ceux qui se sentent concernés par ce thème accéderont volontiers aux informations, les autres les ignoreront.

Cet identifiant est d'une longueur de 11 bits dans sa configuration standard (29 bits dans le mode étendu) ce qui permet tout de même 2048 thèmes de message distincts !

Même si l'on ne peut pas parler de temps réel pour le bus CAN, celui-ci a été conçu pour transmettre le plus rapidement possible des données urgentes sur le réseau. Il était donc primordial que le bus CAN véhicule des trames courtes qui occupent très peu le bus (8 octets de données par trame seulement), et le plus rapidement possible (jusqu'à 1Mbit/s).

Le bus CAN est donc composé de petites trames courtes transmises rapidement par tout équipement connecté au bus et désireux de transmettre des informations liées à un thème particulier. Mais alors que ce passe-t-il lorsque plusieurs équipements souhaitent dialoguer sur le bus en même temps ? C'est ce que l'on appelle l'arbitrage de bus : tout comme pour le bus I2C, les lignes CANL et CANH peuvent être considérée comme pilotée par des transistors à collecteurs ouverts avec résistance de pull-up. Pour émettre un *1* il suffit de relâcher le transistor et pour émettre un *0*, il faut rendre le transistor passant. Si 2 équipements sont connectés au bus CAN, il est tout à fait possible pour l'un d'émettre un *1* alors que l'autre cherche à émettre un *0*... et ceci sans endommager le circuit. Dans ce cas c'est le *0* qu'il l'emporte : on dit que le *0* est l'état dominant et le *1* l'état récessif.

Comme chaque équipement qui émet écoute en même temps, il va arriver un moment ou l'un des équipements va s'apercevoir que le bit émis à *1* est lu comme étant à *0*. Dans ce cas l'équipement stoppe sa transmission et tentera une ré-émission ultérieurement.

Si l'on cherche à comprendre plus en détail dans la trame, on remarque qu'une trame de bus CAN est composée de plusieurs champs qui sont les suivants :

- SOF (*Start Of Frame*),
- l'identifiant de la trame (son thème),
- RTR pour une trame de donnée ou de demande de message,
- IDE pour indiquer le mode standard ou étendu,
- 1 bit pour une utilisation future,
- 4 bits destinés à la longueur,
- de 0 à 8 octets de données,
- ACK (*acknowledge*),
- EOF (*End Of Frame*).

Mieux encore, l'électronique des contrôleurs CAN intègre des systèmes de détection d'erreur et envoie des trames d'erreur à destination de l'émetteur lorsqu'une erreur a été détectée. Ce mécanisme est transparent pour l'utilisateur même si ce dernier peut consulter des registres de compteurs d'erreurs qui l'aideront à constater le problème.

Contrôleur de bus CAN

Il existe 2 grands types de contrôleurs de bus CAN, compatible avec la norme v2.0B :

- Les contrôleurs sur bus parallèle tel le bien connu SJA1000,
- Les contrôleurs sur bus série synchrone tel le MCP2515.

Le driver pour MPC2515 existe pour Linux 2.6, mais pas pour uClinux 2.4. Nous pourrions l'adapter pour nos besoins mais nous préférons utiliser finalement le SJA1000 et le driver PCM3680.

Le SJA1000 est habituellement placé sur des cartes PCI, et un contrôleur PCI permet d'accéder à la carte simplement et de piloter le SJA1000 localement. Dans notre cas nous souhaitons interfacer le SJA1000 directement au processeur S3C44 sur son bus système. Or, le SJA1000 ne possède pas des bus data et commande séparés, mais multiplexés. Le composant ne comporte ainsi qu'un seul bus ce qui réduit sa taille. Par contre cela implique aussi l'emploi d'une broche spécifique ALE (*Address Latch Enabled*) qui indique si la donnée présente sur le bus est une adresse.

Le circuit nécessite 3 portes NAND et 4 portes OU, ce qui revient à utiliser 2 boîtiers 74-HC00 et 74HC32. Le principe est le suivant : nous allons simuler le fait que le SJA1000 possède 2 adresses de base, la première pour les données et la seconde pour les adresses.

De façon plus détaillée, vous remarquez l'utilisation du ChipSelect nGCS4, qui sera automatiquement activé lors d'un accès à l'adresse *0x08000000* et au delà (sur une plage de 2Mo). Lorsque l'on écrit à l'adresse *0x08000001*, le bit ADDR0 vaut 1 (bit de poids faible du bus de données), on accède donc au registre d'adresse par l'activation du signal ALE.

Les 4 portes du haut du schéma permettent justement d'activer le signal ALE lorsqu'on accède à cette adresse de base en écriture (nWe actif), autrement c'est le registre de données qui est sélectionné (ALE inactif).

Au final pour écrire dans un des registres du SJA1000, il suffit d'écrire la routine suivante :

```
inline void can_write_reg (unsigned
      char addr, unsigned char data) {
   writeb(addr, 0x08000001);
   writeb(data, 0x08000000);
}
```

On remarque donc qu'il faut accéder à 2 reprises aux registres du contrôleur pour écrire une donnée à une certaine adresse du composant. En *0x08000001* on écrit le registre d'adresse que l'on souhaiterait accéder au sein du contrôleur et en *0x08000000* on écrit la donnée proprement dite.

Cette approche peut paraître un peu surprenante de prime abord, mais elle permet d'éviter d'utiliser des IO libre du CPU (ce qui occasionne en général des pertes de performances) ou un FPGA (gourmand en temps de développement et de validation).

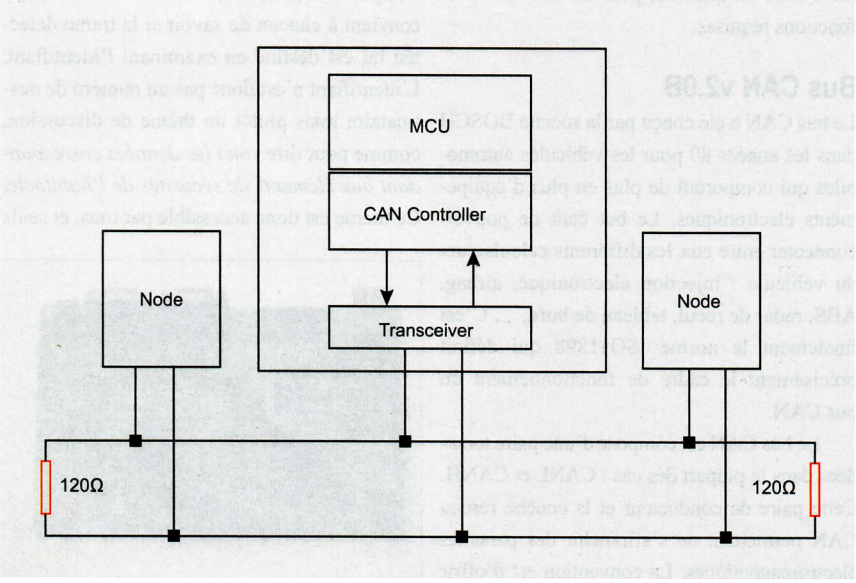

Figure 2. Connexion au bus CAN