Systèmes Embarqués: périphériques et langages

Brique ROSE

Samuel Tardieu
sam@rfc1149.net

École Nationale Supérieure des Télécommunications

Périphériques : introduction

Qu'est-ce qu'un périphérique ?

- Définition : matériel **annexe**, étendant les possibilités du système
- Exemples : disque dur, écran, clavier, souris, carte son, modem ADSL
- En général, inutilisable sans l'unité centrale (exceptions : modems intelligents)
- Connectés à l'ordinateur via un bus (échange de données) et éventuellement des lignes d'interruption (signalement d'un événement)

Exemples de bus

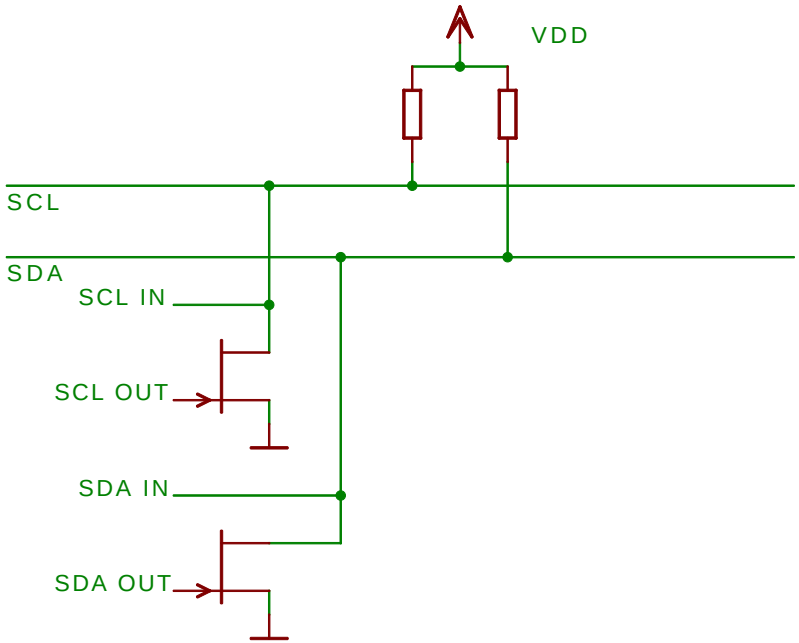
- PCI : bus « local », qui fonctionnait à la même vitesse que le processeur (33MHz) à 32 bits ; maintenant, 133MHz à 64 bits
- Série : bus lent bidirectionnel à un seul périphérique
- I^2C : bus lent avec adressage des périphérique
- CAN : bus peu sensible au bruit
- Ethernet : rapide avec gestion de collision
- DCC : bus lent superposé à l'alimentation

Bus série

- Bus minimal : uniquement deux entités sur le bus
- Bus *full duplex* : les deux parties peuvent émettre en même temps
- Contrôle de flux logiciel (échappement) ou matériel (fils supplémentaires)
- Signaux logiques 1 (-12V, repos) et 0 (12V, actif)

Bus I^2C

- Bus « deux fils » (plus la masse)
- Connexion au bus en mode « collecteur ouvert »

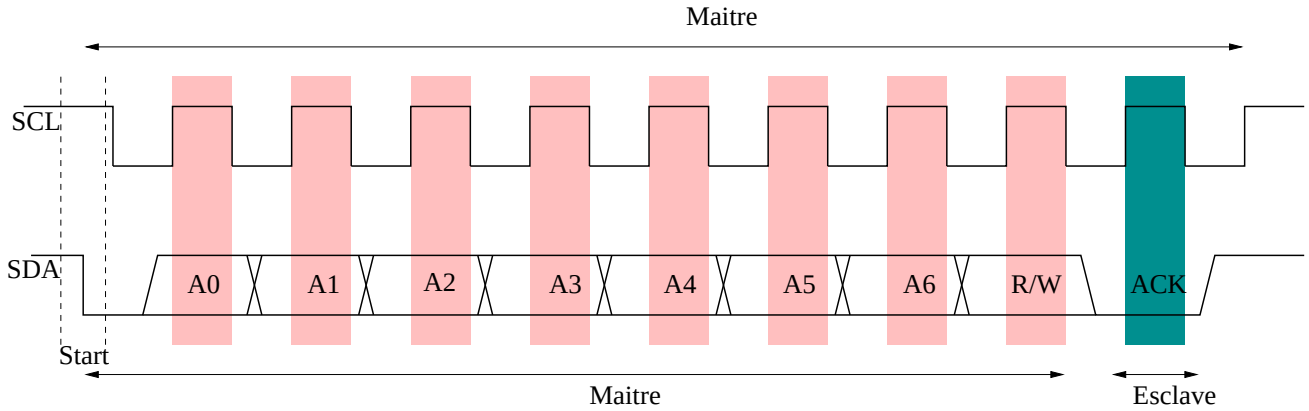


Bus I^2C : signalisation

- Un maître, des esclaves
- Le maître impose l'horloge
- À chaque front haut de l'horloge, la ligne de données transmet un 0 ou un 1
- Descendre la ligne de données lorsque l'horloge est haute est un **start**
- Monter la ligne de données lorsque l'horloge est basse est un **stop**

Bus I²C : protocole

- Le maître envoie un **start** puis un mot de 8 bits : 7 bits d'adresse du périphérique, 1 bit de lecture (1) ou écriture (0)
- Le périphérique signale un **ack** au front suivant, en maintenant la ligne de données basse, s'il a reconnu son adresse



Bus I^2C : écriture

- Le maître présente l'adresse puis le bit R/W à l'état bas
- Le maître transmet un ou plusieurs mots interprétés par l'esclave
- L'esclave les interprète comme il le souhaite (exemple : une adresse pour une EEPROM, ou une commande et des paramètres)

Bus I^{2}C : lecture

- Le maître présente l'adresse puis le bit R/W à l'état haut
- L'esclave renvoie un octet
- Le maître transmet un **ack** pour lire un octet suivant, un **nack** pour arrêter

Bus I^2C : commande bi-directionnelle

Pour effectuer une commande distante :

- Le maître envoie un ordre d'écriture
- Le maître transmet des paramètres à l'esclave
- L'esclave prépare les données
- Le maître envoie un ordre de lecture
- L'esclave renvoie les données préparées

Bus CAN

- Vitesse variable : 1Mbps (40m) à 5kbps (10km)
- Bus différentiel : peu sensible au bruit
- Messages courts (8 octets de données max)
- Diffusion des messages (*broadcast*)
- Très utilisé dans le monde automobile

Bus CAN : signalisation

- Deux états :
  - dominant : transmission d'un zéro (différence de potentiel)
  - récessif : transmission d'un un (pas de ddp, résistance de terminaison du bus)
- Au repos, l'état est récessif
- Pas de retour au zéro entre les bits
- Il suffit d'un élément qui impose l'état dominant pour fixer cet état au bus

Bus CAN : format des messages

Un message est composé de :

- *Data Frame* : voici la donnée X
- *Remote Frame* : qui a la donnée X ?
- *Error Frame* : je n'ai pas bien compris
- *Overload Frame* : je ne suis plus

Bus CAN : Data Frame

Un message de données comprend :

- le champ *Arbitration* : 11 ou 29 bits d'identifiant (prioritisation possible)
- le champ *Data* : de 0 à 8 octets de données
- le *CRC* : 15 bits de somme de contrôle
- un slot pour le *ack* : n'importe quel autre contrôleur CAN peut dire qu'il a correctement reçu le message

Bus CAN : caractéristiques

- Deux nœuds ne doivent pas transmettre le même champ *Arbitration*
- Chaque contrôleur maintient deux compteurs :
  - erreurs d'émission
  - erreurs de réception
- Un contrôleur fautif peut se mettre hors-bus

Bus CAN : exemple

- Dans les voitures modernes, on trouve deux bus CAN :
  - Un bus haute-vitesse contrôlant l'injection, le freinage, l'ABS, l'ESP, le tableau de bord
  - Un bus basse-vitesse contrôlant les équipements annexes (auto-radio), les lumières, les portières
- Avantage : on peut ajouter n'importe quel équipement sur un des bus exploitant les informations existantes

Bus Ethernet

- Utilisé dans les réseaux
- Supporte la gestion de collision
- En cas de collision, attente d'un temps aléatoire avant de retransmettre
- Possibilité de prioritiser en jouant sur ce temps
- Aucune prédictibilité possible

Bus DCC

- Utilisé dans les modèles réduits de train
- But : piloter indépendamment chaque locomotive
- Contraintes :
  - Ne pas tirer d'alimentation supplémentaire (les locomotives ne touchent que deux rails)
  - Ne pas utiliser de radio
  - Permettre un pilotage indépendant des lampes

Bus DCC : principes

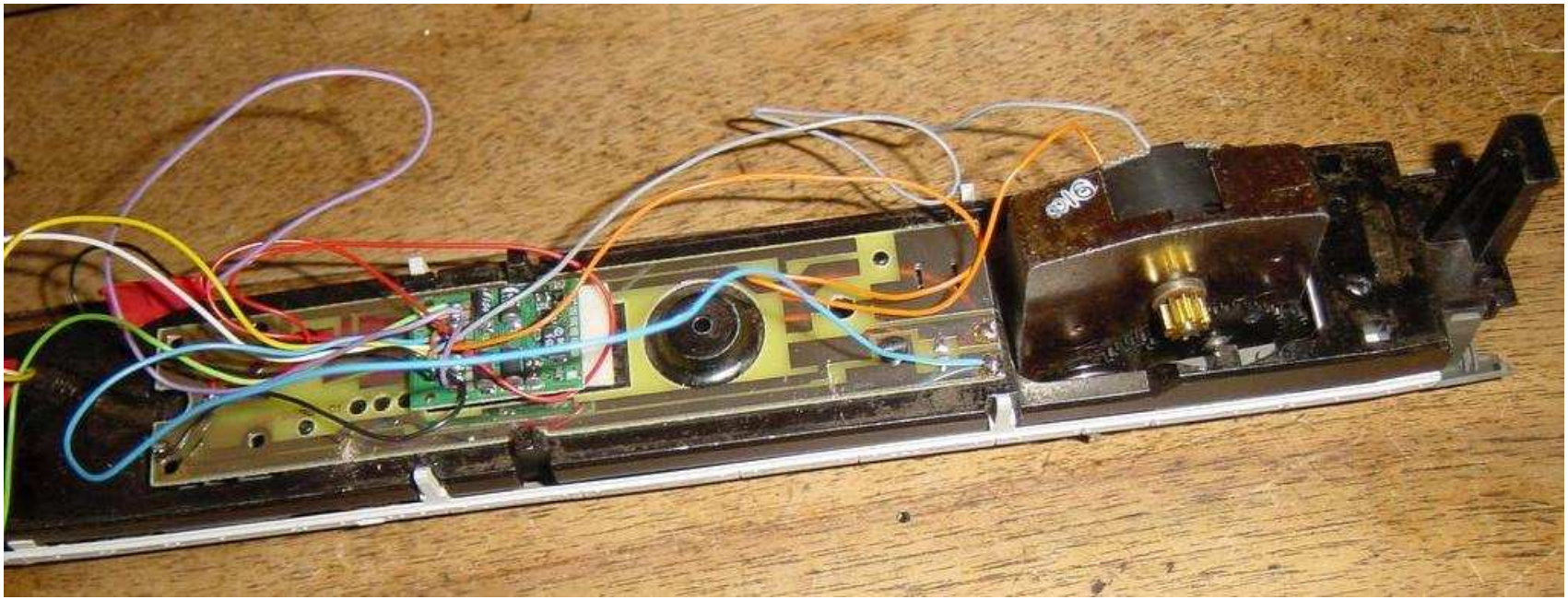
- La différence de potentiel entre les voies est de 15V
- La polarité s'inverse régulièrement
- Un 1 est transmis avec deux inversions de 58 microsecondes chaque
- Un 0 est transmis avec deux inversions de plus de 100 microsecondes chaque

Bus DCC : avantages

- Les véhicules disposent de 15V en permanence
- Un micro-contrôleur est embarqué dans chaque locomotive (en général un PIC) et contrôle la vitesse avec un PWM
- Il est possible de piloter une machine analogique en plus des numériques

Bus DCC : petits décodeurs

Le décodage est très aisé, les décodeurs sont minuscules.



Autres bus

- Bus SPI : bus série synchrone
- Bus USB : bus série avec alimentation
- Bus AGP :
  - bus PCI plus 20 signaux additionnels (132 fils)
  - un seul maître, un seul périphérique, uniquement des entrées/sorties en mémoire
  - permet l'enchaînement des requêtes (*pipelining*)
  - fonctionne à diverses vitesse : en $\times 8$, horloge à 533MHz, bande passante de 2,1Gbps, différence de tension de 0,8V

Gestion des interruptions

- Le bus ne suffit pas toujours
- Problème : comment être prévenu quand un événement a lieu sur un périphérique ?
- Solutions :
  - scruter les données
  - utiliser des interruptions

Principe des interruptions

- Permet d'interrompre la tâche en cours
- Faible latence
- Routine d'interruption : minimiser les traitements effectués
- Pas toujours opportun : dans le cas d'un microcontrôleur dédié, il peut être plus judicieux de scruter un événement

Périphériques sous Linux

- Rappel : séparation entre mode utilisateur et mode noyau
- Partie noyau, deux possibilités :
  - soit réception des interruptions et lecture des données du périphérique
  - soit autorisation pour un programme en mode utilisateur d'accéder à ces données
- Partie utilisateur : traitement des données

Voir `http://www.xml.com/ldd/chapter/book/`

Choix d'un module

- Type « bloc » :
  - Disque
  - Accès à une mémoire flash
- Type « caractère » :
  - Ligne série
  - Capteur de température
- Compilé dans le noyau ou chargé dynamiquement

Module PlaySkool®

```
#define MODULE
#include <linux/module.h>
MODULE_AUTHOR("Samuel Tardieu");
MODULE_DESCRIPTION("My first module");
MODULE_LICENSE("Dual BSD/GPL");

int init_module (void)
{
printk ("Module loading\n");
return 0;
}

void cleanup_module (void)
{
printk ("Module unloading\n");
}
```

Test du module

```
% gcc -O -c playskool.c
% insmod playskool.o
Module loading
% rmmod playskool
Module unloading
% modinfo playskool.o
filename:    ./playskool.o
description: "My first module"
author:      "Samuel Tardieu"
license:     "Dual BSD/GPL"
```

Les macros permettent de stocker certaines informations dans une section spéciale `.modinfo`.

Chargement/déchargement

- Lors du chargement, `init_module` est invoqué et effectue les créations de périphériques (enregistrement majeur/mineur) et enregistre les fonctions associées
- Lors du déchargement, `cleanup_module` désenregistre les opérations et détruit les périphériques.

En profitant de ces opérations automatiques, on s'insère très facilement dans le système.

Exemple : driver I^2C ?

Un pilote (en mode « caractère ») I^2C doit implémenter :

- `write` : envoi de caractères sur le bus I^2C
- `read` : lecture de caractères sur le bus I^2C
- `ioctl` : sélection du périphérique et de la vitesse (stockés en données privées)

Même `open` et `close` ne sont pas obligatoires ; mais ils sont nécessaires pour implémenter du verrouillage.

Stratégie de verrouillage

Si on écrit un driver I^2C, quel est la stratégie de verrouillage à adopter ?

- Verrouiller dans `open` :
  - Pro : pas besoin de verouillage explicite
  - Cons : pas d'accès concurrent possible
- Verrouiller explicitement :
  - Pro : les descripteurs peuvent rester ouverts
  - Cons : il faut verrouiller explicitement

Problèmes liés aux pilotes

- Si on a des créneaux très courts à générer :
  - on peut attendre dans le noyau
  - on peut redonner la main à l'ordonnanceur (en risquant de rater une échéance)
- L'allocation de mémoire est plus compliquée :
  - On peut demander à allouer avec ou sans attente
  - On peut demander des zones spéciales (DMA et mémoire haute)
  - On peut préciser qu'on cherche de la mémoire pour les tampons d'entrée-sortie

Mise au point

Les techniques de mise au point disponibles sont :

- les traces (`printk`)
- les informations dans `/proc`
- la trace des appels systèmes avec `strace`
- `kdb`, le débermineur de noyau

Linux temps-réel

Les périphériques sont souvent non temps-réel. Cependant, ils peuvent être utilisés avec Linux temps-réel :

- un ordonnanceur temps-réel est lancé
- Linux est la tâche de plus basse priorité
- tous les chargements de tâches sont faits depuis Linux
- des files sont utilisées pour la communication entre les tâches temps-réel et les processus Linux

Utilisation de GDB

- GDB peut être utilisé à distance
- Dans la cible, on compile une souche (*stub*) qui traite les commandes de GDB
- Sur l'hôte, on utilise un lien (p.e. série) pour envoyer des commandes à la cible
- GDB peut être utilisé pour charger un programme, on peut l'utiliser comme complément de moniteur
- Il n'est pas nécessaire de charger les symboles sur la cible

Utilisation d'un JTAG

- *Joint Test Action Group*
- Interface pour « parler » à la mémoire (RAM et Flash)
- Sert pour installer un moniteur ou pour le déverminage
- Périphérique en général installé sur le port parallèle
- Peut être vu depuis Linux comme de la mémoire

Quand utiliser un périphérique ?

- Il vaut mieux utiliser un périphérique :
  - pour décharger le processeur principal
  - pour effectuer des fonctions qui demandent un `timing` précis
- Il vaut mieux ne pas utiliser un périphérique :
  - quand le code du pilote est plus compliqué que faire le travail directement
  - quand l'accès au périphérique monopolise le processeur

Les microcontrôleurs

Un microcontrôleur est un microprocesseur qui incorpore des fonctions périphériques supplémentaires, par exemple :

- un ou plusieurs ports série (synchrones/asynchrones)
- un contrôleur I^2C
- un contrôleur CAN
- des convertisseurs analogiques/numériques
- de la mémoire vive, de l'EEPROM ou de la Flash

Mise en veille

Les périphériques peuvent économiser de l'énergie :

- en se mettant en veille
- en se réveillant sur un événement extérieur
- en mettant les périphériques dépendants de lui en veille

Langages de l'embarqué

- Un système embarqué peut être de toute taille
- Il n'y a pas **un** mais **plusieurs** langages utilisés dans le monde de l'embarqué
- Parmi eux, on peut citer : les assembleurs, C, C++, Java, Ada, Forth

Assembleur

- Autant d'assembleur que d'architecture
- Très bas niveau
- Permet un contrôle très fin du temps d'exécution (au niveau de l'instruction)
- N'offre aucun garde-fous
- N'offre que peu de structures complexes
- Toujours disponible

C

- Proche de l'assembleur
- N'a pas les avantages de l'assembleur en terme de finesse
- Offre peu de garde-fous
- Offre des structures complexes
- Quasiment toujours disponible

C++

- Permet d'utiliser des objets facilement
- Offre peu de garde-fous, mais plus que le C
- A en général un surcoût associé à l'utilisation d'objets
- Souvent disponible

Java

- Oblige à utiliser des objets
- Est très peu adapté au temps-réel
- Cache la complexité des opérations effectuées (conversion de nombre en chaîne)
- A une grosse empreinte mémoire
- Parfois disponible

Java : profils réduits

- Bibliothèque réduite
- Multi-tâches réduit
- Aspect graphique quasi-inexistant
- Beaucoup plus adapté à l'embarqué
- J2ME (*Mobile Edition*)

Ada

- Permet d'utiliser des objets facilement
- L'utilisation de méthodes n'est pas coûteux (une méthode n'est **jamais** virtuelle, c'est le type de ses arguments qui la rend virtuelle)
- Offre énormément de garde-fous
- Cache partiellement la complexité des opérations
- Souvent disponible

Ada : profils réduits

- Profils permettant de désactiver des opérations dangereuses ou coûteuses
- Ravenscar : modèle simplifié de multi-tâches, d'où des optimisations très nombreuses sur les accès concurrents
- `pragma No_Runtime` : tout le code est certifiable

Forth

- Permet de développer un langage spécifique au domaine d'application
- Langage très facile à implémenter
- Multi-tâches coopératif très facile à implémenter
- N'offre aucun garde-fou
- Permet d'écrire des programmes tellement simples qu'ils n'ont pas de bugs
- Souvent disponible

Calculs en virgule fixe

- La virgule flottante garantit une précision relative (liée à chaque nombre)
- En embarqué, on utilise souvent les nombres représentés en virgule fixe :
  - précision garantie absolue (par exemple 1mm, liée au type)
  - peu de nombres de grande amplitude (1mm et 10000km sur un robot ?)
  - calculs beaucoup plus rapides (sur des entiers)

Adaptation d'échelle

Exemple : position d'un robot, on souhaite un angle précis à 2 degrés près.

- Comment stocker l'angle ? En degrés, en radians ou autrement ?
- On souhaite calculer un sinus. Comment faire ?
- On souhaite calculer un cosinus. Comment faire ?