cartes. Des milliers de pilotes et de bibliothèques ont été écrits pour Arduino. Si une application ou une bibliothèque utilise l'API Arduino Core, elle peut être facilement portée sur n'importe quelle autre plateforme supportée avec des périphériques similaires, tout comme une application Zephyr. Les deux sont des logiciels libres. Et donc ?

Zephyr est conçu comme un RTOS robuste de qualité industrielle, doté de fonctions telles que la planification des tâches, la gestion de la mémoire et les pilotes de périphériques. En outre, Zephyr est conçu pour s'adapter à un large éventail de complexités de projets, des petits appareils IdO aux systèmes embarqués complexes. Il offre une plus grande flexibilité, mais nécessite une compréhension plus approfondie du développement embarqué.

Arduino, bien qu'offrant certaines capacités en temps réel, n'est pas un RTOS mais un cadre de travail pour les applications à un seul fil, axé sur la simplicité et la facilité d'utilisation. Arduino fait abstraction de nombreux détails de bas niveau, ce qui le rend accessible aux débutants. Cependant, il peut être utilisé au-dessus d'un RTOS tel que Mbed OS pour des applications plus complexes. Des travaux visant à faire fonctionner l'API Arduino Core sur Zephyr sont en cours ([7]).

C'est à vous de décider si vous avez besoin de Zephyr pour votre prochain projet ou non. Vous pouvez au moins l'essayer, car il a fière allure sur le CV de tout développeur de systèmes embarqués.

Autres lectures

C'est tout pour l'instant. Comme vous l'aurez constaté, le système d'exploitation Zephyr est compliqué et la courbe d'apprentissage est quelque peu abrupte. Dans cet article, j'ai essayé de rendre ce parcours un peu plus facile. Cependant, il y a beaucoup, beaucoup plus à dire et à apprendre sur Zephyr, alors ne pensez pas maintenant que vous savez tout. Les références [8] et [9] fournissent des liens vers deux sujets qui pourraient vous intéresser.

VF : Maxime Valens — 230713-04

Questions ou commentaires ?

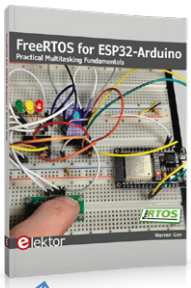
Envoyez un courriel à l'auteur (clemens.valens@elektor.com) ou contactez Elektor (redaction@elektor.fr).

À propos de l'auteur

Après une carrière dans l'électronique marine et industrielle, Clemens Valens a commencé à travailler pour Elektor en 2008 en tant que rédacteur en chef d'Elektor France. Il a occupé différents postes depuis lors et a récemment rejoint le département de développement des produits. Ses principaux centres d'intérêt sont le traitement du signal et la génération de sons.

Produits

- **BBC micro:bit V2**
  www.elektor.fr/19488
- **Get Started with the NXP i.MX RT1010 Development Bundle**
  www.elektor.fr/20699
- **Warren Gay, *FreeRTOS for ESP32-Arduino*, Elektor 2020**
  www.elektor.fr/19341



LIENS

[1] À propos du projet Zephyr - Un projet de la Fondation Linux : https://zephyrproject.org/learn-about
[2] Spécifications de l'arbre des périphériques : https://devicetree.org/specifications
[3] Démarrer avec Zephyr : https://docs.zephyrproject.org/latest/develop/getting_started/index.html
[4] Plugin de mise en évidence de la syntaxe Devicetree pour Visual Code Studio: https://github.com/plorefice/vscode-devicetree
[5] Devicetree API : https://docs.zephyrproject.org/latest/build/dts/api/api.html
[6] Le périphérique LED : https://docs.zephyrproject.org/latest/hardware/peripherals/led.html
[7] Exécuter Arduino sur Zephyr : https://dhruvag2000.github.io/Blog-GSoC22
[8] Voir le livre blanc d'Eli Hughes : « From Hardware Concept to Zephyr Bring Up » : https://zephyrproject.org/white-papers
[9] Sortie de la console série USB dans une application RTOS Zephyr : https://www.gnd.io/zephyr-console-output