bricolons avec le bus CAN

aux commandes des 4000 lampes d'un immeuble

Grâce à la prolifération d'Arduino et à la vague du matériel et du code source libres, utiliser un bus CAN pour expérimenter est désormais un jeu d'enfant. Il est possible d'acheter en ligne du matériel d'interfaçage avec le bus CAN, compatible avec Arduino et bon marché, et de télécharger gratuitement les bibliothèques logicielles de prise en charge. Il ne reste plus qu'à bâtir l'application. Nous vous montrons ici combien c'est devenu facile.

Clemens Valens (labo Elektor)
Sur une idée de **Bera Somnath** (Inde)

Le bus CAN — officiellement *Controller Area Network* — permet d'échanger des messages courts entre les dispositifs connectés. À l'origine c'est un bus de communication fiable pour l'automobile qui n'utilise qu'un minimum de câbles (paire torsadée) ; aujourd'hui il a trouvé sa voie dans toutes sortes d'autres applications.

CAN est un réseau basé sur les messages, pas sur une connexion maître-esclave, pair-à-pair (P2P) ou autre. De ce fait les dispositifs connectés — les nœuds — envoient des messages quand bon leur semble, sans se soucier des récepteurs. La détection des collisions et erreurs ainsi qu'un système de priorisation garantissent que tout fonctionne tant que l'état du trafic sur le réseau demeure normal.

Les nœuds n'ont ni adresses ni identifiants (ID), mais les messages sont typés. Les nœuds n'écoutent ou n'envoient que des messages d'un type donné ; ils ne peuvent pas se parler directement sans ajouter un niveau de plus au protocole. Toutefois, comme un nœud peut demander un type de message donné, et que la spécification de CAN exige que les nœuds utilisent des ID de message uniques, une communication pair-à-pair élémentaire est possible (voir l'encart sur les trames de requête).

Le CAN standard autorise jusqu'à 2048 types de messages (ID de message sur 11 bits), le CAN étendu plus de 500 millions (2^{29} exactement, avec des ID de message sur 29 bits). Pour les deux variantes de CAN, la charge utile peut aller jusqu'à huit octets.

Comme sur le bus CAN, un « 0 » est plus fort (dominant) qu'un « 1 » (récessif), plus l'ID du message est bas, plus haute est sa priorité ; ID=0 surpasse tout. L'expérience a montré que pour obtenir un haut niveau d'utilisation du bus, on doit attribuer les ID selon l'importance du message, pas de son contenu.

Ce qui précède est tout ce que vous devez connaître pour les expériences décrites dans la suite de cet article. Pour

Caractéristiques

- Pas de soudure
- Capacité de plus de 4000 nœuds
- Logiciel facile à comprendre
- Basé sur des modules bon marché

plus d'informations sur le mode de transit des bits sur les fils et autres détails technoïdes, nous recommandons vivement l'internet.

Matériel nécessaire

Nos expériences reposeront sur l'AVR Playground, mais au début n'importe quelle carte compatible Arduino Uno fera l'affaire. Comme le « N » de CAN signifie *Network*, nous aurons besoin d'au moins deux nœuds, concrètement deux cartes AVR Playground. Il faut une interface CAN pour les deux nœuds. Plusieurs solutions existent ; nous en avons choisi une basée sur un contrôleur CAN MCP2515 de Microchip en raison de sa notoriété. Comme l'AVR Playground a un emplacement mikroBUS, nous avons opté pour les cartes CAN SPI click de MikroElektronika (en version 5 V, la version 3,3 V devrait aussi fonctionner) et avons laissé libres pour d'autres usages les connecteurs pour les *shields* Arduino. Vous trouverez en ligne, pour moins de 5 €, des modules presque identiques, mais de forme différente et qui nécessitent des fils de liaison pour leur connexion.
Pour nos expériences plus sérieuses (voir ci-dessous), nous recommandons un de ces *shields* relais à quatre canaux bon marché et très répandus sur l'internet.
Une fois les modules CAN connectés aux cartes compatibles Arduino Uno, il est temps de câbler le réseau. Normalement il faut un câble à paire torsadée, mais pour une faible longueur de simples fils suffisent. On peut trouver des paires torsadées dans des câbles Ethernet par exemple (**fig. 1**).
On doit mettre des terminaisons de bus à chaque extrémité du câble ; les modules CAN ont un cavalier pour cela. Les nœuds intermédiaires *ne* doivent *pas* terminer le câble.

Logiciel nécessaire

D'abord et avant tout vous aurez besoin de votre EDI Arduino.
Pour accéder facilement aux périphériques de l'AVR Playground, installez son paquetage de cartes dans l'EDI Arduino. Reportez-vous à [2] pour les détails. Après l'installation, sélectionnez l'AVR Playground comme carte Arduino et choisissez le port COM correspondant.
Installez aussi la bibliothèque du bus CAN pour Arduino — mcp_can — aimablement mise à notre disposition par SeeedStudio [3].
C'est tout, on y va !

Premiers pas

Nous commençons simplement par vérifier que l'interface CAN peut être initialisée correctement. Les instructions se rapportent au menu principal de l'EDI Arduino.

1. Si besoin, ouvrez un nouveau croquis : Fichier → Nouveau
2. Incluez la bibliothèque CAN : Croquis → Inclure une bibliothèque → mcp_can
3. Ajoutez ce qui suit après les directives `include` mais avant la fonction `setup` :

```
const int SPI_CS_PIN = 10;
MCP_CAN CAN(SPI_CS_PIN);
```

   Ceci suppose que la broche CS de votre interface CAN soit connectée à la broche 10 de l'Arduino.
4. Ajouter ce qui suit à la fonction `setup` (entre les {}) :

```
Serial.begin(115200);
while (CAN_OK!=CAN.
begin(CAN_200KBPS))
// 125 Kbps avec quartz 10 MHz
{
  Serial.print("*");
  delay(100);
}
Serial.println();
Serial.print("CAN init OK.");
```

5. Compilez le croquis et chargez-le dans l'AVR Playground (on vous demandera de le sauvegarder d'abord) : Croquis → Téléverser
6. Ouvrez le moniteur série : Outils → Moniteur série
7. Sélectionnez un débit de 115200 bauds (coin inférieur droit).

Si le texte « CAN init OK. » n'apparaît pas, vérifiez les connexions de votre interface CAN et examinez votre code avec soin.
Si vous pouvez connecter les deux nœuds en même temps à votre PC, répétez les étapes ci-dessus dans une seconde instance de l'EDI Arduino connectée au second nœud. Si vous ne pouvez connecter qu'un nœud à la fois, répétez seulement les étapes 5 et 6. Les deux nœuds devraient s'initialiser correctement (ouf).

Envoyer et recevoir des messages

Pour les expériences suivantes, nous aurons besoin d'au moins deux nœuds connectés au bus CAN. N'oubliez pas de terminer le bus.
Le **listage 1** montre le programme, identique pour tous les nœuds, sauf les valeurs de `my_message_id` et `my_message` qui doivent être uniques pour chaque nœud.

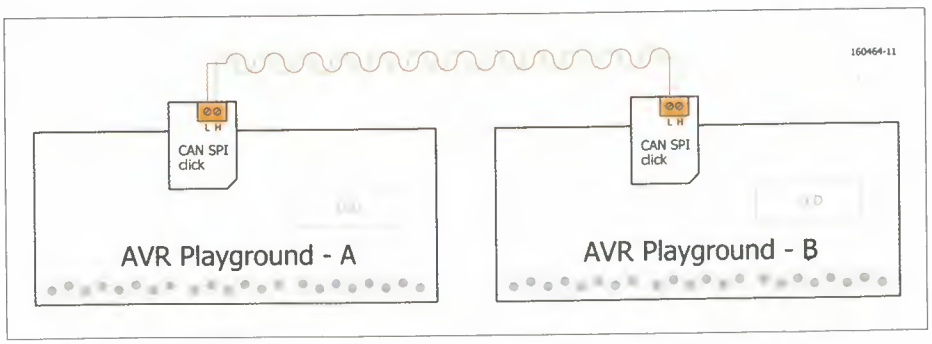

Figure 1. Un CAN élémentaire pour nos premières expériences.

Listage 1. Envoie et reçoit des messages de type texte courts. Utilisez des valeurs différentes pour my_message_id et my_message pour chaque noeud.

```
/*
 * Elektor article 160464 Bricolons avec le bus CAN
 * Envoi et réception de messages
 */

#include <mcp_can.h>
#include <mcp_can_dfs.h>

const int SPI_CS_PIN = 10;
MCP_CAN CAN(SPI_CS_PIN);
const int my_message_id = 100;
const char *my_message = "Node 1";  // 8 caractères max !
uint8_t can_buffer[MAX_CHAR_IN_MESSAGE];

void setup(void)
{
  Serial.begin(115200);
  while (CAN_OK!=CAN.begin(CAN_200KBPS))    // 125 Kbps avec quartz 10 MHz
  {
    Serial.print("*");
    delay(100);
  }
  Serial.println();
  Serial.print("CAN init OK.");
}

void loop(void)
{
  if (CAN_MSGAVAIL==CAN.checkReceive())
  {
    uint8_t len;
    CAN.readMsgBuf(&len,can_buffer);
    uint8_t message_id = CAN.getCanId();
    Serial.println("---");
    Serial.print("Message ID: ");
    Serial.print(message_id);
    Serial.print(", received ");
    Serial.print((int)len);
    Serial.println(" bytes.");
    for (int i=0; i<len; i++)
    {
      Serial.print((char)can_buffer[i]);
    }
    Serial.println();
  }

  CAN.sendMsgBuf(my_message_id,0,strlen(my_message),(uint8_t*)
   my_message);
  delay(1000);
}
```

La fonction `setup` est la même que dans l'exemple précédent, mais `loop` a été remplacée. Elle commence par vérifier si un message a été reçu. Si oui, il est lu et son contenu envoyé sur le port série, accompagné de l'ID du message. Rappelez-vous qu'un message doit être lu avant de pouvoir vérifier son ID avec `getCanId`. On peut utiliser la fonction `readMsgBufID` pour lire en même temps un message et son ID. Qu'un message soit reçu ou non, la fonction se termine en envoyant le message propre au nœud, puis attend une seconde.

Seules les lignes contenant « CAN. » (notez ce point) ont à voir avec l'interface CAN, toutes les autres sont là pour du support et de la (re)présentation de données.

Filtrage de message

Dans l'exemple précédent, la fonction `loop` de tous les nœuds recevait tous les messages circulant sur le bus CAN. C'est bien pour le débogage ou quand l'hôte n'a rien d'autre à faire. En revanche, quand il est sensé réaliser aussi d'autres tâches, devoir inspecter chaque message pour voir s'il contient de l'information utile n'est pas efficace. Il est possible d'améliorer nettement cette situation en exploitant les capacités de filtrage du MCP2515.

Le MCP2515 n'est pas qu'une simple interface de bus CAN, mais un contrôleur CAN avec de nombreuses fonctions. Ainsi il propose des filtres et masques qui peuvent être appliqués aux messages entrants. Seuls les messages qui passent ces filtres seront transmis à l'hôte. Les masques et filtres sont appliqués à l'ID du message, et comme le CAN étendu utilise des ID à 29 bits, les masques et filtres font 29 bits de long. Toutefois, le CAN standard travaille avec des ID à 11 bits, ce qui laisserait la majorité (18) des bits de filtres et masques inutilisés. Pour cette raison, lorsqu'on utilise le CAN standard, 16 des bits de filtres et masques restants sont appliqués aux deux premiers octets de la charge utile. Quand un bit du masque est à 0, le bit correspondant dans l'ID du message est toujours accepté, quel que soit le bit de filtre correspondant. Quand un bit du masque est à 1, le bit correspondant de l'ID du message sera accepté seulement s'il a la même valeur que le bit de filtre correspondant (**fig. 2**). Au démarrage, les masques sont effacés et donc tous les messages seront acceptés.

Le MCP2515 utilise trois tampons pour la réception des messages. Le premier, le *Message Assembly Buffer* ou MAB reçoit tous les messages. Quand un message est complet, il est transféré du MAB vers un des tampons de réception RXB0 ou RXB1. RXB0 a un masque et deux filtres (1 & 2) ; RXB1 a un masque et quatre filtres (3 à 6). Les masques et filtres de RXB0 sont appliqués en premier, donnant à ce tampon une priorité supérieure à RXB1. L'idée est que les messages à haute priorité traversent RXB0 tandis que les messages moins importants transitent par RXB1.

La bibliothèque `mcp_can` que nous utilisons n'autorise pas l'utilisateur à choisir le tampon à lire, elle vérifie simplement RXB0 en premier puis RXB1. Ceci implique que les deux masques doivent être réglés pour éviter de recevoir quand même tous les messages. De plus, la bibliothèque ne prend pas en charge le filtrage des bits de données dans le mode CAN standard.

Pour activer le filtrage de message, ajoutez les lignes suivantes à la fin de la fonction `setup` du précédent exemple :

```
// Activer les masques & filtres
   en mode standard (0).
CAN.init_Mask(0,0,0x3ff);  // RXB0
CAN.init_Mask(1,0,0x3ff);  // RXB1
// Accepter seulement les ID 100
   & 103
CAN.init_Filt(0,0,100);  // RXB0
CAN.init_Filt(1,0,100);  // RXB0
CAN.init_Filt(2,0,103);  // RXB1
CAN.init_Filt(3,0,103);  // RXB1
CAN.init_Filt(4,0,103);  // RXB1
CAN.init_Filt(5,0,103);  // RXB1
```

Ces lignes configurent les masques pour que seuls les filtres déterminent quels messages passent. Ici nous avons dirigé les messages avec ID 100 vers RXB0 et ceux avec ID 103 vers RXB1 (bien que vous ne le verrez pas). Notez que le filtrage cesse dès qu'une correspondance est trouvée, donc seuls les filtres 0 et 2 produiront des correspondances. Pour voir le filtrage fonctionner, vous devez, bien sûr, programmer un nœud pour envoyer ces ID de message et quelques autres. Vous pouvez télécharger depuis [1] un croquis qui vous permet de le faire. Le croquis envoie six messages avec six ID différents et écoute les messages que vous avez spécifiés pour les filtres. Il peut être utilisé par tous les nœuds sur le bus sous réserve que chaque nœud envoie des messages avec des ID uniques. La variable `tx_id_base` permet d'assurer cela. Une `tx_id_base` de nœud doit être choisie pour que les six ID de message qu'il transmet (`tx_id_base` à `tx_id_base+5`) soit uniques sur le bus.

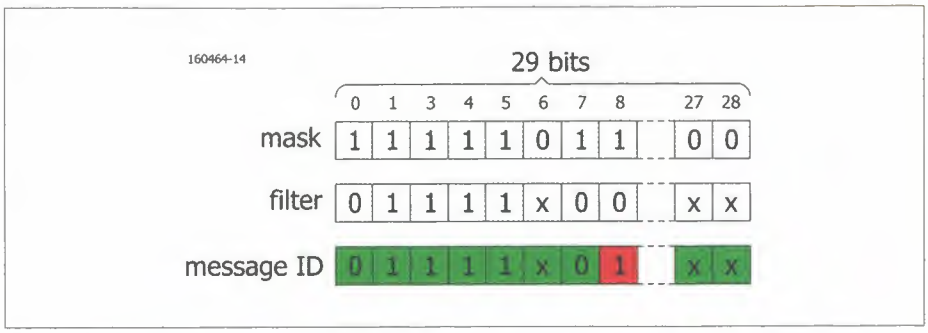

Figure 2. Pour limiter le nombre de messages à lire par l'hôte, le MCP2515 applique d'abord un masque puis jusqu'à six filtres. Si le bit du masque est à 0, peu importent les bits du filtre et du message ; si le bit du masque est à 1, alors le bit du filtre doit être identique à celui du message. Cet ID de message est rejeté à cause du bit 8.

Commande d'éclairage

Parvenus jusqu'ici, nous savons presque tout ce qu'il faut savoir pour créer une application intéressante, il reste à ajouter du sens aux messages et à implémenter des fonctions pour faire des choses utiles, comme activer un relais ou lire un bouton-poussoir.

Pourquoi ne pas mettre en place un CAN dans un immeuble pour commander les lampes dans chaque pièce et à tous les étages ? Tous les interrupteurs de l'immeuble deviendront des nœuds CAN et des nœuds contrôleurs peuvent être placés à quelques endroits stratégiques pour allumer ou éteindre ces lampes. Le concept peut, bien sûr, être étendu aux radiateurs, à l'air conditionné…

Nous simulerons le système avec nos cartes AVR Playground, l'une d'entre elles équipée d'un *shield* relais à 4 canaux (disponible en ligne, env. 10 €) pour commuter les lampes. Les cartes AVR Playground disposent de LCD, buzzers, boutons-poussoirs, LED et potentiomètres utiles pour commander les relais et fournir un retour de ce qui se passe (**fig. 3**).

Mappage des ID de message

Avant de commencer à programmer, nous devons d'abord réfléchir au plan du réseau et au mappage des ID de message. Prenons un immeuble de dix étages, avec dix pièces par étage et dix nœuds (ou prises) par pièce, 1000 ID seront nécessaires pour adresser chaque nœud individuellement. Le CAN standard dispose de 2048 ID, nous avons de la marge. Il faut des ID de messages uniques pour les nœuds autorisés à envoyer des messages (comme des informations d'état). Sinon, un seul ID de message est suffisant pour commuter toutes les lampes individuellement en stockant simplement son adresse et son état dans la charge utile de 8 octets. Supposons que notre système exige que chaque nœud soit capable d'émettre des messages (d'état). Associer 1000 ID de 11 bits à 10 lampes dans 10 pièces

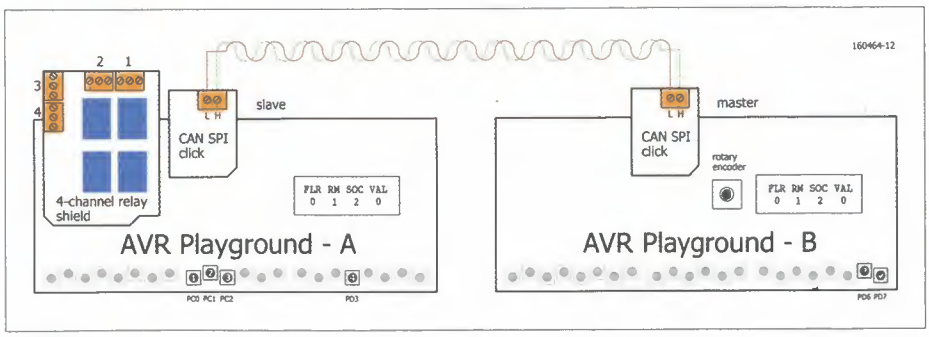

Figure 3. Avec ce système nous pouvons simuler une centrale de commande d'éclairage pour un immeuble à étages.

À propos du débit binaire et des fréquences de quartz

La bibliothèque *mcp_can* considère que le MCP2515 est rythmé par un quartz à 16 MHz. Toutefois, les modules trouvés en ligne peuvent comporter d'autres quartz (par ex. 8 MHz) ; les cartes CAN SPI click que nous avons utilisées fonctionnent à partir de 10 MHz. Mixer ces cartes sur un bus sans ajuster le débit binaire de chaque module dans le micrologiciel conduira à des débits binaires incompatibles sur le réseau. Malheureusement, la bibliothèque *mcp_can* n'offre pas de fonction pour fixer le débit binaire, seules quelques constantes traitées ultérieurement en interne sont accessibles. Mais il y a une bonne nouvelle : sans modifier la bibliothèque, trois débits binaires prédéfinis fonctionnent avec les modules CAN MCP2515 à 8, 10 et 16 MHz (voir le tableau ci-dessous). Utilisez ces constantes à l'appel de la fonction *begin()* de la bibliothèque (voir par ex. listage 1). Si vous tenez à un autre débit binaire, alors vous devrez vous plonger dans la bibliothèque et l'adapter à vos besoins.

	8 MHz	10 MHz	16 MHz
25 Kbps	CAN_50KBPS	CAN_40KBPS	CAN_25KBPS
50 Kbps	CAN_100KBPS	CAN_80KBPS	CAN_50KBPS
125 Kbps	CAN_250KBPS	CAN_200KBPS	CAN_125KBPS

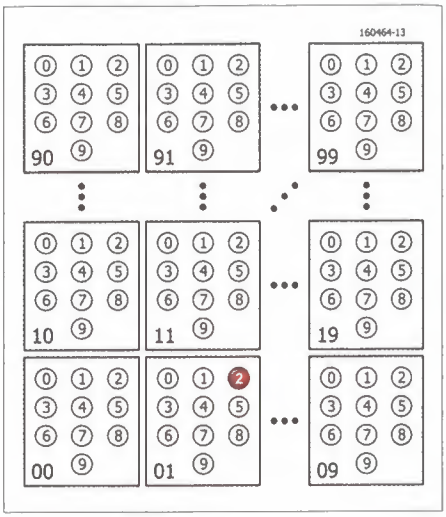

Figure 4. La charge utile <2><1><0><1> du message avec l'ID 0xffff allumera le nœud indiqué.

sur 10 étages nécessiterait une table de correspondance, car il n'est pas facile de parvenir à une association intuitive des ID en les numérotant simplement de 1 à 1000. Un tel schéma introduirait aussi des différences de priorité importantes dans un système où tous les nœuds sont supposés équivalents.

Pour simplifier l'installation, il serait préférable que chaque nœud produise lui-même un ID unique basé sur son emplacement. L'installateur aurait à préciser les numéros d'étage, de pièce et de prise et le nœud ferait le reste. Le CAN étendu avec ses ID à 29 bits peut traiter cela très facilement : réservez simplement des champs de quatre bits pour l'étage, la pièce et la prise et ajoutez un grand décalage. Ceci les repartira tous dans un espace d'adressage restreint (12 bits) avec des priorités équivalentes dans l'énorme réservoir d'ID possibles et il serait facile pour un nœud contrôleur de déterminer où est un nœud. Utilisons donc le CAN étendu avec les adresses de nœud formatées ainsi <n3n2n1n0> <p3p2p1p0><e3e2e1e0> et ajoutons un décalage de, disons 0x10000. Le nœud (prise) 0 dans la pièce 0 de l'étage 0 aurait l'adresse 0x10000 et le nœud (prise) 9 dans la pièce 9 de l'étage 9 aurait l'adresse 0x10999. (Où serait le nœud 0x10123 ? Exactement : prise 1 dans la pièce 2 de l'étage 3). Ceci laisse de l'espace pour quelque 3000 nœuds supplémentaires...

Pour utiliser la bibliothèque `mcp_can` en mode CAN étendu, il suffit de mettre l'argument « ext » (des fonctions qui l'utilisent) à 1 (au lieu de 0).

Un ID supplémentaire (arbitrairement choisi à 0xffff) serait nécessaire pour commander les nœuds. Une commande de commutation préciserait alors l'étage, la pièce et la prise dans sa charge utile, avec l'action de commutation comme ON ou OFF, ou, puisque les LED RVB dominent le monde aujourd'hui, une valeur RVB (ou une vitesse de ventilateur, un niveau de chauffage, etc.). Pour prendre en charge toutes ces options, nous pourrions coder l'action sur trois octets et garder deux octets pour de futures extensions. Ici, par souci de simplicité, nous avons codé l'action sur un octet avec OFF=0 et ON=1 (**fig. 4**).

Bien sûr, commander une lampe à distance c'est bien, mais le vrai pouvoir serait de commander des groupes de lampes simultanément. Pour cela, nous utiliserons la valeur 255 (0xff). Quand l'étage 255 est spécifié, tous les nœuds avec les numéros de pièce et de prise

Tableau 1. On peut commander des groupes en utilisant l'adresse de groupe ou « toutes », 255.

Étage	Pièce	Prise	Action
255	2	9	La prise 9 des pièces 2 de tous les étages répondra
5	255	2	La prise 2 de toutes les pièces de l'étage 5 répondra
0	3	255	Toutes les prises de la pièce 3 de l'étage 0 répondront
255	6	255	Toutes les prises de la pièce 6 de tous les étages répondront
1	255	255	Toutes les prises de toutes les pièces de l'étage 1 répondront.
255	255	255	Toutes les prises de l'immeuble répondront

Tableau 2. Configurations des commutateurs DIP du maître CAN AVR Playground (« H » pour Haut, « B » pour Bas, « M » pour Milieu).

Commutateur DIP	Position (gauche à droite)
S24	MMMMMMMM
S33	HBBHHHMMMM
S15	HBBBHHH
S25	BHHHBBH
S27	MMMHHHHH

Tableau 3. Configuration des commutateurs DIP de l'esclave CAN (multinœuds) AVR Playground (« H » pour Haut, « B » pour Bas, « M » pour Milieu).

Commutateur DIP	Position (gauche à droite)
S24	MMMMMMMM
S33	HBBHHHMMMM
S15	HBBBHHH
S25	BHHHBBH
S27	MMMHMMMM

spécifiés de tous les étages seront commutés. De même, quand la pièce 255 est spécifiée, tous les nœuds avec le numéro de prise spécifié de l'étage en question répondront. Quand l'étage, la pièce et la prise sont tous mis à 255, alors tous les nœuds de l'immeuble écouteront et ils peuvent être commutés (ON ou OFF) tous ensemble. Le **tableau 1** montre quelques exemples pour clarifier le concept.

Programmation

Nous avons implémenté la description ci-dessus dans trois croquis Arduino. Le premier croquis, le maître, à lancer sur l'AVR Playground (voir **tableau 2** pour la configuration de ses commutateurs DIP), permet à l'utilisateur de sélectionner un étage, une pièce et une prise en tournant le codeur rotatif (changement de valeur) et en le poussant (passer au paramètre suivant, noter le curseur « > »), et de changer l'état du nœud en pressant le bouton PD7. On peut changer l'état d'un nœud parce qu'au préalable on a demandé son état actuel. Ceci est réalisé automatiquement dès que le codeur est immobile pendant un moment. Si un nœud ne répond pas à cette demande au bout d'une seconde, parce qu'il n'existe pas par exemple, l'état affiché est « --- » et on entend un son à basse fréquence (si le buzzer est activé). Presser le bouton PD6 enverra aussi une demande d'état. Presser PD7 ne fera rien pour les nœuds « morts ». Lorsqu'un nœud répond, son état est affiché et une tonalité haut-perchée retentit. Si un groupe d'adresses est sélectionné pour un ou plusieurs paramètres (indiqué par « All »), il faut choisir l'état avec le codeur rotatif et presser PD7 pour envoyer la commande.

Esclave multimessages

Le croquis maître est complété par un croquis esclave multinœuds, aussi prévu pour l'AVR Playground (voir **tableau 3** pour la configuration de ses commutateurs DIP) parce qu'il utilise le LCD pour afficher les informations d'état. L'esclave dispose d'un *shield* relais à 4 canaux où chaque relais correspond à un nœud avec ses propres numéros d'étage, de pièce et de prise (à définir au début du croquis dans la structure de données des nœuds). Ces valeurs sont utilisées pour créer l'ID de message d'un nœud.
Chaque nœud a aussi sa propre tonalité, ce qui permet d'utiliser l'esclave multinœuds sans la carte relais.
Les boutons-poussoirs PC0, PC1, PC2 et PD3 (pas PC3) commutent les nœuds et leur état (le nœud le plus récemment mis à jour) est affiché sur le LCD. Chaque fois qu'un nœud change d'état, soit localement par pression d'un bouton soit à distance par le maître, un message de mise à jour d'état est transmis sur le bus CAN. Ceci fournit non seulement un retour au maître, mais permet aussi un balayage total ou partiel du réseau en une seule fois simplement en utilisant la valeur « All » pour un ou plusieurs des paramètres étage, pièce et prise. Comme tous les nœuds vivants du réseau vont répondre à ce message avec une mise à jour d'état, le maître peut maintenant facilement en établir une liste. Nota : cette toute dernière étape sera un exercice pour le lecteur.

Esclave simple

Un second croquis d'esclave a aussi été écrit, beaucoup plus simple et prenant en charge un seul nœud (ou ID de message). Des constantes de configuration figurent en haut de ce croquis, le rendant utilisable avec une carte Arduino Uno (ou compatible) et d'autres modules MCP2515 qui peuvent comporter des quartz différents (c.à.d. fréquences).

Conclusion

Nous avons montré ici comment configurer sans trop d'effort un réseau CAN grâce aux matériel et code source libres et aux modules contrôleurs CAN compatibles MCP2515 largement diffusés. Avec cette approche, nous avons pu nous concentrer sur l'application, un système de commande de l'éclairage d'un immeuble.
Le code présenté dans cet article peut être librement téléchargé en [1].

(160464 – version française : Denis Lafourcade)

LISTE DES COMPOSANTS

Divers
2× carte AVR Playground
2× carte CAN SPI click
1× *shield* relais à 4 canaux
Câble à paire torsadée

Les trames de requête

Le croquis maître envoie des demandes d'état directement à un nœud, avec l'ID (message) du nœud, ce qui signifie qu'il y a deux nœuds utilisant le même ID de message sur le bus, le maître et le nœud lui-même, une situation que la spécification CAN essaye d'éviter. La Trame de Requête (*Remote Frame*), spécifiée par la norme CAN, permet cela. Dans une trame de requête, le bit RTR est récessif alors que dans une trame normale il est dominant. Ainsi, si une trame de données et une trame de requête avec le même ID de message sont transmises en même temps, les données gagneront l'arbitrage, car elles ont une priorité plus haute, le demandeur recevant alors ce qu'il a demandé, tout est pour le mieux.
La principale raison de l'existence des trames de requête est d'éviter que les nœuds inondent le réseau avec des messages qui n'intéressent personne. Cela semble en contradiction avec ce que nous expliquions en début d'article, mais en autorisant les nœuds à ne demander que les données dont ils ont réellement besoin, on réduit le trafic sur le réseau.

Liens

[1] Page de l'article : www.elektormagazine.fr/160464
[2] Carte AVR Playground : www.elektormagazine.fr/labs/avr-playground-129009-2
[3] Bibliothèque CAN : https://github.com/Seeed-Studio/CAN_BUS_Shield