guide de programmation Bare-Metal (3)

en-têtes CMSIS, tests automatiques et serveur web

Sergey Lyubka (Irlande)

Dans les deux premières parties de ce guide, nous avons appris à accéder aux broches d'un microcontrôleur, à l'horloge système et à l'UART, et à réaliser nos premiers exemples de micrologiciels avec des scripts de l'éditeur de liens et des fichiers Makefile. Dans cette dernière partie de la série, nous allons nous simplifier encore la tâche grâce à des en-têtes et des bibliothèques prédéfinies. Nous programmerons un serveur web et apprenons à automatiser les constructions et les tests de ces micrologiciels plus complexes.

Note de la rédaction : ce guide est un document vivant sur GitHub [1].

En-têtes CMSIS du fabricant

Dans les articles précédents [2][3], nous avons développé le micrologiciel uniquement avec les fiches techniques, l'éditeur et le compilateur GCC. Nous avons créé les définitions des structures pour les périphériques manuellement, en utilisant les fiches techniques. Maintenant que vous avez une idée générale du fonctionnement, il est temps de présenter les en-têtes CMSIS. Il s'agit de fichiers d'en-tête contenant toutes les définitions, créés et fournis par le fabricant du microcontrôleur. Ils contiennent des définitions pour les blocs internes et les périphériques de ce microcontrôleur, et sont donc assez volumineux.

CMSIS est l'acronyme de *Common Microcontroller Software Interface Standard* (norme d'interface logicielle commune pour les microcontrôleurs). Il s'agit d'une base commune permettant aux fabricants de microcontrôleurs de spécifier les API des périphériques. Étant donné que CMSIS est une norme ARM et que les en-têtes CMSIS sont fournis par le fabricant, ils constituent une source d'autorité. Il est donc préférable de les utiliser plutôt que d'écrire les définitions soi-même.

Il existe deux séries d'en-têtes CMSIS :

- En-têtes CMSIS du cœur ARM. Ils décrivent le cœur ARM et sont publiés par ARM sur GitHub [4]
- En-têtes CMSIS des fabricants de microcontrôleurs. Ils décrivent les périphériques du microcontrôleur et sont publiés par le fabricant du microcontrôleur. Dans notre cas, ST les publie à l'adresse [5]

Nous pouvons extraire ces en-têtes avec un simple extrait de Makefile :

(voir *https://github.com/cpq/bare-metal-programming-guide/blob/main/steps/step-5-cmsis/Makefile*)

```
cmsis_core:
  git clone --depth 1 -b 5.9.0
   https://github.com/ARM-software/CMSIS_5 $@
cmsis_f4:
  git clone --depth 1 -b v2.6.8
   https://github.com/STMicroelectronics/
   cmsis_device_f4 $@
```

Le package CMSIS de ST fournit également des fichiers de démarrage pour tous leurs microcontrôleurs. Nous pouvons les utiliser au lieu d'écrire nous-mêmes le fichier *startup.c*. Le fichier *startup* fourni par ST appelle la fonction `SystemInit()`, nous la définissons donc dans le fichier *main.c*.

Maintenant, remplaçons nos fonctions API dans *hal.h* par des définitions CMSIS, et laissons le reste du micrologiciel intact. Dans *hal.h*, supprimez toutes les API et définitions de périphériques, et ne laissez que les *includes* C standard, l'inclusion CMSIS du fabricant, les définitions de PIN, BIT, FREQ, et la fonction d'aide `timer_expired()`.

Si nous essayons de reconstruire le micrologiciel – `make clean build`, GCC échouera, signalant l'absence de `systick_init()`, `GPIO_MODE_OUTPUT`, `uart_init()`, et UART3. Ajoutons ces éléments, en utilisant les fichiers CMSIS de STM32.

Commençons par `systick_init()`. Les en-têtes CMSIS d'ARM fournissent une fonction `SysTick_Config()` qui effectue la même tâche, nous allons donc l'utiliser.

Vient ensuite la fonction `gpio_set_mode()`. L'en-tête *stm32f429xx.h* contient une structure `GPIO_TypeDef`, identique à `struct gpio`. Utilisons-la :

(voir *https://github.com/cpq/bare-metal-programming-guide/blob/main/steps/step-5-cmsis/hal.h*)

```
#define GPIO(bank) ((GPIO_TypeDef *)
  (GPIOA_BASE + 0x400U * (bank)))
enum { GPIO_MODE_INPUT, GPIO_MODE_OUTPUT,
   GPIO_MODE_AF, GPIO_MODE_ANALOG };

static inline void gpio_set_mode
      (uint16_t pin, uint8_t mode) {
  GPIO_TypeDef *gpio =
   GPIO(PINBANK(pin));  // GPIO bank
```

Les fonctions `gpio_set_af()` et `gpio_write()` sont également simples – il suffit de remplacer `struct gpio` par `GPIO_TypeDef`. Vient ensuite l'UART. Il y a un `USART_TypeDef`, et des définitions pour USART1, USART2, USART3. Utilisons-les :

```
#define UART1 USART1
#define UART2 USART2
#define UART3 USART3
```

Dans `uart_init()` et le reste des fonctions UART, remplacez `struct uart` par `USART_TypeDef`. Le reste est inchangé !
Et voilà, c'est fait. Reconstruisez et reflashez le micrologiciel. La LED clignote, l'UART affiche la sortie. Félicitations, nous avons adapté le code du micrologiciel pour utiliser les fichiers d'en-tête CMSIS du fabricant. Maintenant, réorganisons un peu le dépôt en déplaçant tous les fichiers standards dans le répertoire *include* et en mettant à jour *Makefile* pour que GCC en soit averti :

(voir *https://github.com/cpq/bare-metal-programming-guide/blob/main/steps/step-5-cmsis/Makefile*)

```
-I. -Iinclude -Icmsis_core/CMSIS/Core/Include -Icmsis_
f4/Include \
```

Incluons également l'en-tête CMSIS en tant que dépendance pour le fichier binaire :

```
firmware.elf: cmsis_core cmsis_f4 mcu.h
      link.ld Makefile $(SOURCES)
```

Nous nous retrouvons avec un exemple de projet réutilisable. Vous pouvez trouver le code source complet du projet dans le répertoire *step-5-cmsis* du projet [6].

Réglage des horloges

Après le démarrage, le processeur Nucleo-F429ZI fonctionne à 16 MHz. La fréquence maximale est de 180 MHz. Notez que la fréquence de l'horloge système n'est pas le seul facteur à prendre en compte. Les périphériques sont connectés à différents bus, APB1 et APB2, qui sont cadencés différemment. Leurs vitesses d'horloge sont configurées par les valeurs du prédiviseur de fréquence (*prescaler*), définies dans le contrôleur RCC (RCC gère les horloges du système et des périphériques). La source d'horloge principale du CPU peut également être différente – nous pouvons utiliser soit un oscillateur à cristal externe (HSE), soit un oscillateur interne (HSI). Dans notre exemple, nous utiliserons l'HSI.
Lorsque le CPU exécute des instructions à partir de la mémoire flash, la vitesse de lecture (environ 25 MHz) devient un goulot d'étranglement si l'horloge du CPU est plus rapide. Plusieurs astuces peuvent aider. La préfixation des instructions en est une. Nous pouvons également donner un indice au contrôleur de la mémoire flash quant à la vitesse de l'horloge système : cette valeur est appelée `FLASH_LATENCY`. Pour une horloge système de 180 MHz, la valeur `FLASH_LATENCY` est de 5. Les bits 8 et 9 du contrôleur de la flash activent les caches d'instructions et de données :

```
FLASH->ACR |= FLASH_LATENCY | BIT(8) |
    BIT(9); // Flash latency, caches
```

La source d'horloge (HSI ou HSE) utilise un matériel appelé PLL, qui multiplie la fréquence de la source par une certaine valeur. Ensuite, un ensemble de diviseurs de fréquence est utilisé pour régler l'horloge système et les horloges APB1 et APB2. Afin d'obtenir une valeur maximale de l'horloge système (180 MHz), plusieurs valeurs de diviseurs PLL et de prescaler APB sont possibles. La section 6.3.3 du manuel de référence du contrôleur STM32F4xx [7] nous indique les valeurs maximales pour l'horloge APB1 : ≤ 45 MHz, et l'horloge APB2 : ≤ 90 MHz. Cela réduit la liste des combinaisons possibles. Ici, nous avons choisi les valeurs manuellement. Notez que des outils comme CubeMX peuvent automatiser le processus et le rendre simple et visuel.

(voir *https://github.com/cpq/bare-metal-programming-guide/blob/main/steps/step-6-clock/hal.h*)

```
 // 6.3.3: APB1 clock <= 45MHz;
 //       APB2 clock <= 90MHz
 // 3.5.1, Table 11: configure flash
// latency (WS) in accordance to clock freq
 // 33.4: The AHB clock must be at least
 // 25 MHz when Ethernet is used
 enum { APB1_PRE = 5 /* AHB clock / 4 */,
   APB2_PRE = 4 /* AHB clock / 2 */ };
 enum { PLL_HSI = 16, PLL_M = 8,
   PLL_N = 180, PLL_P = 2 };
     // Run at 180 Mhz
 #define FLASH_LATENCY 5
 #define SYS_FREQUENCY ((PLL_HSI * PLL_N /
       PLL_M / PLL_P) * 1000000)
 #define APB2_FREQUENCY
   (SYS_FREQUENCY / (BIT(APB2_PRE - 3)))
 #define APB1_FREQUENCY
   (SYS_FREQUENCY / (BIT(APB1_PRE - 3)))
```

Nous sommes maintenant prêts à utiliser un algorithme simple pour régler l'horloge des bus et des périphériques. Il peut ressembler à ceci :

- Optionnellement, activer le FPU
- Définir la latence de la flash
- Choisir une source d'horloge, une PLL et des prescalers APB1 et APB2
- Configurer le RCC en fixant les valeurs respectives
- Déplacer l'initialisation de l'horloge dans un fichier séparé *sysinit.c* ; la fonction `SystemInit()` qui est automatiquement appelée par le code de démarrage

Voir **listage 1** !

Listage 1. Initialisation de l'horloge.

[voir https://github.com/cpq/bare-metal-programming-guide/blob/main/steps/step-6-clock/sysinit.c]

```
uint32_t SystemCoreClock = SYS_FREQUENCY;

 void SystemInit(void) {  // Called automatically by startup code
   SCB->CPACR |= ((3UL << 10 * 2) | (3UL << 11 * 2));  // Enable FPU
   FLASH->ACR |= FLASH_LATENCY | BIT(8) | BIT(9);      // Flash latency, prefetch
   RCC->PLLCFGR &= ~((BIT(17) - 1));                   // Clear PLL multipliers
   RCC->PLLCFGR |= (((PLL_P - 2) / 2) & 3) << 16;      // Set PLL_P
   RCC->PLLCFGR |= PLL_M | (PLL_N << 6);               // Set PLL_M and PLL_N
   RCC->CR |= BIT(24);                                 // Enable PLL
   while ((RCC->CR & BIT(25)) == 0) spin(1);           // Wait until done
   RCC->CFGR = (APB1_PRE << 10) | (APB2_PRE << 13);    // Set prescalers
   RCC->CFGR |= 2;                                     // Set clock source to PLL
   while ((RCC->CFGR & 12) == 0) spin(1);              // Wait until done

   RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN;    // Enable SYSCFG
   SysTick_Config(SystemCoreClock / 1000);  // Sys tick every 1ms
 }
```

Nous devons également modifier *hal.h* – en particulier le code d'initialisation de l'UART. Les différents contrôleurs UART fonctionnent sur des bus différents : UART1 fonctionne sur un APB2 rapide, et le reste des UART fonctionne sur un APB1 plus lent. À une vitesse par défaut de 16 MHz, rien ne change. Mais, lorsqu'ils fonctionnent à des vitesses plus élevées, APB1 et APB2 peuvent avoir des horloges différentes, nous devons donc adapter le calcul du débit en bauds pour l'UART. Voir **listage 2**.
Reconstruisez et re-flashez, et notre carte fonctionnera à sa vitesse maximale, 180 MHz ! Le code source complet du projet peut être trouvé dans le répertoire *step-6-clock* du projet [8].

Serveur web avec tableau de bord

Le Nucleo-F429ZI est équipé d'Ethernet. Le matériel Ethernet nécessite deux composants : un PHY (qui transmet/reçoit les signaux électriques vers et au média, tel que le cuivre, le câble optique, etc.) et un MAC (qui pilote le contrôleur PHY). Sur la carte Nucleo, le contrôleur MAC est intégré et le PHY est externe (plus précisément, il s'agit du LAN8720a de Microchip).
MAC et PHY peuvent utiliser plusieurs interfaces. Nous utiliserons RMII. Pour cela, nous devons configurer un certain nombre de broches pour qu'elles utilisent leur fonction alternative (AF). Pour implémenter un serveur web, nous avons besoin de trois composants logiciels :

- un pilote réseau, qui envoie/reçoit des trames vers/depuis le contrôleur MAC via Ethernet
- une pile de réseau, qui analyse les trames et interprète TCP/IP
- une bibliothèque réseau qui interprète le protocole HTTP

Nous utiliserons la bibliothèque réseau Mongoose [9] qui implémente tout cela dans un seul fichier. Il s'agit d'une bibliothèque à double licence (*GPLv2/commercial*) conçue pour rendre le développement de réseaux intégrés rapide et facile.
Copiez donc *mongoose.c* [10] et *mongoose.h* [11] dans votre projet. Nous disposons désormais d'un pilote, d'une pile réseau et d'une bibliothèque. Mongoose fournit également un grand nombre d'exemples, dont un exemple de tableau de bord [12]. Il met en œuvre de nombreuses fonctions, telles que la connexion au tableau de bord, l'échange de données en temps réel via WebSocket, un système de fichiers intégré, la communication MQTT, etc. Utilisons donc cet exemple. Copiez deux fichiers supplémentaires :

- *net.c* [13] — implémente les fonctions du tableau de bord
- *packed_fs.c* [14] — contient des fichiers GUI HTML/CSS/JS

Nous devons indiquer à Mongoose les fonctionnalités à activer. Cela peut se faire via les drapeaux de compilation (*flags*), en définissant des constantes de préprocesseur. Alternativement, les mêmes constantes peuvent être définies dans le fichier *mongoose_custom.h*. Adoptons la seconde solution. Créez un fichier *mongoose_custom.h* contenant le code suivant :

```
#pragma once
#define MG_ARCH MG_ARCH_NEWLIB
#define MG_ENABLE_MIP 1
#define MG_ENABLE_PACKED_FS 1
#define MG_IO_SIZE 512
#define MG_ENABLE_CUSTOM_MILLIS 1
```

Il est temps d'ajouter du code réseau à *main.c*. Nous incluons `#include «mongoose.c»`, initialisons les broches RMII Ethernet et activons Ethernet dans le RCC. Voir **listage 3**.
Le pilote de Mongoose utilise l'interruption Ethernet, nous devons donc mettre à jour le fichier *startup.c* et ajouter *ETH_IRQHandler* à la table vectorielle. Réorganisons la définition de la table vectorielle dans *startup.c* pour ce qu'aucune modification ne soit nécessaire pour ajouter une fonction de gestion des interruptions. L'idée est d'utiliser un concept de "symbole faible".
Une fonction peut être marquée comme *faible* et fonctionner normalement. La différence consiste à définir une fonction portant le même nom ailleurs dans le code source. Normalement, deux

Listage 2. Initialisation de l'UART.

[voir https://github.com/cpq/bare-metal-programming-guide/blob/main/steps/step-6-clock/hal.h]

```
static inline bool uart_init(USART_TypeDef *uart, unsigned long baud) {

  // https://www.st.com/resource/en/datasheet/stm32f429zi.pdf
  uint8_t af = 7;           // Alternate function
  uint16_t rx = 0, tx = 0;  // pins
  uint32_t freq = 0;        // Bus frequency. UART1 is on APB2, rest on APB1

  if (uart == USART1) {
    freq = APB2_FREQUENCY, RCC->APB2ENR |= BIT(4);
    tx = PIN('A', 9), rx = PIN('A', 10);
  } else if (uart == USART2) {
    freq = APB1_FREQUENCY, RCC->APB1ENR |= BIT(17);
    tx = PIN('A', 2), rx = PIN('A', 3);
  } else if (uart == USART3) {
    freq = APB1_FREQUENCY, RCC->APB1ENR |= BIT(18);
    tx = PIN('D', 8), rx = PIN('D', 9);
  } else {
    return false;
  }
```

Listage 3. Initialisation de l'Ethernet, activation des broches MAC GPIO.

[voir https://github.com/cpq/bare-metal-programming-guide/blob/main/steps/step-7-webserver/nucleo-f429zi/main.c]

```
uint16_t pins[] = ;
  for (size_t i = 0; i < sizeof(pins) / sizeof(pins[0]); i++) {
    gpio_init(pins[i], GPIO_MODE_AF, GPIO_OTYPE_PUSH_PULL, GPIO_SPEED_INSANE,
              GPIO_PULL_NONE, 11);
  }
  nvic_enable_irq(61);                          // Setup Ethernet IRQ handler
  RCC->APB2ENR |= BIT(14);                      // Enable SYSCFG
  SYSCFG->PMC |= BIT(23);                       // Use RMII. Goes first!
  RCC->AHB1ENR |= BIT(25) | BIT(26) | BIT(27);  // Enable Ethernet clocks
  RCC->AHB1RSTR |= BIT(25);                     // ETHMAC force reset
  RCC->AHB1RSTR &= ~BIT(25);                    // ETHMAC release reset
```

fonctions portant le même nom font échouer la compilation. Cependant, si une fonction est marquée comme faible, la compilation réussit et l'éditeur de liens sélectionne une fonction non faible. Cela permet de fournir une fonction "par défaut" dans un modèle de code, avec la possibilité de l'ignorer en créant simplement une fonction portant le même nom ailleurs dans le code.

Voici comment cela fonctionne dans notre exemple. Nous souhaitons remplir une table vectorielle avec des gestionnaires par défaut, mais donner à l'utilisateur la possibilité de remplacer n'importe quel gestionnaire. Pour cela, nous créons une fonction `DefaultIRQHandler()` et la marquons comme faible. Ensuite, pour chaque gestionnaire d'IRQ, nous déclarons un nom de gestionnaire et en faisons un alias de la fonction `DefaultIRQHandler()` :

```
void __attribute__((weak)) DefaultIRQHandler(void) {
for (;;) (void) 0;
}
#define WEAK_ALIAS
  __attribute__((weak, alias("DefaultIRQHandler")))
WEAK_ALIAS void NMI_Handler(void);
WEAK_ALIAS void HardFault_Handler(void);
WEAK_ALIAS void MemManage_Handler(void);
...
__attribute__((section(".vectors")))
  void (*tab[16 + 91])(void) =
    { 0, _reset, NMI_Handler,
     HardFault_Handler, MemManage_Handler,
    ...
```

Listage 4. Initialisation de la bibliothèque Mongoose.

```
struct mg_mgr mgr;        // Initialise Mongoose event manager
mg_mgr_init(&mgr);        // and attach it to the MIP interface
mg_log_set(MG_LL_DEBUG);  // Set log level
struct mip_driver_stm32 driver_data = {.mdc_cr = 4};  // See driver_stm32.h
struct mip_if mif = {
     .mac {2, 0, 1, 2, 3, 5}
     .use_dhcp = true,
     .driver = &mip_driver_stm32,
     .driver_data = &driver_data,
};
mip_init(&mgr, &mif);
extern void device_dashboard_fn(struct mg_connection *, int, void *, void *);
mg_http_listen(&mgr, "http://0.0.0.0", device_dashboard_fn, &mgr);
MG_INFO(("Init done, starting main loop"));
```

Listage 5. Makefile avec les références des bibliothèques.

```
847 3 mongoose.c:6784:arp_cache_add     ARP cache: added 0xc0a80001 @
90:5c:44:55:19:8b
 84e 2 mongoose.c:6817:onstatechange     READY, IP: 192.168.0.24
 854 2 mongoose.c:6818:onstatechange            GW: 192.168.0.1
 859 2 mongoose.c:6819:onstatechange            Lease: 86363 sec
 LED: 1, tick: 2262
 LED: 0, tick: 2512
```

Maintenant, nous pouvons définir le gestionnaire d'IRQ de notre choix dans notre code, et il remplacera le gestionnaire par défaut. C'est ce qui se passe dans notre exemple : il existe une fonction `ETH_IRQHandler()` définie par le pilote STM32 de Mongoose, qui remplace le gestionnaire par défaut.

L'étape suivante consiste à initialiser la bibliothèque Mongoose : créez un gestionnaire d'événements, configurez le pilote réseau et démarrez une connexion HTTP en écoute. Voir **listage 4**.

Il ne reste plus qu'à ajouter un appel à `mg_mgr_poll()` dans la boucle *main*.

Maintenant, ajoutez les fichiers *mongoose.c*, *net.c*, et *packed_fs.c* au Makefile. Reconstruisez et re-flashez la carte. Connectez une console série à la sortie de débogueur et observez la carte obtenir une adresse IP par DHCP. Voir **listage 5**.

Ouvrez un navigateur à cette adresse IP, et vous obtenez un tableau de bord fonctionnel, avec un graphique en temps réel via WebSocket, MQTT, l'authentification, et d'autres choses encore ! Voir la description complète pour plus de détails.

Le code source complet du projet se trouve dans le répertoire *step-7-webserver*. [15].

Constructions automatisées du micrologiciel (Software CI)

Avoir un test d'intégration continue (CI) est une bonne pratique dans tout projet de programmation. À chaque modification apportée au répertoire, l'intégration continue reconstruit et teste automatiquement tous les composants.

GitHub facilite cette tâche. Nous pouvons créer un fichier de configuration *CI .github/workflows/test.yml*. Dans ce fichier, nous pouvons installer le GCC d'ARM et exécuter `make` dans chaque répertoire d'exemple pour construire les fichiers respectifs du micrologiciel. En bref, cela indique à GitHub d'exécuter chaque répertoire lors d'un `push` :

(voir *https://github.com/cpq/bare-metal-programming-guide/blob/main/.github/workflows/test.yml*)

```
name: build
on: [push, pull_request]
```

Ceci installe le compilateur GCC d'ARM :

```
- run: sudo apt -y install
  gcc-arm-none-eabi make stlink-tools
```

Cela permet de construire le micrologiciel dans chaque répertoire :

```
- run: make -C step-0-minimal
- run: make -C step-1-blinky
- run: make -C step-2-systick
- run: make -C step-3-uart
- run: make -C step-4-printf
- run: make -C step-5-cmsis
- run: make -C step-6-clock
- run: make -C step-7-webserver/nucleo-f429zi
- run: make -C step-7-webserver/pico-w
```

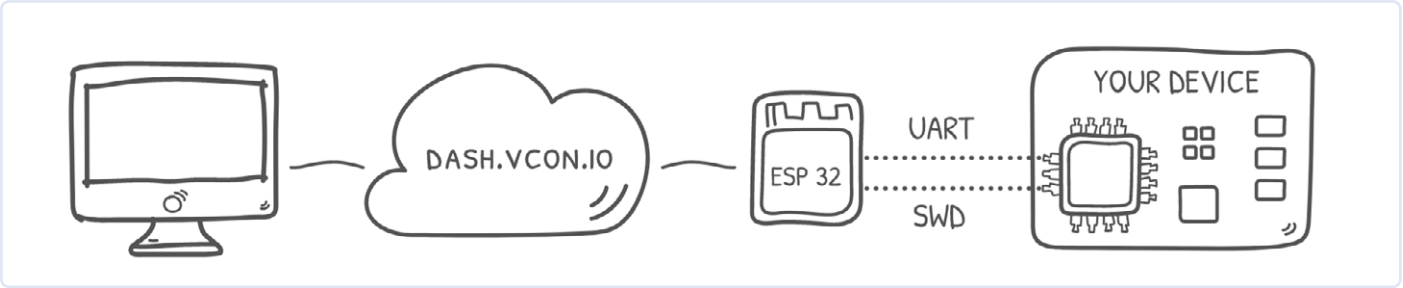


Figure 1. Configuration de l'ESP32 en tant que programmateur contrôlé à distance et inscription sur vcon.io.

C'est tout ! C'est très simple et très puissant. Maintenant, si nous apportons une modification au répertoire qui interrompt une construction, GitHub nous en informera. En cas de succès, GitHub ne fera rien. Voici un exemple d'exécution réussie [16].

Tests automatisés des micrologiciels (Hardware CI)

Serait-il possible de tester les fichiers binaires du micrologiciel construit sur un matériel réel, pour assurer que le micrologiciel construit est correct et fonctionnel ?
La construction d'un tel système ad hoc n'est pas simple. Par exemple, on peut installer une poste de travail de test dédié, y connecter un dispositif de test (par exemple une carte Nucleo-F429ZI), écrire un logiciel pour télécharger un micrologiciel à distance, et effectuer des tests avec un débogueur intégré. Cette méthode est possible, mais délicate, et demande beaucoup d'efforts et d'attention.
L'autre solution consiste à utiliser l'un des systèmes commerciaux de test de matériel, les *Embedded Eoard Farms* (EBF), bien que ces solutions commerciales soient assez coûteuses. Mais il existe un moyen plus simple.

Solution : ESP32 + vcon.io

Utilisons le service *http://vcon.io* service, qui permet la mise à jour à distance du micrologiciel et la surveillance de l'UART :

- Utilisez un ESP32 ou ESP32-C3 (une carte de développement abordable).
- Flasher un micrologiciel pré-construit, transformant ainsi l'ESP32 en un programmateur contrôlé à distance.
- Connectez l'ESP32 à l'appareil cible : broches SWD pour le clignotement, broches UART pour la lecture des données de sortie.
- Configurez l'ESP32 pour qu'il s'enregistre sur le tableau de bord de gestion *vcon.io* [17].

Lorsque cela est fait, votre appareil cible aura une API RESTful authentifiée et sécurisée pour reflasher et recueillir les données de sortie de l'appareil. Elle peut être appelée depuis n'importe où, par exemple depuis le logiciel CI (voir **figure 1**).

Note : le service *vcon.io* est géré par Cesanta - la société pour laquelle je travaille. Il s'agit d'un service payant avec un quota de gratuité : si vous n'avez que quelques appareils à gérer, le service est entièrement gratuit.

Configuration et câblage de l'ESP32

Utilisez n'importe quel ESP32 ou ESP32-C3 - une carte de développement, un module, ou votre appareil personnalisé. Je recommande la carte ESP32-C3 XIAO pour son prix abordable et sa petite taille. Nous allons supposer que l'appareil cible est une carte Raspberry Pi W5500-EVB-Pico [18] avec une interface Ethernet intégrée. Si votre appareil est différent, modifiez le câblage en fonction du brochage.

- Suivez les instructions [19] pour flasher votre ESP32.
- Configurez le réseau [20] pour enregistrer l'ESP32 sur le tableau de bord
- Suivez la section *Câblage* [21] pour connecter l'ESP32 à votre appareil.

La **figure 2** montre à quoi ressemble la configuration sur une plaque d'essai. La **figure 3** montre à quoi ressemble le tableau de bord d'un appareil configuré.
Vous pouvez maintenant reflasher votre appareil avec une seule commande :

```
curl -su :API_KEY https://dash.vcon.io/api/v3/devices/
ID/ota --data-binary @firmware.bin
```

où `API_KEY` est la clé d'authentification sur *dash.vcon.io*, `ID` est le numéro de l'appareil enregistré, et *firmware.bin* est le nom du nouveau micrologiciel. Vous pouvez obtenir la clé API_KEY en cliquant sur le lien *api key* dans un tableau de bord. L'ID de l'appareil est indiqué dans le tableau.
Nous pouvons également capturer la sortie d'un appareil avec une seule commande :

```
curl -su :API_KEY https://dash.vcon.io/api/v3/devices/
ID/tx?t=5
```

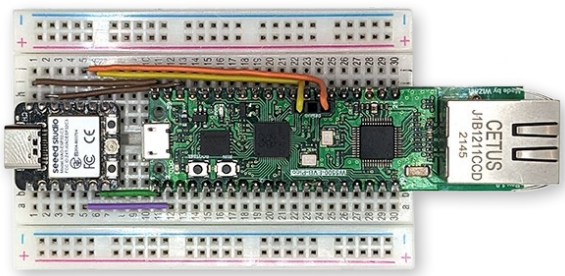


Figure 2. Configuration de l'appareil sur une plaque d'essai.

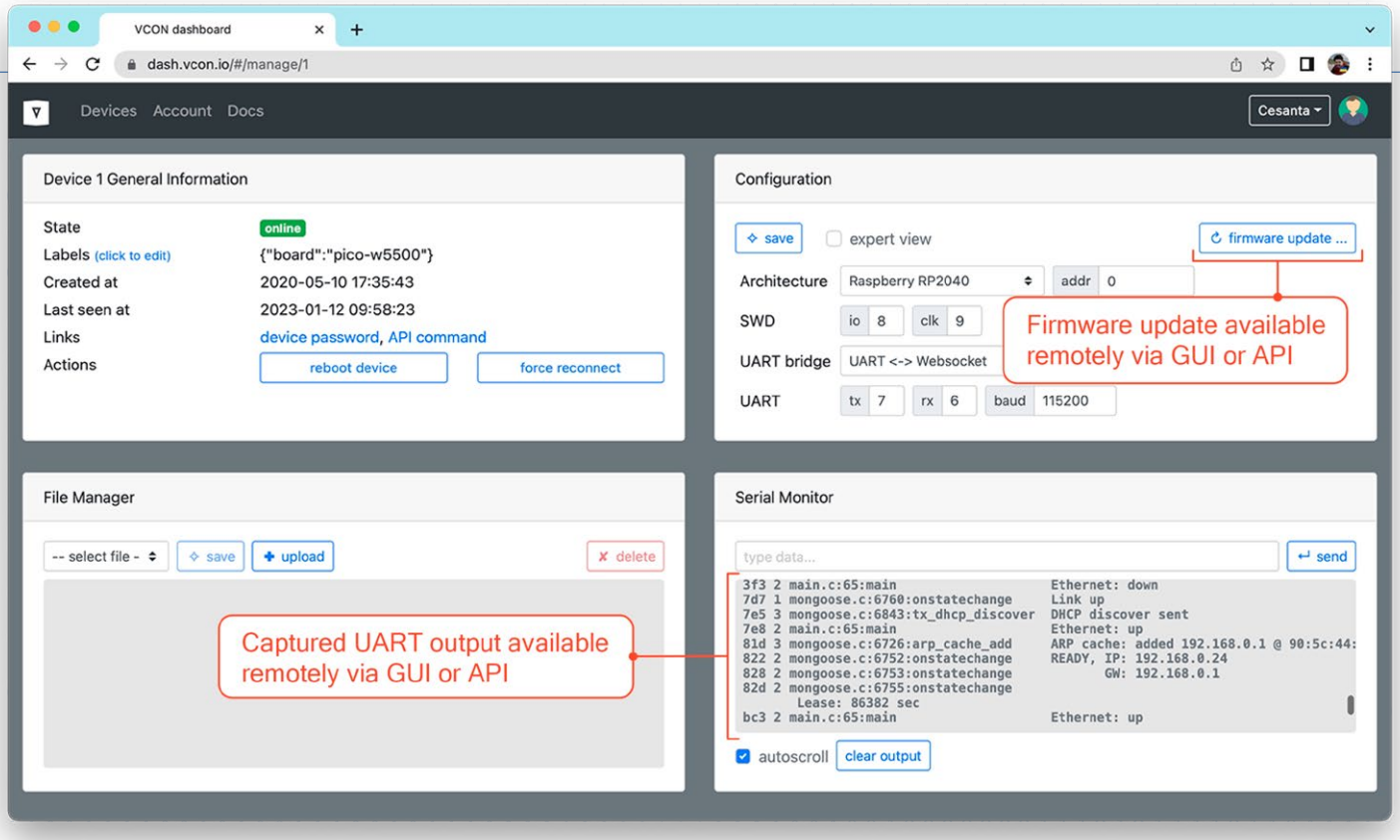


Figure 3. Tableau de bord de l'appareil configuré.

Ici, `t=5` signifie : attendre 5 secondes pendant la capture du signal de sortie de l'UART.
Maintenant, nous pouvons utiliser ces deux commandes dans n'importe quelle plateforme CI logicielle pour tester un nouveau micrologiciel sur un appareil réel, et tester la sortie UART de l'appareil par rapport à certains mots-clés prévus.

Intégration avec les actions de GitHub

Le logiciel CI construit une image du micrologiciel, et maintenant nous pouvons même la tester sur un vrai matériel ! Nous devrions ajouter quelques commandes supplémentaires qui utilisent l'utilitaire `curl` pour envoyer le micrologiciel construit à la carte de test, et ensuite capturer sa sortie de débogage.
La commande `curl` nécessite une clé API secrète, que nous ne voulons pas exposer au public. En général, la bonne façon de procéder est la suivante :

- Allez dans les paramètres de votre projet sur GitHub (vous pouvez cloner le dépôt avec les exemples du serveur web), puis allez sur / *Secrets* / *Actions*
- Cliquez sur le bouton *New repository secret*
- Nommez-le `VCON_API_KEY`, collez la valeur dans une case *Secret*, cliquez sur *Add secret*

L'un des projets d'exemple [22] construit un micrologiciel pour la carte RP2040-W5500, alors flashons-le avec une commande `curl` et une clé API sauvegardée. La meilleure façon est d'ajouter une cible de Makefile pour les tests, et de laisser *GitHub Actions* (logiciel *CI*) l'appeler :

(voir *https://github.com/cpq/bare-metal-programming-guide/blob/main/.github/workflows/test.yml*)

```
- run: make -C step-7-webserver/pico-w5500 test VCON_API_KEY=${{secrets.VCON_API_KEY}}
```

Notez que nous donnons une variable d'environnement `VCON_API_KEY` à `make`. Notez également que nous invoquons la cible `test` de Makefile, qui devrait construire et tester le micrologiciel. Dans le **listage 6**, vous pouvez voir la cible `test` de Makefile.

Explication :

- ligne 34 : la cible `test` dépend de la cible du téléversement, c'est pourquoi le téléversement est exécuté en premier. (voir ligne 38)
- ligne 35 : lit l'UART pendant 5 secondes et l'enregistre dans */tmp/output.txt*
- ligne 36 : cherche la chaîne de caractères `Ethernet : up` dans la sortie, et échoue si elle n'est pas trouvée.
- ligne 38 : la cible de `upload` dépend de la version, c'est pourquoi `build` s'exécute avant de tester le micrologiciel.
- ligne 39 : nous flashons le micrologiciel à distance. Le drapeau `--fail` de l'utilitaire `curl` le fait échouer si la réponse du serveur n'est pas réussie (not HTTP 200 OK).

Dans le **listage 7**, vous pouvez trouver l'exemple de sortie de la commande `make test` décrite ci-dessus.
C'est fait ! Maintenant, nos tests automatiques assurent que le micrologiciel peut être construit, qu'il est bootable, et qu'il initialise la pile réseau correctement. On peut facilement développer cette méthode : il suffit d'ajouter des actions plus complexes dans les fichiers binaires de votre micrologiciel, d'imprimer le résultat via l'UART, et de vérifier la sortie attendue au cours du test.

Bons tests !

220665-C-04

Listage 6. Makefile pour le serveur web sur le Pico.

[voir https://github.com/cpq/bare-metal-programming-guide/blob/main/steps/step-7-webserver/pico-w5500/Makefile]

```
32 # Requires env variable VCON_API_KEY set
33 DEVICE_URL ?= https://dash.vcon.io/api/v3/devices/1
34 test: update
35   curl --fail -su :$(VCON_API_KEY) $(DEVICE_URL)/tx?t=5 | tee /tmp/output.txt
36   grep 'Ethernet: up' /tmp/output.txt

38 update: build
39   curl --fail -su :$(VCON_API_KEY) $(DEVICE_URL)/ota --data-binary @firmware.bin
```

Listage 7. Sortie de la commande Make Test.

```
$ make test
 curl --fail ...
 {"success":true,"written":59904}
 curl --fail ...
 3f3 2 main.c:65:main                    Ethernet: down
 7d7 1 mongoose.c:6760:onstatechange     Link up
 7e5 3 mongoose.c:6843:tx_dhcp_discover  DHCP discover sent
 7e8 2 main.c:65:main                    Ethernet: up
 81d 3 mongoose.c:6726:arp_cache_add     ARP cache: added 192.168.0.1 @
90:5c:44:55:19:8b
 822 2 mongoose.c:6752:onstatechange     READY, IP: 192.168.0.24
 827 2 mongoose.c:6753:onstatechange            GW: 192.168.0.1
 82d 2 mongoose.c:6755:onstatechange            Lease: 86336 sec
 bc3 2 main.c:65:main                    Ethernet: up
 fab 2 main.c:65:main                    Ethernet: up
```

Questions ou commentaires ?

Envoyez un courriel à l'auteur (sergey.lyubka@cesanta.com) ou contactez Elektor (redaction@elektor.fr).

À propos de l'auteur

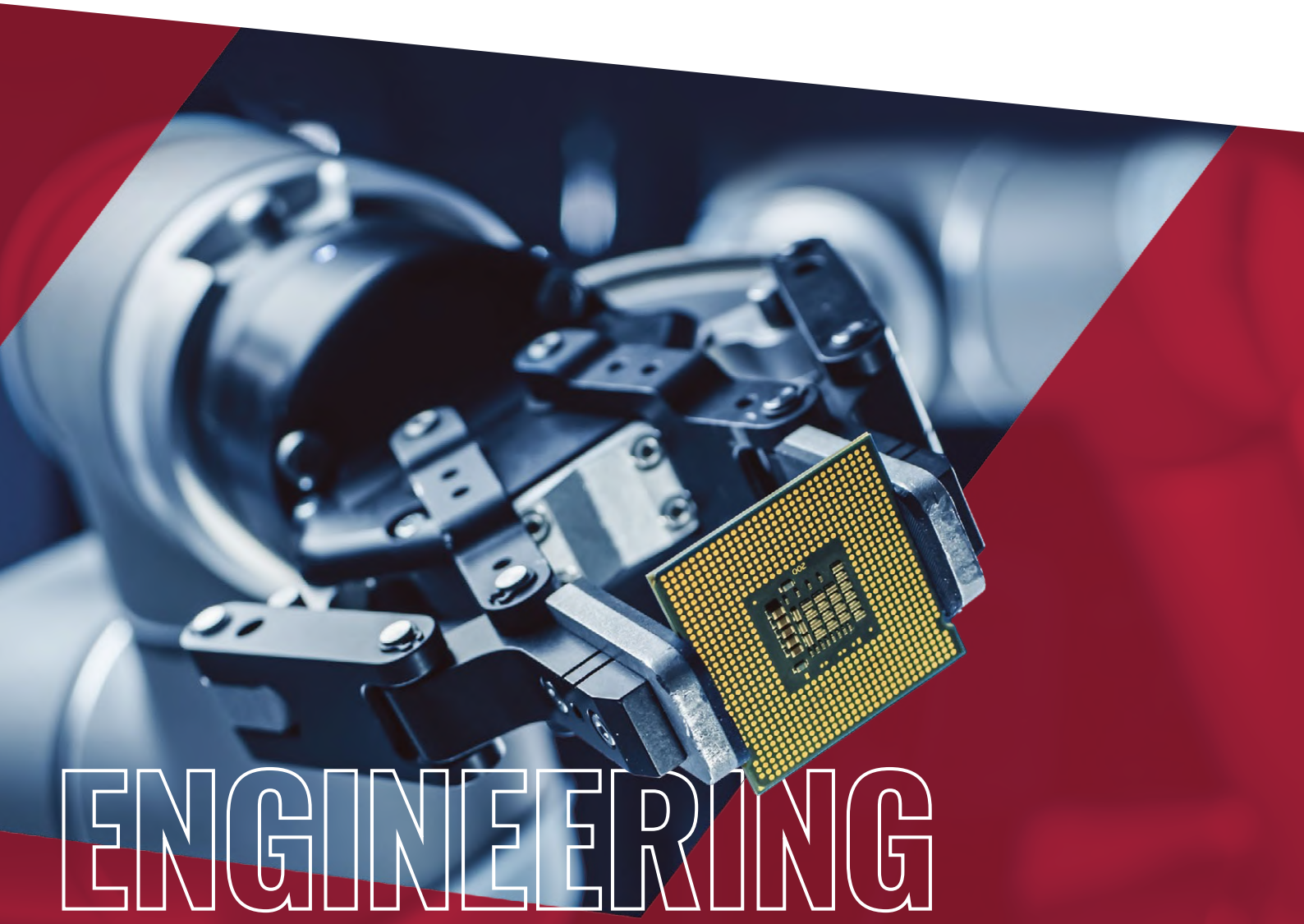
Sergey Lyubka est ingénieur et entrepreneur. Il est titulaire d'un Master en physique de l'université d'État de Kiev, en Ukraine. Sergey est directeur et cofondateur de Cesanta, une entreprise technologique basée à Dublin, en Irlande (Embedded Web Server for electronic devices : https://mongoose.ws). Il est passionné par la programmation embarquée « bare-metal » de réseaux

Produits

- **Dogan Ibrahim,** *Nucleo Boards Programming with the STM32CubeIDE* **(Elektor 2020)**
  www.elektor.fr/19530
- **Dogan Ibrahim,** *Programming with STM32 Nucleo Boards* **(Elektor 2015)**
  www.elektor.com/18585

LIENS

[1] Ce guide sur GitHub : https://github.com/cpq/bare-metal-programming-guide
[2] Sergey Lyubka, « Guide de programmation bare-metal (1) » Elektor 7-8/2023 : https://elektormagazine.fr/220665-04
[3] Sergey Lyubka, « guide de programmation bare-metal (2) » Elektor 9-10/2023 : https://elektormagazine.fr/220665-B-02
[4] CMSIS version 5 : https://github.com/ARM-software/CMSIS_5
[5] En-têtes STM32 CMSIS pour la famille F4 : https://github.com/STMicroelectronics/cmsis_device_f4
[6] Step 5 CMSIS folder : https://github.com/cpq/bare-metal-programming-guide/tree/main/steps/step-5-cmsis
[7] Manuel de référence RM0090 pour les contrôleurs STM32F4xx [PDF] : https://tinyurl.com/stm32f4man
[8] Step 6 Clock Folder : https://github.com/cpq/bare-metal-programming-guide/tree/main/steps/step-6-clock
[9] Bibliothèque du réseau Mongoose : https://github.com/cesanta/mongoose
[10] mongoose.c : https://raw.githubusercontent.com/cesanta/mongoose/master/mongoose.c
[11] mongoose.h : https://raw.githubusercontent.com/cesanta/mongoose/master/mongoose.h
[12] Exemples de Mongoose : https://github.com/cesanta/mongoose/tree/master/examples/device-dashboard
[13] Exemple de serveur web, net.c : https://raw.githubusercontent.com/cesanta/mongoose/master/examples/device-dashboard/net.c
[14] Exemple de serveur web, packed_fs.c : https://tinyurl.com/packedfsc
[15] Step 7 Webserver directory : https://github.com/cpq/bare-metal-programming-guide/tree/main/steps/step-7-webserver
[16] Example: Successful run of Continuous Integration : https://tinyurl.com/bmgoodrun
[17] vcon.io: https://dash.vcon.io/
[18] Carte Raspberry Pi W5500-EVB-Pico : https://docs.wiznet.io/Product/iEthernet/W5500/w5500-evb-pico
[19] Flashage de l'ESP32 : https://vcon.io/docs/#module-flashing
[20] Configuration du réseau : https://vcon.io/docs/#module-registration
[21] Câblage : https://vcon.io/docs/#module-to-device-wiring
[22] Exemple de serveur web pour la carte Pico : https://tinyurl.com/picowebeg



ENGINEERING A BETTER WORLD

IMAGINE THAT. WE CAN.

Let us take you on a journey of discovery. Whether you design, build or maintain, we have ingenious technologies and solutions to help engineer a better world for everyone.

LoRa, un couteau suisse (2)

le matériel et le logiciel

Gilles Brocard (France)

Après avoir abordé le protocole LoRa dans la première partie, nous allons entrer dans le détail de la réalisation d'un émetteur-récepteur LoRa à partir d'un petit module du commerce. Piloté par un Arduino Nano et un programme entièrement dédié au paramétrage et à l'émission/réception LoRa, le montage final constitue un véritable couteau suisse LoRa.

Le petit circuit imprimé décrit ici réuni un module LoRa, une carte Arduino Nano, un régulateur 5 V, quelques LED et des bornes à vis pour connecter aisément toute sorte de capteurs, de commandes, un écran OLED I²C ou des LED supplémentaires. Ce montage est d'une grande flexibilité, due en partie à son programme. Il constitue un véritable couteau suisse LoRa.

Le schéma (**figure 1**) se résume aux liaisons entre l'Arduino Nano et le module E220-900M30S avec un régulateur de tension et une LED confirmant la présence du +5 V et deux autres LED indiquant une émission (TX) ou une réception (RX).

K2 permet de connecter d'autres composants au circuit. Les ports A0 à A5 sont des entrées analogiques ou E/S numériques suivant le paramétrage. En plus, A4 et A5 peuvent être configurés en bus I²C pour piloter des capteurs en I²C ou encore un petit afficheur OLED 128 × 64 pixels. A6 et A7 sont deux entrées uniquement analogiques. Les entrées analogiques du Nano ont une résolution de 10 bits. La borne +5 V peut fonctionner en entrée ou en sortie.

La sortie DIO2 du module LoRa est reliée à l'entrée TX de l'aiguillage d'antenne et à la grille de T1, qui sert d'inverseur logique. Son drain commande l'entrée RX de ce même aiguilleur d'antenne.

LED1 (blanche) signale une émission (TX). La LED2 (verte) indique la réception (RX) d'un signal LoRa valide, mais sans renseignement sur son contenu. En complément, le programme active pendant une seconde le port A2 si le message attendu est détecté. Cela permet facilement de transformer le montage en une télécommande à grande portée.

Alimentation

La tension d'entrée sur K3 est de 24 V maximum. Il est aussi possible d'alimenter le montage directement en 5 V par K4. Dans les deux cas, l'alimentation doit être capable de fournir jusqu'à 750 mA.

Un dissipateur pour IC1 est optionnel, puisqu'en mode LoRa le rapport cyclique émission/repos est tellement petit (1% maxi) que la dissipation moyenne reste toujours très faible. En réalité, on peut presque aller jusqu'à 35 V à l'entrée. En effet, même avec 750 mA consommé pour un signal de sortie de 30 dB, la puissance à dissiper n'atteint même pas 0,25 W (en supposant que le circuit soit au repos le reste du temps).

Le circuit imprimé

Le circuit imprimé (**figure 2**) mesure 87 mm × 50 mm et peut se loger dans la plupart des boitiers de blindage (optionnel). La sortie antenne est faite avec un connecteur SMA, parfaitement adaptée à cette fréquence proche du gigahertz (ce qui n'est pas le cas des BNC). Le câblage est facile. Les LED, les résistances et les condensateurs de 100 nF et de 1 µF sont en CMS 1206, facile à souder. Les autres composants sont traversants. Si votre platine à des trous non métallisés (**figure 3**), n'oubliez pas les straps, les quelques traversées du plan de masse et ceux pour les liaisons de D4 et D5 entre les deux faces du circuit.

Si vous souhaitez réaliser des liaisons avec de grandes portées (10 km et plus), il est conseillé de placer le circuit imprimé dans un petit boitier étanche, directement au pied d'une antenne placée en hauteur (2 m du sol est un optimum), afin de minimiser les pertes dues au câble HF. Une autre solution consiste à remplacer l'Arduino Nano par un ESP32. Ainsi, les messages pourront transiter en Wi-Fi, avant d'être relayés en LoRa. Les modifications à apporter au programme sont mineures, une version adaptée pourra être fournie.

Un commutateur de mode TX/RX peut être connecté entre les contacts D2 et GND du bornier K1. Il permet de changer, à tout moment, le mode de fonctionnement du module en émission (ouvert) ou en réception (fermé). Si le montage n'est utilisé que dans le mode RX, pontez les contacts 1 et 2 de K1. En mode TX fixe, laissez les contacts ouverts.

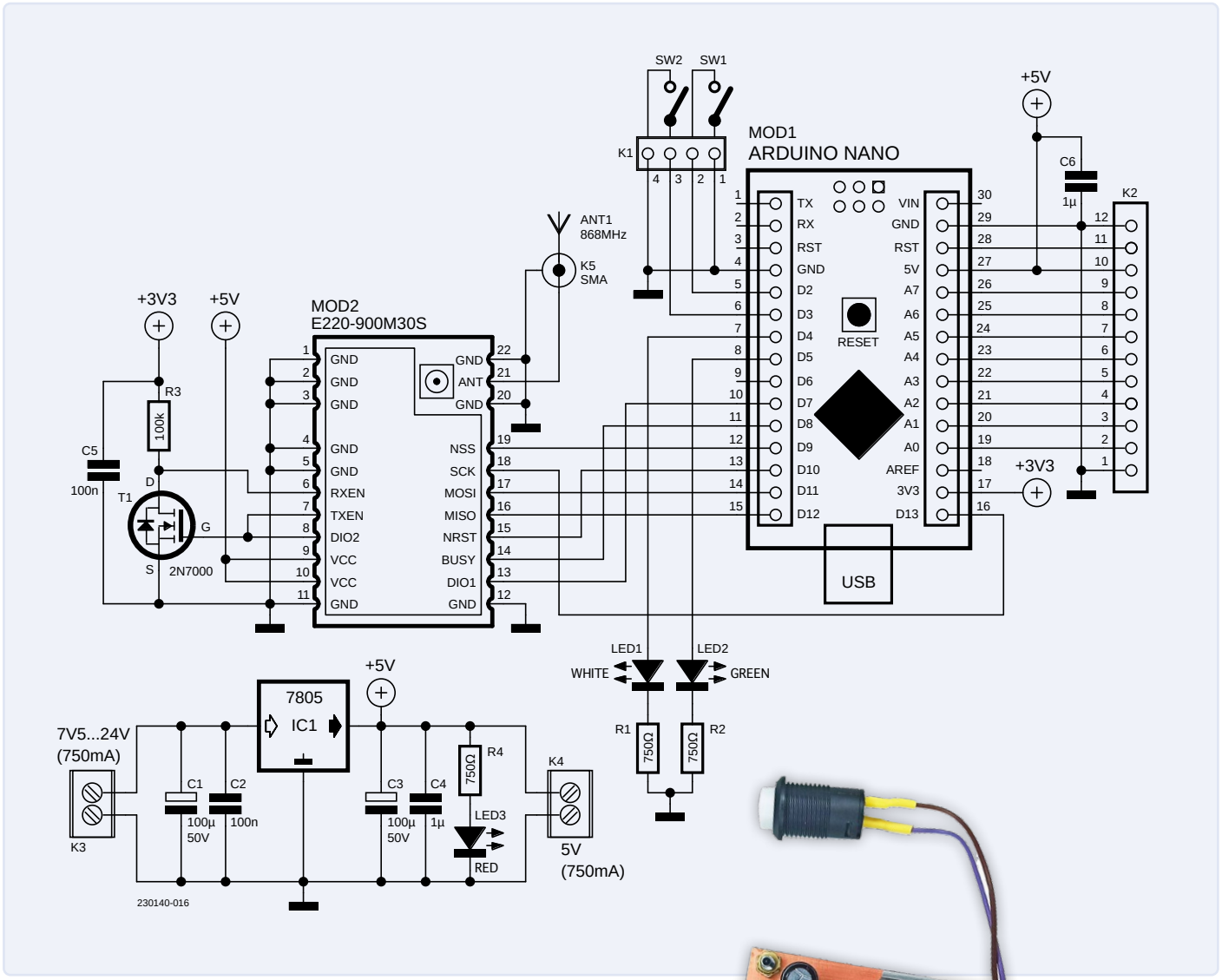


Figure 1. Le schéma du couteau suisse LoRa.

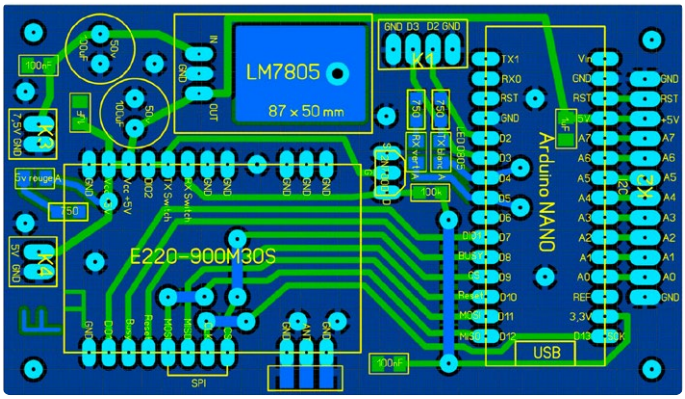


Figure 2. Les fichiers du circuit imprimé sont disponibles avec le code source du programme sur la page internet de cet article [1].

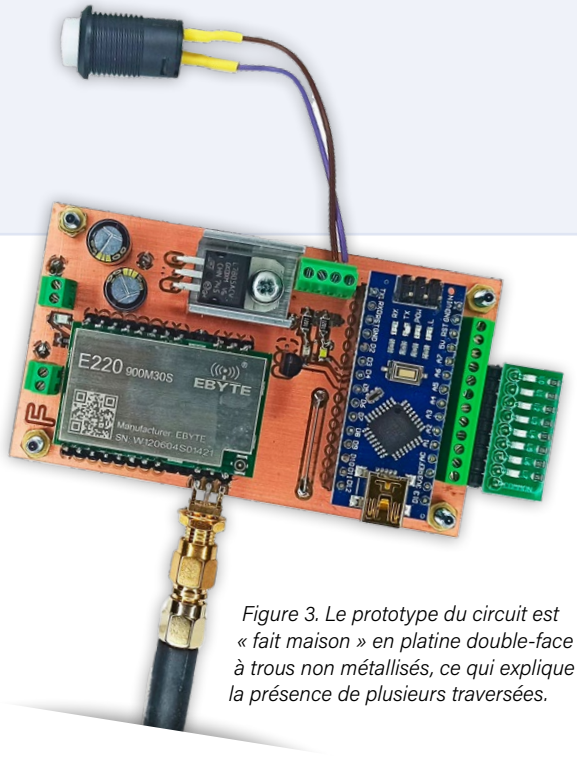


Figure 3. Le prototype du circuit est « fait maison » en platine double-face à trous non métallisés, ce qui explique la présence de plusieurs traversées.

Le programme

Le programme du Nano [1] vous permet de modifier facilement les paramètres LoRa pour toute sorte d'expérimentation et de tentative de grande portée d'une transmission LoRa. Il y a un record à battre… Le logiciel permet également le basculement à chaud entre l'envoi (TX) et la réception (RX) de messages. La connexion de nombreux capteurs analogiques ou numériques (I^2C) est également possible. Voir l'encart « Présentation rapide du programme de gestion du LLCC68 » pour savoir comment le programme est organisé.

Optimisation LoRa

Le LLCC68 permet d'informer l'émetteur de la qualité de réception du signal pour qu'il puisse adapter les paramètres d'émission (puissance, SF, CR, etc.). C'est l'un des moyens d'ajustement utilisé par la couche LoRaWAN.

- Le RSSI (*Received Signal Strength Indication*) est la valeur de la puissance du signal reçu que le récepteur peut renvoyer à l'émetteur. Ce dernier peut ensuite modifier les paramètres d'émission, augmenter ou réduire sa puissance d'émission afin d'augmenter l'autonomie tout en maintenant un niveau suffisant de transmission.
- La sensibilité. Le LLCC68 peut mesurer le niveau de réception minimum lui permettant d'extraire un signal sans erreur. Cette valeur est très faible dans le cas du LLCC68, puisqu'elle peut atteindre –129µdBm en fonction des paramètres choisis.
- Le SNR (*Signal-to-Noise Ratio*) est le rapport entre la valeur de la puissance du signal reçu et le niveau du bruit ambiant que le LLCC68 mesure en dehors des périodes de réception de données. Dans le cas de LoRa, le niveau du bruit peut être très largement supérieur à celui du signal (SNR négatif).
- La fonction *GetStats* fournit le nombre total de paquets reçus, le nombre de ceux reçus avec erreur et le nombre de ceux reçus avec erreur d'entête. La fonction *ResetStats* réinitialise toutes les statistiques.

La lecture des 106 pages de la fiche technique n'est pas indispensable, mais très utile pour approfondir certains sujets traités dans cet article. Nous vous conseillons également de lire le remarquable document rédigé par les enseignants du Lycée Dorian [3]. Il traite de la couche physique LoRa, mais aussi du LoRaWAN et de la démodulation LoRa pour laquelle Semtech ne fournit aucune info.

Les principaux paramètres LoRa

Tous les paramètres LoRa sont rassemblés entre les lignes 83 et 111 :

Ligne 83 : la bascule pour le fonctionnement en LoRa.
Ligne 84 : le paramétrage de FS, BW, CR et LDRO.
Ligne 99 : la configuration du paquet envoyé est paramétrée à l'aide de neuf octets, les trois derniers sont communs avec la modulation FSK.

Pour chacune de ces trois lignes de commande (lignes 83, 84 et 99), le premier octet n'est pas un octet de paramétrage, mais l'*opcode* (la commande) qui indique au LLCC68 ce qu'il doit faire des octets suivants. Il y a 41 opcodes en total. Consultez la fiche technique pour plus de détails. Vous y trouverez également les 36 adresses des registres de configuration.

SF, le facteur d'étalement

La sensibilité de la partie analogique de la réception du signal ne peut être modifiée que par le gain du préamplificateur d'entrée, qui est paramétrable (voir ligne 154 du programme). Mais, comme nous l'avons vu, plus on augmente la valeur de SF, plus le débit binaire diminue. Or l'efficacité du traitement numérique de reconstitution du message étant inversement proportionnelle au débit binaire, la sensibilité numérique augmente donc avec la valeur de SF (ainsi que le temps de transmission). Cela s'ajoute à la sensibilité de la partie analogique, permettant de grandes valeurs de la sensibilité, et donc des portées plus importantes.
SF correspond aussi au nombre de bits transmis pendant la durée T_s, c'est-à-dire par symbole.
Les temps de transmission d'un symbole T_s et d'un message de 24 octets sont présentés dans le **tableau 1**. Semtech fournit sur son site un calculateur (**figure 4**, [2]) qui permet une évaluation précise des durées de transmission, de la consommation et du budget de liaison. N'hésitez pas à l'utiliser, car bien qu'il ne soit pas fait spécifiquement pour le LLCC68 (mais pour les SX1272 à SX1277), les résultats sont quasiment identiques. L'augmentation d'une valeur de SF à la suivante double presque la durée de transmission, comme on peut le constater sur le tableau 1.
La valeur du débit de transmission binaire (*bit rate*) est R_b et est exprimée par la formule :

$$R_b = SF \cdot (BW / 2^{SF}) \cdot 4/(4+CR) \text{ (bits/s)} \qquad (1)$$

Puisque SF est égal au nombre de bits par symbole, on peut en déduire la vitesse en baud R_s (*symbol rate*) :

$$R_s = R_b / SF \text{ (baud)}$$

Comme les trois valeurs BW, SF et CR sont accessibles, on paramètre facilement la vitesse de transmission que nous souhaitons.

CR, le taux de codage

LoRa permet d'augmenter la robustesse du codage en augmentant la redondance du signal. Pour CR = 1, le taux de redondance est de 4/5, ce qui veut dire que cinq bits sont utilisés pour en coder quatre.

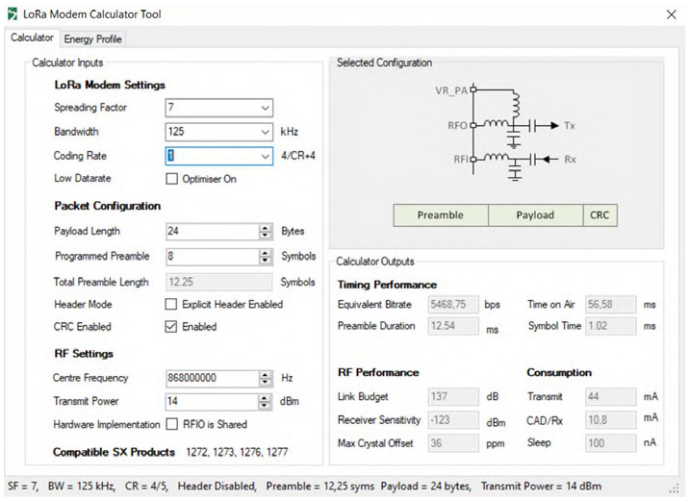


Figure 4. Le calculateur modem LoRa de Semtech existe aussi en version en-ligne.

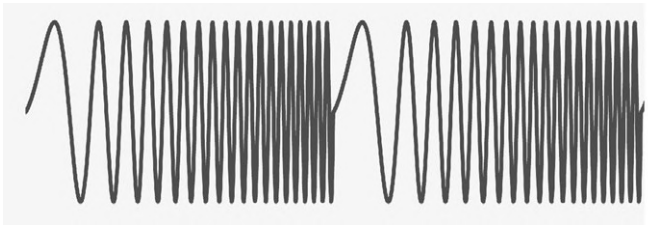


*Figure 5. Représentation de deux **chirps** (avec une modulation augmentée pour l'illustration).*

Tableau 1. Les durées de transmission en fonction du facteur d'étalement SF.

SF	R_b (bit/s)	R_s (baud)	Durée préam-bule (ms)	T_s (ms)	Durée trans-mission (ms)
5	15,625	3,125	3,135	0,256	16,37
6	9,375	1,562,5	6,27	0,51	30,85
7	5,468,75	781,3	12,54	1,02	56,58
8	3,125	390,6	25,09	2,05	102,91
9	1,757,8	195,3	50,18	4,10	185,34
10	976,5	97,7	100,35	8,19	370,69
11	537,11	48,8	200,70	16,38	659,46

Ces valeurs sont sans entête. La présence d'un entête augmentera les durées. Ici, CR = 1, BW = 125 kHz, un message (payload) contient 24 octets et un préambule consiste en 8+4,25 = 12,25 symboles. $T_s = 2^{SF}/BW$.
Le facteur d'étalement SF correspond également au nombre de bits transmis pendant la durée T_s. *Les valeurs* R_b *et* R_s *du tableau sont celles obtenues avec le calculateur Semtech données pour être compatibles avec les produits SX1272 à 1277. Bien que le fondeur ne le précise pas, elles sont également compatibles avec le LLCC68. Remarque : les délais de transmission* R_b *d'un bit du payload (message utile) et d'un bit du préambule ne sont pas identiques.*

Pour CR = 4, le taux est de 4/8, soit huit bits pour en coder quatre. Puisque tout avantage à une contrepartie, augmenter CR fait diminuer R_b (voir équation 1) et, par conséquent, augmente la durée de transmission d'un message.

BW, la bande passante

La largeur de bande BW est aussi appelée encombrement spectral du signal modulé. La variation d'un *chirp* (**figure 5**) qui s'étend de F – df à F + df détermine la bande passante occupée. Puisque, selon l'équation 1, R_b est proportionnel à BW, on peut réduire la durée de transmission d'un message en choisissant une valeur plus grande pour BW. Contrairement aux autres paramètres, il n'y a pratiquement aucune contrepartie à cette augmentation, hormis l'encombrement de la bande de fréquence.

L'optimisation des données pour les faibles débits

Compte tenu de la durée potentiellement longue du paquet à des valeurs de SF élevées (10 ou 11), l'option d'optimisation des données pour les faibles débits (LDRO, *Low Data Rate Optimize*) doit être activée. Elle améliore la fiabilité de la transmission vis-à-vis des variations de fréquence pendant la durée de la transmission et de la réception du paquet. Son utilisation est obligatoire lorsque la durée du symbole T_s dépasse 16 ms. Par exemple, on peut voir au tableau 1 que si SF = 11, LDRO doit être activé, car T_s > 16 ms. L'émetteur et le récepteur doivent utilisent la même valeur pour LDRO.

Présentation rapide du programme

Le programme de 48 Ko comporte 571 lignes, dont une grande partie de commentaires très détaillés (en français). Ils renvoient aux numéros de pages de la fiche technique du LLCC68. Le programme est structuré simplement afin de pouvoir être adapté à tout type d'application.

Ligne 1 : le programme inclut la bibliothèque *SPI.h*. Elle fait partie de la dotation initiale de l'EDI Arduino. Aucune installation de bibliothèque n'est à prévoir.
Lignes 5 à 15 : deux tableaux, l'un pour stocker le message à émettre (TX), le second pour stocker le message reçu (RX).
Lignes 17 à 28 : regroupement de l'initialisation des bornes d'entrées et de sorties pour faciliter l'adaptation à tout type d'application.
Lignes 31 à 48 : déclaration des variables nécessaires pour les calculs internes du programme et les adresses des registres du LLCC68 utilisées par le programme.
Lignes 50 à 56 : déclaration des variables du paramétrage général du LLCC68.
Lignes 57 à 81 : déclaration des variables dédiées à la modulation FSK (les trois derniers paramètres du paquet sont communs avec LoRa).
Lignes 83 à 111 : déclaration des variables pour la modulation LoRa.
Lignes 113 à 170 : déclaration des variables pour le paramétrage de toutes les fonctions RF du LLCC68
Ligne 172 : la fonction Arduino obligatoire *setup*
Ligne 197 : l'autre fonction Arduino obligatoire *loop*. Elle se divise en deux parties. Le commutateur de mode RX/TX détermine quelle partie est exécutée.
Lignes 232 à 277 : cinq fonctions de correction de certaines limites du LLCC68 (voir chapitre 15 de la fiche technique)
Lignes 278 à 311 : les fonctions *Message, busy* et *RX_Wait*, liés au mode RX. Dans la fonction *RX_Wait*, une boucle infinie attend la réception d'un message LoRa valide. Le commutateur de mode permet de sortir de la boucle et de passer du mode RX au mode TX.
Lignes 313 à 355 : *TXRX_Setup*, le paramétrage commun aux deux modes RX et TX, puis *TX_Setup* pour les paramétrages du mode TX et *RX_Setup* pour les paramétrages du mode RX
Ligne 356 : la fonction *TX_Send* qui permet le lancement du mode TX et démarre l'émission d'un message.
Ligne 371 : la fonction *RX_Read* qui lance le mode RX et place le programme dans une boucle infinie en attendant un message LoRa valide.
Ligne 411 : la fonction *Fsk_Setup* qui paramètre le mode FSK.
Ligne 433 : la fonction *Lora_Setup* qui paramètre le mode LoRa.
Lignes 454 à 483 : regroupent les fonctions de *commande SPI* puis *Status* qui permet de connaitre l'état du LLCC68.
Lignes 484 à 571 : regroupent les fonctions de mesure *RSSI*, *SNR* et les *statistiques* des erreurs de transmission et du nombre de messages reçus. Ils peuvent être réémis pour optimiser la liaison.

Vitesse de transmission

La vitesse de transmission d'un message en LoRa et la portée de la transmission est la résultante des valeurs de SF, CR et BW et LDRO, mais également du choix d'avoir un entête (option ***explicite***) ou pas (option ***implicite***) et de la longueur de l'entête, lorsqu'il est présent.

Retour sur les modules E220-900T

Les logiciels destinés au pilotage des modules E220-900T 22 ou 30 présentés dans la première partie [5] ne permettent pas le paramétrage LoRa comme vous pouvez le faire maintenant. Ils ajustent automatiquement les paramètres qui vont bien pour obtenir la vitesse de transmission désirée. C'est fort dommage, car chaque arrangement a ses avantages et ses inconvénients. Il est donc intéressant de pouvoir choisir. C'est la raison qui nous a fait sélectionner le module E220-900M30S qui vous donne, via sa liaison SPI, toute latitude dans son paramétrage.

Utilisation du logiciel

Toutes les commandes et les paramétrages, hormis celles qui sont spécifiques à votre capteur, sont rassemblés en début de programme et sont abondamment commentés. D'autre part, pendant l'exécution du programme, vous pouvez suivre pas à pas son déroulement, sur le moniteur série de votre PC, car le programme comporte un grand nombre de `serial.print`. Voir aussi l'encart **Présentation rapide du programme** pour plus de détails.

Envoyer un message (TX)

En tout début de programme, ligne 6, il y a un tableau `write_buffer` qui, au moment du lancement du programme, doit contenir votre message. Pour cela, remplacez les codes ASCII des lignes 7 et 8 par votre message (127 octets maxi en ASCII). Si votre message est une valeur numérique (p. ex. issue de la mesure d'un capteur), il vous faudra la convertir en ASCII avant de l'inscrire dans le tableau.
Respectez le format du tableau. Ne changez pas les deux premiers octets : 0x0E est la commande (*opcode*) nécessaire au LLCC68 pour écrire dans le tampon ; 0x80 est l'adresse du début de la partie TX du tampon. Toutefois, il faut modifier la valeur du troisième octet, qui est le nombre total d'octets de votre message. Elle sera utile lors de la récupération du message au moment de la réception. Attention, toutes les valeurs sont en hexadécimale.
Suivant la longueur de votre message, vous pourrez être amené à modifier la valeur de plusieurs paramètres : longueur maximale du message (ligne 99, octet 4 et 7), mais également les lignes 125 et 126 pour la réception. Ligne 122, vous pourrez changer la répartition du Buffer entre réception et émission et les paramètres spécifiques à LoRa (lignes 83 à 112).
Mettez l'interrupteur SW_1 en position TX (ouvert), puis relancez le programme. Votre message est ensuite envoyé périodiquement, espacé d'un délai préprogrammé (ligne 229). Il faudra, vraisemblablement, réajuster le rapport temps d'occupation/temps de repos pour qu'il ne dépasse pas 1 % (contrainte de la règlementation française pour l'émission LoRa).

Recevoir un message (RX)

Positionnez l'interrupteur SW_1 en position RX (fermé), puis relanciez le programme. Il va alors attendre la réception d'un message LoRa valide qui sera automatiquement placé dans le tableau `recevoir_buffer` (ligne 14). Il vous suffira de le lire.
Une autre manière de faire consiste à modifier la fonction `RX_Read`, ligne 371, qui permet de récupérer un message reçu par la boucle, ligne 381 à 396. Il contient une fonction de reconnaissance (optionnelle) de la lettre « A » qui déclenche la mise à 1 du drapeau `Message_Valide`. C'est un exemple qui allume une LED de la carte, mais qui peut faire une télécommande.

Quelques idées d'application

La balise GPS pour animal, personne, drone ou autre objet est un exemple d'application de notre couteau suisse LoRa. Mais, le programme que nous avons réalisé est très versatile et il permet un grand nombre d'autres applications, tel que :

- La centralisation de données (humidité du sol, ensoleillement, etc.) d'une exploitation agricole ou d'une station météo.
- Une centrale domotique étendue à un jardin, un parc, plusieurs maisons, une rue entière ou même un village.
- La surveillance (incendie, sécurité) ou la télémétrie d'une zone étendue ou isolée (piscine, terrain de sport, unité industrielle, etc.).
- Une centrale de distribution de l'heure, des phases de la lune et celles des principales planètes vers des horloges réceptrices réparties dans une maison ou un monument plus important (musée, mairie, etc.) ou dans tout le quartier d'une ville.

Il y en a bien plus encore, mais il est probable que d'autres idées germent déjà dans votre imagination…

230140-B-04

La norme européenne

Vous pouvez émettre avec le protocole LoRa sur la bande 868 MHz en respectant la réglementation européenne [4]. En voici le résumé :

1. Ne pas émettre en dehors de la bande des 868 MHz.
2. Ne pas utiliser la bande passante BW = 500 kHz.
3. Ne pas dépasser 14 dBm (25 mW) de puissance rayonnée dans l'air. Cela peut correspondre à beaucoup plus en sortie du module suivant les pertes entre le module et l'antenne.
4. Ne pas émettre pendant plus de 1% du temps (rapport cyclique). Cela veut dire que si le temps de transmission d'un message est de 0,1 seconde, nous ne pouvons effectuer une nouvelle émission que toutes les 0,1 × 100 = 10 secondes, soit six fois par minute (ce qui suffit à de nombreuses applications).

Prenons un exemple. Si, comme l'auteur, votre compagnon est un chien Beagle, très fugueur, dont vous souhaitez connaitre les coordonnées GPS afin de pouvoir le retrouver lors de ses fugues, un collier contenant un récepteur GPS, un émetteur LoRa et une petite batterie suffit. Il vous renvoie six fois par minute la position du chien dans un rayon de 10 km en campagne. L'ensemble consomme peu d'énergie et on a un système léger, petit et avec une grande autonomie.

À propos de l'auteur

Gilles Brocard est ingénieur-conseil à la retraite et formateur LTspice. Il est également spécialiste de la conception de convertisseurs (SMPS).

Questions ou commentaires ?

Envoyez un courriel à l'auteur (brocard.gilles.b26@gmail.com), ou contactez Elektor (redaction@elektor.fr).

Produits

- **Radio Logicielle HackRF One Great Scott Gadgets (1 MHz à 6 GHz) (SKU 18306)**
  www.elektor.fr/18306
- **CircuitMess Chatter – DIY LoRa Communicator (SKU 20407)**
  www.elektor.fr/20407
- **Claus Kühnel,** *Develop and Operate Your LoRaWAN IoT Nodes*, **Elektor 2023 (SKU 20147)**
  www.elektor.fr/20147

LIENS

[1] Code source et circuit imprimé : https://www.elektormagazine.fr/230140-B-04
[2] LoRa modem calculator tool : https://lora-developers.semtech.com/build/tools/calculator
[3] Lycée Dorian: « Caractérisation de l'interface radio LoRa d'un réseau de communication LoRaWAN » , eduscol.education.fr : https://www.elektormagazine.fr/lycee-dorian
[4] Réglementation officielle radio pour la France: https://www.arcep.fr/uploads/tx_gsavis/21-1589.pdf
[5] Première partie : https://www.elektormagazine.fr/230140-04

conception et construction de **microphones MEMS**

Peter Riccardi (États-Unis))

Vous pouvez dépenser des milliers d'euros pour un microphone à champ libre. Mais avant de le faire, découvrez ce circuit basé sur les MEMS, qui vous offrira de superbes performances à un coût bien moindre.

Les acousticiens et les ingénieurs en acoustique ont souvent recours à des microphones de mesure pour acquérir des signaux physiques au cours de leurs expériences. Les microphones à condensateur typiques de sociétés telles que PCB Piezotronics coûtent plus de 1000 USD. La qualité de ces microphones est superbe, mais il existe des alternatives disponibles pour les amateurs et les passionnés qui vous permettront d'obtenir les performances dont vous avez besoin pour une fraction du coût. Ici, on étudie une conception basée sur les MEMS avec l'objectif d'avoir un plancher de bruit aussi bas que possible et une réponse stable en fréquence dans la bande audio de 20 Hz à 20 kHz.
Le prototype doit être peu coûteux, simple à concevoir et à assembler, et s'intégrer de manière simple dans le flux de travail typique d'un acousticien expérimentateur. Les projets typiques en acoustique sont alimentées par des entrées IEPE DAQ, de sorte que le microphone fonctionnera avec une excitation IEPE [1], ce qui permet d'alimenter le préamplificateur et les éléments MEMS. La technologie des microphones MEMS s'est améliorée au cours des dernières années, de sorte que leur bruit de fond est assez faible. Le moyennage analogique par l'utilisation d'un réseau en parallèle permet de réduire davantage le bruit ; nous utiliserons cette méthode ici.

Schéma électrique

Il existe de nombreux types de microphones MEMS qui pourraient répondre aux exigences de ce projet. Ici, nous avons choisi le TDK ICS-40300 pour sa réponse étendue en basse fréquence (grâce à un volume arrière étendu) : 6-20 000 Hz, sa sensibilité raisonnable : -45 dBV à 94 dB SPL, et son faible bruit de fond : -108 dBV. Sur le plan mécanique, il sera logé dans un boîtier d'un diamètre nominal de 0,5 pouce. Cela lui donnera un petit facteur de forme et permettra une utilisation facile dans un laboratoire acoustique avec les calibrateurs existants, les supports de micro, etc. Un connecteur BNC situé à l'arrière du boîtier alimentera l'IEPE et émettra la forme d'onde modulée en tension. Un réseau sera conçu pour réduire le bruit de fond. L'objectif est de disposer de quatre éléments dans le réseau, pour une amélioration maximale du bruit de 6 dB. Pour obtenir la gamme dynamique la plus large possible, le préamplificateur fournira un gain pour amener la sortie maximale prévue du microphone au rail d'alimentation. Ici, ce sera 3,3 V. Selon la fiche technique, la tension maximale de sortie est de 0,355 V_{rms} @ 130 dB SPL. Quelques calculs rapides permettent d'obtenir des valeurs plus intuitives.
Tout d'abord, la sensibilité doit être exprimée en mV/Pa. Le niveau de pression acoustique (SPL) est :

$$dB\ SPL = 20 \log_{10} (P_{rms} / P_{ref})$$

SPL est une valeur en décibels. P_{rms} est la moyenne quadratique de la pression mesurée. P_{ref} est la pression de référence (20 µPa dans l'air). Avec un simple calcul, on obtient 1,417 Pa_{pk} à la sensibilité nominale de -45 dBV. Maintenant, la tension (en mV) doit être dérivée de façon similaire à partir de la valeur donnée de -45 dBV. Ici, la tension de "référence" est supposée être de 1 V. L'équation de base serait la suivante :

$$dBV = 20 \log_{10} (V / 1\ V)$$

produisant une tension de crête de 5,623 mV_{pk}. Remarque : par définition, le niveau de pression acoustique correspond à une forme d'onde de pression moyenne quadratique. La tension étant une simple valeur en décibels, le numérateur et le dénominateur de la fonction logarithmique sont supposés être des valeurs de crête. En divisant la tension de crête par la pression de crête, on obtient la sensibilité suivante :

$$sensitivity = 1.417\ Pa_{pk} / 5.623\ mV_{pk} = 3.968\ mV / Pa$$

La sensibilité sera multipliée par tout gain dans le préamplificateur, de sorte que la sensibilité peut être réécrite comme suit :

$$sensitivity = 3.968\ {}^{mV}/_{Pa} * A$$

où :

A = gain

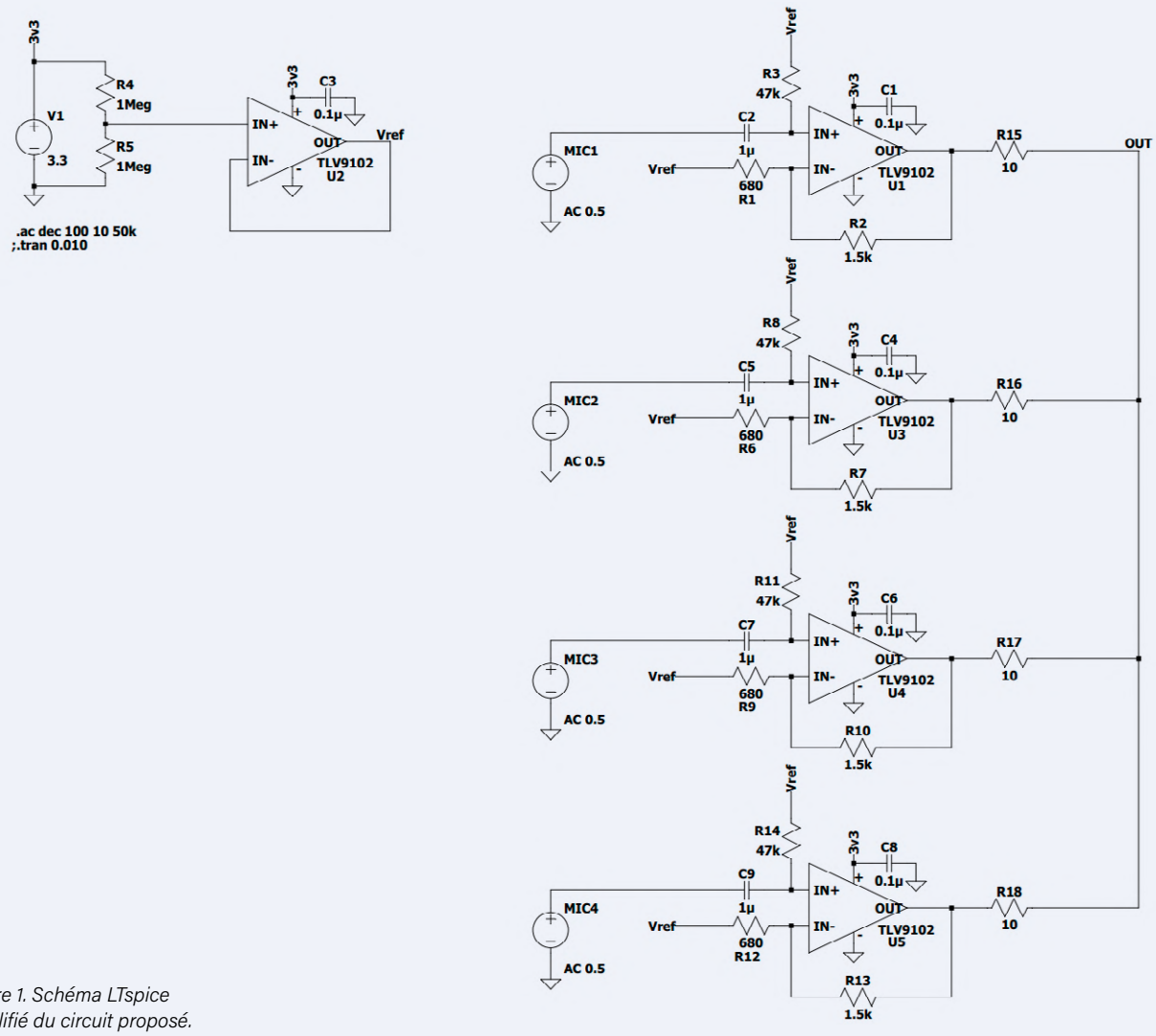


Figure 1. Schéma LTspice simplifié du circuit proposé.

On est loin de la qualité de mesure la plus typique de 50 mV/Pa, mais c'est un compromis nécessaire pour utiliser les éléments MEMS à basse tension avec un large intervalle dynamique d'amplitudes de pression – jusqu'à 130 dB SPL. Un microphone à condensateur ordinaire peut osciller jusqu'à 20 V_{pk-pk} pour les modèles à faible tension. Il est possible d'augmenter cette sensibilité en ajoutant du gain au préamplificateur. Pour maximiser la gamme dynamique de notre système, nous devons tenir compte de la marge de manœuvre. Nous avons choisi des amplificateurs à basse tension, fonctionnant sur une rampe de 3,3 V. Cela garantit une faible consommation d'énergie, tout en maximisant le niveau de variation de tension que le système peut produire. La fiche technique précise la tension maximale de sortie du microphone à la pression nominale la plus élevée qu'il peut atteindre ; cette valeur peut être utilisée pour calculer le gain. Selon la fiche technique, la tension de sortie maximale est de :

V_{out} = 0.355 V_{rms}

La tension de sortie crête à crête est d'environ :

V_{out} = 0.355 * 1.414 = 1 V_{pk-pk}

L'oscillation totale d'un amplificateur opérationnel rail à rail fonctionnant sur un rail unipolaire de 3,3 V est d'environ 3,3 V. Le gain, pour atteindre l'oscillation maximale, serait donc :

$$A = \frac{output}{input} = \frac{3.3\ V_{pk-pk}}{1.00\ V_{pk-pk}} = 3.3$$

Cela implique également l'utilisation d'une masse virtuelle à 3,3 V/2 = 1,65 V et qui constituera donc la tension de référence, V_{ref}, utilisée dans ce circuit. Les éléments du microphone peuvent avoir un décalage de tension continue considérable, de sorte que les sorties de chaque microphone doivent être couplées en courant alternatif par un simple filtre RC. Ici, un condensateur de 1 µF et une résistance de 47 kΩ sont utilisés, produisant une fréquence de coupure de :

$$f_c = \frac{1}{2\pi RC} = \frac{1}{6.28 * 47{,}000 * 0.000001} = 3.4\ Hz$$

On a choisi une fréquence de coupure relativement élevée pour pouvoir utiliser un petit condensateur 0603 avec une tension nominale appropriée.

Circuit

Le circuit de la figure 1 est un schéma simplifié illustrant la topologie proposée. Le nœud de sortie constitué de résistances de 10 Ω forme un circuit de moyennage analogique. Théoriquement, le bruit généré par l'ensemble des ampli-op et le bruit généré (électriquement) dans les

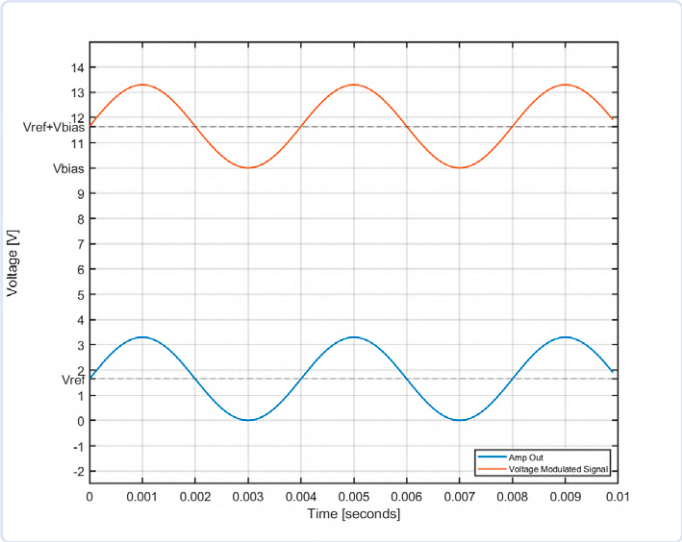


Figure 2. Formes d'ondes typiques et polarisation dans un système alimenté par IEPE. Ici, le signal injecté est une onde sinusoïdale de 250 Hz est. Le tracé bleu correspond à la sortie obtenue au niveau du nœud de calcul de la moyenne. La sortie est polarisée sur la tension de référence, V_{ref}, et on peut voir le signal osciller autour de cette tension continue. En décalant cette tension par une autre tension, V_{bias}, on peut programmer la sortie pour qu'elle fonctionne à une tension continue égale à environ la moitié de la tension de conformité prévue d'un DAQ IEPE. On a utilisé le 4431 de NI comme référence, qui produit 2,1 mA à une tension devant dépasser 20 VDC. Par conséquent, on a choisi une valeur autour de 10-15 VDC comme point de fonctionnement du système.

microphones MEMS ne sont pas corrélés. Il est donc possible d'obtenir une amélioration maximale du bruit de fond de 6 dB. Ce problème est compliqué parce que le bruit qui pourrait se produire à la sortie n'est pas uniquement dû à l'électronique. Les microphones MEMS sont sujets à un bruit acoustique dû aux interactions moléculaires de l'air en contact avec ses diaphragmes. Le phénomène du bruit dans le contexte des circuits électriques et des systèmes acoustiques est extrêmement complexe et dépasse largement le cadre de cet article. On suppose que, sur le plan électrique, les sources de bruit ne sont pas corrélées, et l'on espère que la majorité des bruits acoustiques ne le sont pas non plus. Au moins, le bruit acoustique qui n'est pas corrélé sera moyenné. Les lecteurs intéressés trouveront des informations complémentaires sur le bruit ici [2][3].

Remarque concernant la figure 1 : une tension de 3,3 VDC est généré du rail, avec un diviseur de tension à haute impédance tamponné pour produire une tension de référence égale à la moitié de la tension d'alimentation. Quatre éléments MEMS sont en parallèle, chacun avec son propre préamplificateur avec un gain nominal de A = 3,2. Le filtre RC élimine le décalage en courant continu des éléments MEMS tout en injectant simultanément la tension de référence dans la borne non inverseuse. Les sorties de chaque ampli-op sont reliées entre elles par des résistances à faible impédance. La séparation ohmique permet aux tensions d'être effectivement moyennées à leur nœud de sortie sans créer de court-circuit entre les bornes et saturer les transistors de sortie.

Alimentation IEPE et distribution de puissance

Integrated Electronics Piezo-Electric (IEPE) est une norme intelligente de distribution de puissance utilisée dans de nombreux systèmes d'acquisition de données (DAQ). Elle résout ingénieusement le problème de l'alimentation et de la réception d'un signal sur une interface à deux fils. Pour ce faire, elle exploite les propriétés variables d'une source de courant constant. Rappelons qu'une source de courant constant a - dans le cas idéal - une impédance de sortie

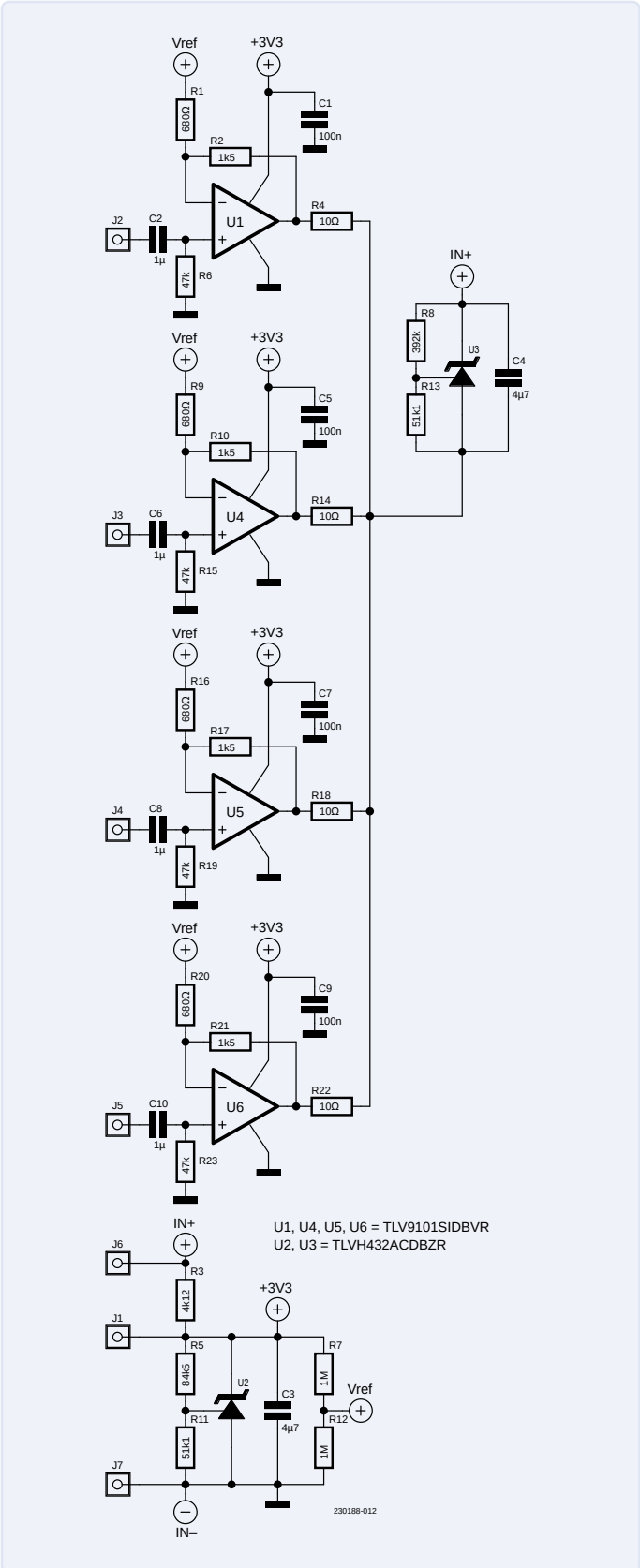


Figure 3. Le schéma du circuit conçu, construit et testé. On a initialement omis le tampon sur la tension de référence (voir figure 1) dans le premier prototype. Il s'agit d'une erreur de conception qui pose des problèmes. Il ne faut donc pas l'omettre si l'on reproduit le travail. Les références "J" sont des pastilles à trous traversants pour souder des "fils volants". Le connecteur BNC dans le boîtier mécanique est câblé point à point au préamplificateur. Le câble positif du connecteur alimente U2, le régulateur de 3,3 V, et est traversé par une tension égale à VIEPE - la tension de conformité définie. La sortie du nœud de sommation alimente un autre régulateur Zener réglé à environ 10 VDC, produisant ainsi la tension de conformité totale comme illustré à la figure 2.

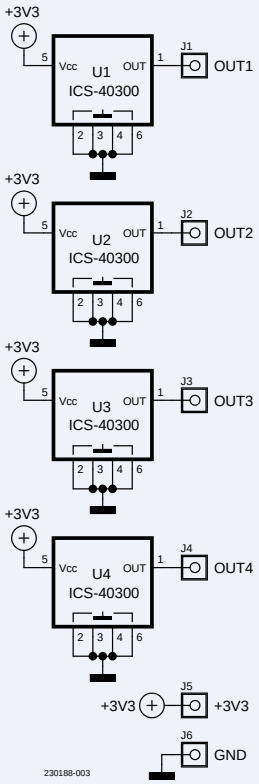


Figure 4. Schéma de la carte MEMS "porteuse" - un petit circuit imprimé circulaire qui abrite les quatre microphones et comporte un certain nombre de pastilles traversantes pour le câblage des câbles de signaux de sortie (entrées du préamplificateur) et pour l'alimentation et la mise à la masse. Comme on peut le voir, il y a au total six références "J", quatre signaux de sortie, plus 3,3 V et GND.

Figure 5. Un microphone construit, emballé dans un boîtier imprimé en 3D avec le couvercle enlevé. L'emballage final est surdimensionné pour une conception nominale de 0,5 pouce, mais il est assez approprié pour une démonstration rapide du concept.

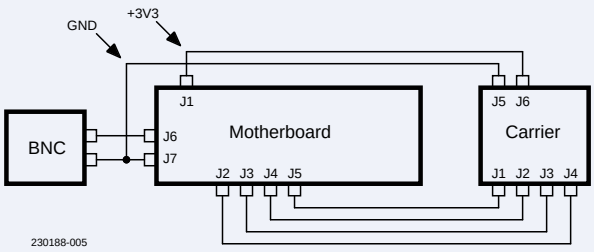


Figure 6. Le guide du câblage des circuits est présenté ici. Les tracés bleus correspondent aux signaux, les tracés noirs à la masse et les tracés rouges au rail d'alimentation. La "carte mère" est le préamplificateur, et le "support" contient les quatre microphones.

infinie. En pratique, cela se traduit par une pente très élevée de la courbe V-I. Cela signifie que la source de courant introduit un courant dans la charge (le préamplificateur et le microphone) quelle que soit la tension qu'elle reçoit. Ainsi, le préamplificateur peut moduler la tension - appelée tension de conformité - et un courant constant circulera toujours dans le préamplificateur pour ses rails d'alimentation et sa distribution. La **figure 2** montre un exemple de forme d'onde observée dans un système basé sur l'IEPE.

Le 4431 de NI a été utilisé comme DAQ de "référence" approprié parce que les composants de National Instruments sont largement utilisés dans la recherche acoustique et que leurs sources d'IEPE sont toutes similaires. Le 4431 délivre 2,1 mA et a une tension de conformité qui dépasse 20 VDC. Si le point de fonctionnement en courant continu du système se situe dans ces limites, le circuit fonctionnera. Il convient de prévoir une marge de manœuvre suffisante pour que la tension alternative maximale polarisée par la tension de conformité continue ne dépasse pas les conditions de fonctionnement maximales.

Nous pouvons maintenant utiliser le courant constant de 2,1 mA pour concevoir notre schéma de distribution de puissance. Nous avons utilisé des régulateurs shunt Zener pour leur faible courant de cathode requis et leur facilité d'utilisation. Les ampli-op et les microphones MEMS consomment le plus d'énergie. D'après leurs fiches techniques, ils ont respectivement les caractéristiques suivantes :

I_q = 150 µA et I_q = 250 µA

En supposant qu'on a cinq ampli-op et quatre MEMS, le tirage de courant pour leur alimentation est égal à 1,75 mA. Les 350 µA restants alimenteront les régulateurs shunt. Le régulateur Zener alimentant le rail 3,3 V aura une résistance réglant son courant de cathode. On a choisi le TLVH432 comme référence Zener car il est ajustable. Il peut fonctionner avec 100 µA de courant de cathode. Le régulateur de tension de polarisation n'aura pas de résistance en série pour régler son courant de cathode. Il " absorbera " simplement la différence entre le courant consommé par le circuit et les 2,1 mA du courant fourni.

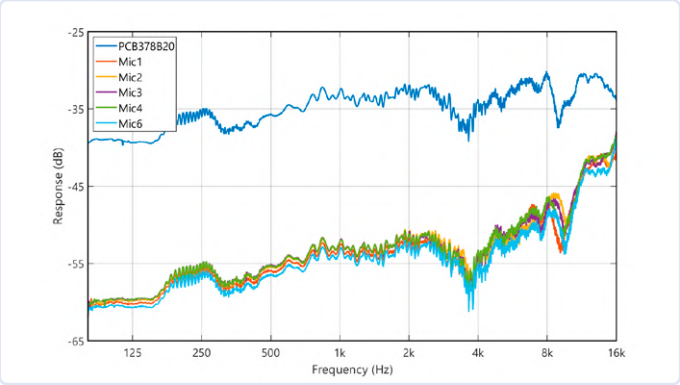


Figure 7. Réponse en fréquence des microphones (cinq d'entre eux) comparée à celle d'un microphone à champ libre PCB378B20. Ces mesures ont été effectuées dans une chambre semi-anéchoïque au-dessus de 200 Hz. Les tendances entre le microphone MEMS et le microphone commercial sont similaires. La plupart de ces "ondulations" ne sont pas de nature électrique, mais plutôt dues aux interactions acoustiques complexes des réflexions et des rebonds indésirables sur le support de la perche et d'autres surfaces dures dans la chambre. La différence notable est la tendance à la hausse des éléments MEMS au-dessus de 10 kHz, qui suit la FRF indiquée dans la fiche technique.

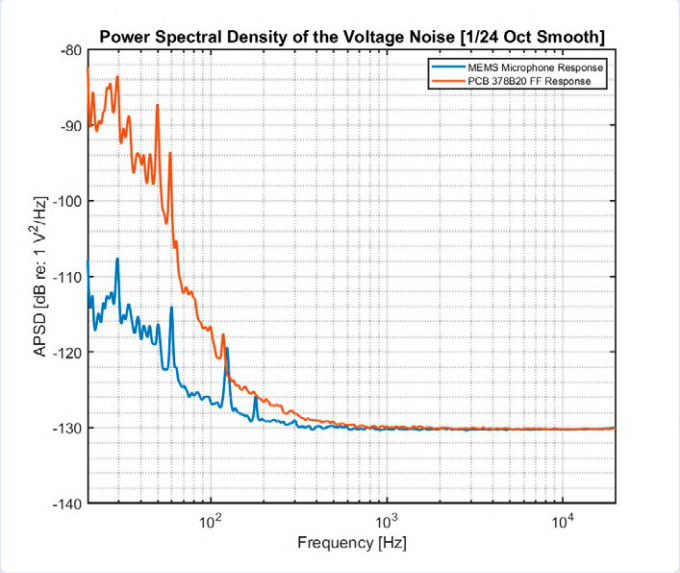


Figure 8. Bruit de tension mesuré dans une chambre semi-anéchoïque sans aucun son diffusé dans la chambre et sans correction de sensibilité (formes d'ondes de tension brutes). Au-dessus de 1 kHz, le microphone MEMS et le microphone commercial s'approchent tous deux du seuil de bruit du DAQ.

Vous trouverez un exemple complet de tous les calculs dans la note d'application complète proposée par Analog Devices [4]. En raison des pénuries d'approvisionnement et des problèmes possibles d'approvisionnement en références Zener à forte demande, une substitution appropriée serait de tirer parti d'une référence de tension flottante utilisant deux résistances et un NPN BJT 2N2222A (indice : consultez "741 op-amp discrete circuitry").

Prototype de préamplificateur et de microphone

Les **figures 3** à **6** illustrent le circuit final. Nous avons conçu deux circuits imprimés, un petit circuit circulaire pour loger les quatre éléments du microphone, et un petit circuit rectangulaire pour loger l'électronique du préamplificateur. Le circuit a été câblé point à point et assemblé dans un boîtier imprimé en 3D avec des dimensions proches de celles d'un microphone de mesure nominalement 0,5 pouce. Pour les résultats des mesures, voir les **figures 7** et **8**.

Conclusions et corrections

La première chose à noter est l'erreur évidente dans le schéma de la figure 3. Le circuit présenté a été conçu rapidement afin de répondre aux exigences de financement du projet initial. Il n'y a pas eu de temps pour la simulation. Ainsi, l'erreur évidente du rail surchargé n'a pas été détectée. Bien que le courant de sortie de ce rail soit très faible, de sorte que le circuit passe la "vérification" normale pour voir si le diviseur de tension est "rigide". Les amplificateurs opérationnels pilotent activement leurs sorties via la rétroaction à faible impédance pour égaliser les tensions des bornes inverseuses et non inverseuses. Il n'y a que 680 Ω de résistance entre la borne non inverseuse et la sortie du réseau diviseur. Le réseau diviseur a une séparation de 1 MΩ entre le point de mesure et le rail régulé. Donc, effectivement, l'ampli-op via sa rétroaction à faible impédance conduit activement la valeur sur ce nœud à être égale à sa propre sortie. L'effet est que le nœud de référence suit la sortie et se réduit essentiellement à un suiveur à gain unitaire, mais avec des interférences supplémentaires possibles. On a effectué toutes ces mesures sur le circuit original, comme le montre la figure 3. Il est donc possible d'améliorer les mesures de bruit. Un lecteur attentif aux calculs pourrait remarquer la faible sensibilité de la fonction de réponse en fréquence mesurée - c'est parce que l'amplificateur fonctionnait en mode gain unitaire, donc il était en baisse d'environ 10 dB. Lorsque cette carte sera remise en service, le tampon sera absolument nécessaire pour un fonctionnement correct.

Assemblage

Les travaux futurs impliquent de reconcevoir l'enceinte pour qu'elle soit plus robuste. L'assemblage a été assez difficile car il y avait plusieurs petites caractéristiques approchant les limites d'épaisseur de paroi de l'imprimante avec une buse de 0,4 mm. La meilleure solution consistera probablement à utiliser un microphone d'un diamètre nominal de 1 pouce. Les circuits imprimés ont toujours été conçus pour être fabriqués avec une technique flexible rigide pour faciliter l'assemblage. Cela n'a pas été possible pour le premier prototype à cause des contraintes de temps. La combinaison de la carte porteuse et du préamplificateur sur un seul modèle, sans la solution de câblage point à point, simplifiera grandement la complexité et réduira le temps d'assemblage. Le coût total de construction d'une de ces unités était d'environ 30 USD ; la figure de bruit mesurée et la réponse en fréquence sont assez bonnes par rapport au coût.

Tous les fichiers sont disponibles sur la page du projet sur le site Hackaday [5].

230188-04

Note de l'auteur

Je tiens à remercier l'équipe SPRAL de l'université d'État de Pennsylvanie pour son aide dans l'acquisition des mesures en chambre anéchoïque, et tout particulièrement M. Zane Rusk pour son soutien infatigable. De même, j'aimerais remercier l'Acoustical Society of America Chapter de l'université d'État de Pennsylvanie pour son soutien apporté tout au long du projet et lors de la présentation des travaux. PCBway a sponsorisé la construction de cinq microphones MEMS et a fourni tous les composants gratuitement. Je tiens à les remercier pour leur soutien au projet, qui a permis à des étudiants de troisième cycle de concevoir, construire et tester ces microphones de mesure gratuitement.

À propos de l'auteur

Peter Riccardi travaille actuellement dans le domaine de l'aérospatiale en tant qu'ingénieur en avionique (électronique). Il a suivi une formation en génie mécanique et a obtenu un diplôme en 2018 à l'université de Rowan. En 2022, Peter a obtenu une M.S. en acoustique à l'Université d'État de Pennsylvanie. Il s'intéresse principalement à l'électronique audio et à l'électroacoustique ; sa recherche consistait à proposer un modèle de transduction linéaire du circuit équivalent du transformateur de mouvement de l'air, un appareil électroacoustique innovant. Ses projets personnels comprennent des réseaux de croisement actifs montés sur mesure en rack avec des amplificateurs de puissance intégrés, des préamplificateurs de platines à faible bruit, des platines sur mesure (cardan mécanique, moteurs, emballage, etc.).

Questions ou commentaires ?

Envoyez un courriel à l'auteur (pjriccardi@pjroses.co) ou contactez Elektor (redaction@elektor.fr).

Produit

> **Elektor Audio Collection (clé USB)**
> https://elektor.fr/19892

LIENS

[1] IEPE [Wikipedia]: https://en.wikipedia.org/wiki/Integrated_Electronics_Piezo-Electric
[2] D. Self, "Chapter 1: Basics," Small Signal Audio Design, Focal Press, New York, NY, 2015, pp. 1–16.: https://amzn.to/43dVyTa
[3] T. B. Gabrielson, "Mechanical-Thermal Noise in Micromachined Acoustic and Vibration Sensors," in IEEE Transactions on Electron Devices, vol. 40, no. 5, pp. 903-909, May 1993, doi: 10.1109/16.210197: https://ieeexplore.ieee.org/document/210197
[4] Analog Devices, "IEPE-Compatible Interface for Wideband MEMS Accelerometer Sensors," CN-0532, 2020: https://bit.ly/CN-0532
[5] P. Riccardi, "MEMS Based IEPE Powered Instrumentation Microphone," Hackaday.io: https://hackaday.io/project/185762-mems-based-iepe-powered-instrumentation-microphone