Enfin, je terminerai par le transceiver : ce composant est placé entre le contrôleur CAN et les lignes du bus proprement dit. Il contient l'électronique adéquate pour filtrer le bruit sur la ligne et offrir les impédances d'entrée et de sortie qui permettront un interfaçage propre avec le bus (c'est-à-dire sans perturber le signal et pouvoir transmettre une information qui soit comprise par les autres nœuds du réseau).

Ici, nous avons choisi le MCP2551 de la société Microchip. Il est disponible en version DIP8 (si vous pensez devoir en changer de temps en temps ...) ou en version SO8. Il est très simple d'utilisation, il suffit de suivre les indications de montage de la datasheet du constructeur.

Driver PCM3680 modifié

Le driver que nous allons utiliser est basé sur un driver pour carte Advantech PCM3680, carte PCI à base de contrôleur SJA1000. Comme nous l'avons dit précédemment, ce type de carte se pilote en transmettant des commandes sur le bus PCI à destination de la carte, nous ne pouvons donc piloter directement le contrôleur CAN.

Nous avons adapté notre électronique afin de pouvoir directement piloter un SJA1000 depuis un bus CPU standard avec bus d'adresses et de données séparées.

Finalement les modifications majeures du driver porte sur l'accès aux registres du contrôleur, donc aux routine *can_write_reg* et *can_read_reg*. Nous donnons à présent ici le code de la fonction de lecture :

```
inline unsigned char can_read_reg (
      unsigned char addr) {
   writeb(addr, 0x08000001);
   return (readb(0x08000000));
}
```

À la fin de la compilation, vous obtenez un fichier *can3680.o*. Transférez le sur la cible et chargez le à l'aide de la commande `insmod` :

```
insmod can3680.o
```

Vous obtiendrez alors immédiatement un message d'erreur car il convient de préciser d'autres paramètres :

- `Irq=4,8` (afin de préciser le numéro d'IT utilisé, ce paramètre est à indiquer tel quel mais n'est pas utilisé),
- `CardBaseIO=0xda000`, qui précise l'adresse de base de la carte, et qui n'est pas utilisé par le driver,
- `Baurdate=50` qui permet à l'utilisateur de stipuler une vitesse de bus lors de l'initialisation.

Le driver permet même de piloter 2 composants, l'un depuis */dev/can0* et l'autre depuis */dev/can1*. En fait vous avez pu constater que les routines d'accès au SJA1000 ne possède pas de paramètre quant au numéro du composant désigné, il suffirait toutefois de rajouter un argument pour se faire... et accéder ainsi à 2 contrôleurs CAN. On peut même ainsi étendre le principe à plusieurs adresses et se connecter à plusieurs contrôleurs CAN :

```
0x0e000000 - 0x0e000001 : /dev/can0
0x0e000002 - 0x0e000003 : /dev/can1
0x0e000004 - 0x0e000005 : /dev/can2
...
```

Toutes ces interfaces posséderont le même numéro de majeur mais un numéro de mineur différent.

Posséder plusieurs contrôleurs CAN, c'est se connecter à plusieurs réseaux CAN, ... mais dans quel but me direz-vous et pourquoi faire circuler de l'information sur 2 réseaux plutôt que sur un seul ? Et bien en réalité l'utilisation de 2 réseaux CAN présente l'intérêt d'utiliser l'un des réseaux pour un besoin spécifique, comme pour du debugage à la volée ou encore de la reprogrammation des cibles (mode DIAGNOSTIQUE).

Enfin, nous définissons notre driver afin qu'il possède 2 buffers circulaires par interface, d'une profondeur de 16 messages en sorties et 50 en entrées. Le SJA1000 possède lui-même un certain nombre de buffers permettant de stocker des messages CAN émis ou reçus.

L'emploi de ces 2 couches de buffers nous permettra de pouvoir intercepter toutes les trames d'un bus CAN à 250Kbit/s chargé à 40% par exemple, ce qui en soit est une belle performance !

Détail de la plate-forme logicielle

Nous avons vu précédemment que notre carte était équipée d'un CPU ARM à 60MHz, de 16Mo de SDRAM et de 16Mo de flash NAND pour le stockage des données et des applications.

Elle possède en outre une mémoire flash NOR de 2Mo qui sert de mémoire de boot et dans laquelle est stockée un BIOS.

La séquence de démarrage est la suivante :

- La carte est mise sous tension, et le code du BIOS situé à l'adresse 0 de la mémoire NOR se recopie en RAM et s'exécute : celui-ci initialise les périphériques comme la mémoire et le contrôleur Ethernet,
- LE BIOS possède un mode AUTOBOOT, c'est-à-dire que si aucun caractère n'est détecté sur le port COM0 durant 4 secondes, le code situé à l'adresse *0x10000* est exécuté,
- Le noyau uClinux compressé a été préalablement flashé en *0x10000* dans la mémoire NOR. Ce dernier est donc recopié en RAM puis s'auto-décompresse et s'exécute,
- Durant la phase d'initialisation, le noyau uClinux utilise la zone flashée à partir de l'adresse *0x100000* en NOR comme partition root en lecture seule (romfs). Ensuite uClinux initialise les différents périphériques et monte les partitions,
- Parmi ces partitions, 2 partitions */usr* et */home* sont montées sur la flash NAND de 16Mo, formatée en YAFFS (pour plus de sécurité lors des coupures secteurs intempestives),
- Lorsque uClinux termine sont initialisation, il exécute le script */etc/rc*. Celui-ci teste la présence d'un fichier *rc.sh* sous */home* (NAND). En cas de détection, le script *rc.sh* est exécuté. Cela permet à l'utilisateur de modifier son script de démarrage sans recompiler le romfs à chaque tentative,
- Le script rc.sh charge le driver *can3680.o* en tant que module, et lance automatiquement une application s'il y a lieu en tâche de fond. Finalement il termine son action et rend la main au système,
- UClinux termine son démarrage en lançant un shell (*Busybox*).

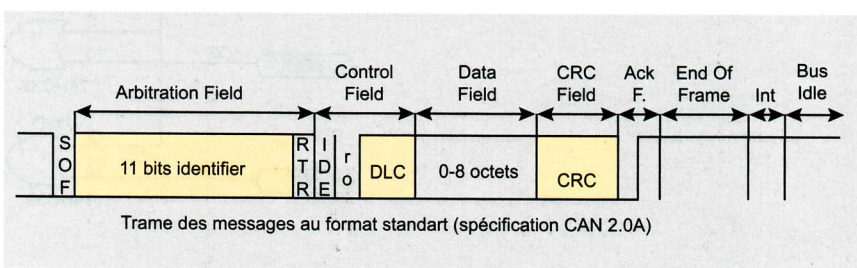

Figure 3. Contenu d'une trame CAN