Modèle de périphérique

M2 SETI B4 / MS SE SE758

Guillaume Duc
guillaume.duc@telecom-paris.fr
2020–2021

Plan

Présentation

- Le noyau gère, par l'intermédiaire de pilotes, de très nombreux périphériques différents sur un grand nombre de plate-formes matérielles différentes
- Certaines fonctions sont communes à tous les pilotes
  - Déclaration de la liste des périphériques matériels supportés
  - Appel par le noyau lorsqu'un tel périphérique est détecté...
- *Device model* : Abstraction introduite à partir de la branche 2.6 du noyau

Modèle

- Un **périphérique** (*device*) physique est connecté au processeur par l'intermédiaire d'un **bus**
  - Exemple : un périphérique de stockage de masse connecté au bus USB
  - À défaut d'un bus explicite (par exemple pour périphérique directement mappé en mémoire) il existe un bus générique, le bus *platform*
- Ce bus est souvent géré par un **contrôleur** (*controler* ou *adapter*) qui est lui-même un périphérique connecté à un autre bus
  - Par exemple un bus USB est géré par un contrôleur (OHCI, UHCI, EHCI...) de bus USB, lui-même périphérique connecté à un bus PCI

Modèle

- Un périphérique physique est géré par un **pilote** (*driver*)
  - Un même pilote peut gérer plusieurs périphériques physiques (par exemple plusieurs accéléromètres du même modèle, ou des accéléromètres de modèles similaires)
- Plusieurs périphériques rendant les mêmes services peuvent être regroupés en **classes**
  - Exemple la classe *sound* qui regroupe tous les périphériques audio

Structures de données

`#include <linux/device.h>`

- `struct bus_type` : représente un bus
- `struct device_driver` : représente un pilote de périphérique
- `struct device` : représente un périphérique physique
- `struct class` : représente une classe de périphériques
- Ces structures peuvent être spécialisées en fonction du type de bus
  - Périphérique USB : `struct usb_device` (qui contient un champ de type `struct device`)
  - Pilote de périphérique USB : `struct usb_device_driver` (qui contient un champ de type `device_driver`)

Architecture de base d'un pilote

- Fonction d'initialisation
  - Déclaration des périphériques physiques supportés
  - Enregistrement auprès du bus correspondant
- Fonction `probe`
  - Appelée par le noyau lorsqu'un périphérique supporté est détecté
- Fonction `disconnect` (ou `remove`)
  - Appelée par le noyau lorsqu'un périphérique supporté est déconnecté
- Fonction de nettoyage
  - Désenregistrement auprès du bus

Plan

Le bus USB

- Le bus USB permet de brancher et de débrancher à chaud des périphériques
  - Le contrôleur de bus en informe le noyau (souvent via une interruption)
- Il offre également un mécanisme permettant d'identifier les périphériques connectés
  - Chaque périphérique transmet un identifiant du vendeur (*idVendor*) et un identifiant de produit (*idProduct*), chacun sur 16 bits
  - Ces deux identifiants sont utilisés pour faire le lien entre un périphérique physique et un pilote capable de le gérer
- Le noyau est donc automatiquement au courant des périphériques présents sur le bus USB et sait les associer avec un pilote

Un exemple de pilote de périphérique USB

- Étude d'un pilote de périphérique USB simple : LED
- `/drivers/usb/misc/usbled.c`
- Dans la version 4.7 du noyau (a été déplacé depuis vers le module HID)
- On ne s'intéressera ici qu'à la partie du pilote gérant la partie matérielle (on ignorera pour l'instant l'interface avec l'espace utilisateur)

usbled.c (simplifié)

```
/* headers */
static const struct usb_device_id id_table[] = {
        { USB_DEVICE(0x0fc5, 0x1223),
          .driver_info = DELCOM_VISUAL_SIGNAL_INDICATOR },
        { USB_DEVICE(0x1d34, 0x0004),
          .driver_info = DREAM_CHEEKY_WEBMAIL_NOTIFIER },
        /* ... */
        { },
};
MODULE_DEVICE_TABLE(usb, id_table);

static int led_probe(struct usb_interface *interface,
                     const struct usb_device_id *id) { /* ... */ }

static void led_disconnect(struct usb_interface *interface) { /* ... */ }

static struct usb_driver led_driver = {
        .name =         "usbled",
        .probe =        led_probe,
        .disconnect =   led_disconnect,
        .id_table =     id_table,
};

module_usb_driver(led_driver);

MODULE_AUTHOR(DRIVER_AUTHOR);
MODULE_DESCRIPTION(DRIVER_DESC);
MODULE_LICENSE("GPL");
```

Première constatation

```
/* ... */
module_usb_driver(led_driver);

MODULE_AUTHOR(DRIVER_AUTHOR);
MODULE_DESCRIPTION(DRIVER_DESC);
MODULE_LICENSE("GPL");
```

- On retrouve bien les macros `MODULE_AUTHOR`, `MODULE_DESCRIPTION` et `MODULE_LICENSE`
- Mais où sont les fonctions d'initialisation et de nettoyage ?

Première constatation

```
/* ... */
module_usb_driver(led_driver);

MODULE_AUTHOR(DRIVER_AUTHOR);
MODULE_DESCRIPTION(DRIVER_DESC);
MODULE_LICENSE("GPL");
```

- On retrouve bien les macros `MODULE_AUTHOR`, `MODULE_DESCRIPTION` et `MODULE_LICENSE`
- Mais où sont les fonctions d'initialisation et de nettoyage ?
- Elles sont cachées sous la macro `module_usb_driver`

Fonctions d'initialisation et de nettoyage

- La macro `module_usb_driver` génère le code suivant

```
static int __init led_driver_init(void)
{
        return usb_register(&(led_driver));
}
module_init(led_driver_init);

static void __exit led_driver_exit(void)
{
        usb_deregister(&(led_driver));
}
module_exit(led_driver_exit);
```

- Dans un pilote, en général, les fonctions d'initialisations et de nettoyage se contentent d'enregistrer le pilote auprès du sous-système approprié (ici USB)
- D'où l'utilisation pratique d'une macro pour les générer automatiquement

Enregistrement du pilote

- La fonction `usb_register` est appelée avec la structure `led_driver`

```
static struct usb_driver led_driver = {
        .name =         "usbled",
        .probe =        led_probe,
        .disconnect =   led_disconnect,
        .id_table =     id_table,
};
```

- Cette structure est une spécialisation de la structure `struct device_driver` et représente un pilote de périphérique USB

Enregistrement du pilote

```
static struct usb_driver led_driver = {
        .name =         "usbled",
        .probe =        led_probe,
        .disconnect =   led_disconnect,
        .id_table =     id_table,
};
```

- `name` : nom du pilote
- `probe` : pointeur vers la fonction à appeler lorsqu'un périphérique physique que le pilote sait gérer est détecté
- `disconnect` : pointeur vers la fonction à appeler lorsqu'un périphérique physique géré par le pilote est déconnecté
- `id_table` : table permettant d'identifier les périphériques USB que notre pilote est capable de gérer

Identification des périphériques gérés

```
static const struct usb_device_id id_table[] = {
        { USB_DEVICE(0x0fc5, 0x1223),
          .driver_info = DELCOM_VISUAL_SIGNAL_INDICATOR },
        { USB_DEVICE(0x1d34, 0x0004),
          .driver_info = DREAM_CHEEKY_WEBMAIL_NOTIFIER },
        /* ... */
        { },
};
MODULE_DEVICE_TABLE(usb, id_table);
```

- `id_table` liste les périphériques que le pilote sait gérer
- Par exemple, le pilote annonce savoir gérer un périphérique USB dont l'identifiant du vendeur est `0x0fc5` et l'identifiant du produit est `0x1223`
- Donc si un tel périphérique est détecté sur le bus, le noyau en informera le pilote (en appelant la fonction `led_probe`)

Identification des périphériques gérés

```
static const struct usb_device_id id_table[] = {
        { USB_DEVICE(0x0fc5, 0x1223),
          .driver_info = DELCOM_VISUAL_SIGNAL_INDICATOR },
        { USB_DEVICE(0x1d34, 0x0004),
          .driver_info = DREAM_CHEEKY_WEBMAIL_NOTIFIER },
        /* ... */
        { },
};
MODULE_DEVICE_TABLE(usb, id_table);
```

- Le champ `driver_info` permet à notre pilote, lors de la détection d'un périphérique, de savoir quelle entrée de la table correspond
  - Utile par exemple si chacun des périphériques doit être géré différemment

Identification des périphériques gérés

```
static const struct usb_device_id id_table[] = {
        { USB_DEVICE(0x0fc5, 0x1223),
          .driver_info = DELCOM_VISUAL_SIGNAL_INDICATOR },
        { USB_DEVICE(0x1d34, 0x0004),
          .driver_info = DREAM_CHEEKY_WEBMAIL_NOTIFIER },
        /* ... */
        { },
};
MODULE_DEVICE_TABLE(usb, id_table);
```

- La macro `MODULE_DEVICE_TABLE` permet d'exporter ces informations vers l'espace utilisateur (via l'outil `depmod` et le fichier `/lib/modules/xxx/modules.alias`)
- Ce fichier est utilisé par `udev` pour charger automatiquement le module approprié lors de la détection d'un périphérique

Détection d'un périphérique géré

- Lorsqu'un nouveau périphérique est branché sur le bus USB (ou lorsqu'un nouveau pilote est enregistré), le noyau va chercher si un pilote sait gérer ce nouveau périphérique
- Si c'est le cas, le noyau va appeler la fonction déclarée dans le champ `probe` de la structure `usb_driver` du pilote correspondant, ici `led_probe`

Détection d'un périphérique géré

```
static int led_probe(struct usb_interface *interface,
                     const struct usb_device_id *id) { /* ... */ }
```

- Cette fonction a plusieurs objectifs
  - Vérifier que le pilote sait effectivement gérer le périphérique
  - Initialiser le périphérique et les structures de données nécessaires pour le gérer (gestionnaire d'interruption...)
  - Mettre en place la communication avec l'espace utilisateur (enregistrement auprès d'un *framework* particulier...)
- Elle renvoie 0 si tout s'est bien passé ou un code d'erreur négatif
- Elle peut être appelée plusieurs fois si plusieurs périphériques différents (exemple, deux LED USB) sont présents

Déconnexion d'un périphérique géré

```
static void led_disconnect(struct usb_interface *interface) { /* ... */ }
```

- Lorsqu'un périphérique géré par notre pilote est déconnecté, le noyau va appeler la fonction déclarée dans le champ `disconnect` de la structure `usb_driver` du pilote correspondant, ici `led_disconnect`
- Cette fonction va devoir libérer les ressources allouées pour ce périphérique

Pour conclure

- Le pilote d'un périphérique USB est composé de trois parties importantes
  - Une table pour déclarer les périphériques qu'il sait gérer
  - Une fonction `probe` appelée par le noyau lorsqu'un périphérique physique est détecté
  - Une fonction `disconnect` appelée par le noyau lorsqu'un périphérique physique est déconnecté
- L'association périphérique physique / pilote est simple car le bus USB permet d'identifier quels sont les périphériques connectés

Plan

Problématique

- Certains bus n'offrent pas la possibilité d'identifier quels composants y sont connectés
  - Exemples : bus SPI, bus I^2C...
- Comment le noyau peut-il savoir qu'un périphérique est connecté ?
- Comment le noyau peut-il faire l'association entre un périphérique et le pilote correspondant ?

Solutions

- Le noyau ne peut pas obtenir ces informations seul
- Deux mécanismes existent pour lui fournir ces informations
  - Description de la plate-forme en dur dans le noyau
  - Arbre des périphériques (*Device Tree*)

Exemple

- Périphérique I^{2}C Seiko Instruments S-35390A
- Horloge temps réel (*real-time clock*, RTC)
- `/drivers/rtc/rtc-s35390a.c`, version 4.18

rtc-s35390a.c (simplifié)

```
static const struct i2c_device_id s35390a_id[] = {
        { "s35390a", 0 },
        { }
};
MODULE_DEVICE_TABLE(i2c, s35390a_id);

static const struct of_device_id s35390a_of_match[] = {
        { .compatible = "s35390a" },
        { .compatible = "sii,s35390a" },
        { }
};
MODULE_DEVICE_TABLE(of, s35390a_of_match);

static int s35390a_probe(struct i2c_client *client,
                         const struct i2c_device_id *id) { /* ... */ }
static int s35390a_remove(struct i2c_client *client) { /* ... */ }

static struct i2c_driver s35390a_driver = {
        .driver         = {
                .name   = "rtc-s35390a",
                .of_match_table = of_match_ptr(s35390a_of_match),
        },
        .probe          = s35390a_probe,
        .remove         = s35390a_remove,
        .id_table       = s35390a_id,
};
module_i2c_driver(s35390a_driver);
```

rtc-s35390a.c (simplifié)

```
static int s35390a_probe(struct i2c_client *client,
                         const struct i2c_device_id *id) { /* ... */ }
static int s35390a_remove(struct i2c_client *client) { /* ... */ }

static struct i2c_driver s35390a_driver = {
        .driver         = {
                .name   = "rtc-s35390a",
                .of_match_table = of_match_ptr(s35390a_of_match),
        },
        .probe          = s35390a_probe,
        .remove         = s35390a_remove,
        .id_table       = s35390a_id,
};
module_i2c_driver(s35390a_driver);
```

- On retrouve :
  - La macro permettant de créer les fonctions d'initialisation et de nettoyage
  - La structure représentant le pilote
  - Les fonctions `probe` et `remove`

rtc-s35390a.c (simplifié)

```
static const struct i2c_device_id s35390a_id[] = {
        { "s35390a", 0 },
        { }
};
MODULE_DEVICE_TABLE(i2c, s35390a_id);

static const struct of_device_id s35390a_of_match[] = {
        { .compatible = "s35390a" },
        { .compatible = "sii,s35390a" },
        { }
};
MODULE_DEVICE_TABLE(of, s35390a_of_match);

static struct i2c_driver s35390a\_driver = {
        .driver         = {
                .name   = "rtc-s35390a",
                .of_match_table = of_match_ptr(s35390a_of_match),
        },
        .probe          = s35390a_probe,
        .remove         = s35390a_remove,
        .id_table       = s35390a_id,
};
```

- Ces deux structures correspondent aux deux mécanismes décrits précédemment pour l'identification des périphériques

Déclaration statique

- Fichier source décrivant programmatiquement les périphériques présents et leurs caractéristiques sur une plate-forme
- La plate-forme est choisie lors de la configuration du noyau (ou identifiée par le bootloader qui fournit alors l'information au noyau)

Déclaration statique

/arch/arm/mach-imx/mach-armadillo5x0.c (extraits)

```
static struct i2c_board_info armadillo5x0_i2c_rtc = {
        I2C_BOARD_INFO("s35390a", 0x30),
};

static void __init armadillo5x0_late(void)
{
        /* ... */
        i2c_register_board_info(1, &armadillo5x0_i2c_rtc, 1);
        /* ... */
}

MACHINE_START(ARMADILLO5X0, "Armadillo-500")
        /* ... */
        .init_late      = armadillo5x0_late,
MACHINE_END
```

Déclaration statique

/arch/arm/mach-imx/mach-armadillo5x0.c (extraits)

```
static struct i2c_board_info armadillo5x0_i2c_rtc = {
        I2C_BOARD_INFO("s35390a", 0x30),
};

static void __init armadillo5x0_late(void)
{
        /* ... */
        i2c_register_board_info(1, &armadillo5x0_i2c_rtc, 1);
        /* ... */
}

MACHINE_START(ARMADILLO5X0, "Armadillo-500")
        /* ... */
        .init_late      = armadillo5x0_late,
MACHINE_END
```

Déclaration statique
Association

- /drivers/rtc/rtc-s35390a.c

```
static const struct i2c_device_id s35390a_id[] = {
        { "s35390a", 0 },
        { }
};
MODULE_DEVICE_TABLE(i2c, s35390a_id);

static struct i2c_driver s35390a_driver = {
        .id_table       = s35390a_id,
        /* ... */
};
```

- /arch/arm/mach-imx/mach-armadillo5x0.c

```
static struct i2c_board_info armadillo5x0_i2c_rtc = {
        I2C_BOARD_INFO("s35390a", 0x30),
};
```

Déclaration statique

Inconvénients

- La configuration de la plate-forme est fixée en dur dans le noyau
- Nécessité de recompiler le noyau en cas de modification des périphériques présents
- Manque de souplesse

Arbre des périphériques (*device tree*)
Introduction

- Structure de donnée arborescente décrivant le matériel présent sur une plate-forme
- Spécifications développées par le projet *Open Firmware* et dérivées des machines à base d'architecture SPARC
- Fichier source (*Device Tree Source*, DTS) « compilé » sous forme binaire (*Device Tree Blob*, DTB) par l'outil `dtc` (*Device Tree Compiler*) fourni par le noyau
- Le DTB est fourni au noyau lors du démarrage

Arbre des périphériques (*device tree*)

`/arch/arm/boot/dts/r8a7740-armadillo800eva.dts` (extraits)

```
/dts-v1/;
/* #include ... */

&i2c2 {
        status = "okay";
        rtc@30 {
                compatible = "sii,s35390a";
                reg = <0x30>;
        };
};
```

Arbre des périphériques (*device tree*)

`/arch/arm/boot/dts/r8a7740-armadillo800eva.dts` (extraits)

```
/dts-v1/;
/* #include ... */

&i2c2 {
        status = "okay";
        rtc@30 {
                compatible = "sii,s35390a";
                reg = <0x30>;
        };
};
```

- Fait référence à un nœud défini ailleurs dans l'arbre

Arbre des périphériques (*device tree*)

`/arch/arm/boot/dts/r8a7740-armadillo800eva.dts` (extraits)

```
/dts-v1/;
/* #include ... */

&i2c2 {
        status = "okay";
        rtc@30 {
                compatible = "sii,s35390a";
                reg = <0x30>;
        };
};
```

- On ajoute à ce nœud un nœud fils

Arbre des périphériques (*device tree*)

`/arch/arm/boot/dts/r8a7740-armadillo800eva.dts` (extraits)

```
/dts-v1/;
/* #include ... */

&i2c2 {
        status = "okay";
        rtc@30 {
                compatible = "sii,s35390a";
                reg = <0x30>;
        };
};
```

- La propriété `compatible` va permettre de faire la relation entre un périphérique et le pilote capable de le gérer

Arbre des périphériques (*device tree*)

`/arch/arm/boot/dts/r8a7740-armadillo800eva.dts` (extraits)

```
/dts-v1/;
/* #include ... */

&i2c2 {
        status = "okay";
        rtc@30 {
                compatible = "sii,s35390a";
                reg = <0x30>;
        };
};
```

- La propriété `reg` va indiquer l'adresse du périphérique

Arbre des périphériques (*device tree*)

Association

- /drivers/rtc/rtc-s35390a.c

```
static const struct of_device_id s35390a_of_match[] = {
        { .compatible = "s35390a" },
        { .compatible = "sii,s35390a" },
        { }
};
MODULE_DEVICE_TABLE(of, s35390a_of_match);

static struct i2c_driver s35390a_driver = {
        .driver         = {
                .name   = "rtc-s35390a",
                .of_match_table = of_match_ptr(s35390a_of_match),
        },
        /* ... */
};
```

- /arch/arm/boot/dts/r8a7740-armadillo800eva.dts

```
&i2c2 {
        rtc@30 {
                compatible = "sii,s35390a";
                reg = <0x30>;
        };
};
```

Arbre des périphériques (*device tree*)
Avantages

- Modifiable sans recompiler le noyau
- Une même image du noyau pour plusieurs plate-formes
- Modifiable dynamiquement (mécanisme d'*overlay*)
- Obligatoire pour les systèmes sur puce à base d'ARM, utilisables sur d'autres architectures (x86, SPARC, PowerPC...)

rtc-s35390a.c (simplifié)

Deux mécanismes présents

```
static const struct i2c_device_id s35390a_id[] = {
        { "s35390a", 0 },
        { }
};
MODULE_DEVICE_TABLE(i2c, s35390a_id);

static const struct of_device_id s35390a_of_match[] = {
        { .compatible = "s35390a" },
        { .compatible = "sii,s35390a" },
        { }
};
MODULE_DEVICE_TABLE(of, s35390a_of_match);

static struct i2c_driver s35390a_driver = {
        .driver         = {
                .name   = "rtc-s35390a",
                .of_match_table = of_match_ptr(s35390a_of_match),
        },
        .probe          = s35390a_probe,
        .remove         = s35390a_remove,
        .id_table       = s35390a_id,
};
module_i2c_driver(s35390a_driver);
```

rtc-s35390a.c (simplifié)

Deux mécanismes présents

```
static const struct i2c_device_id s35390a_id[] = {
        { "s35390a", 0 },
        { }
};
MODULE_DEVICE_TABLE(i2c, s35390a_id);

static const struct of_device_id s35390a_of_match[] = {
        { .compatible = "s35390a" },
        { .compatible = "sii,s35390a" },
        { }
};
MODULE_DEVICE_TABLE(of, s35390a_of_match);

static struct i2c_driver s35390a_driver = {
        .driver         = {
                .name   = "rtc-s35390a",
                .of_match_table = of_match_ptr(s35390a_of_match),
        },
        .probe          = s35390a_probe,
        .remove         = s35390a_remove,
        .id_table       = s35390a_id,
};
module_i2c_driver(s35390a_driver);
```

Résumé

- Deux mécanismes permettent l'association entre un périphérique physique présent sur une machine et le pilote correspondant, si la découverte des périphériques ne peux pas être automatisée
  - Déclaration programmatique dans les sources du noyau spécifique à une plate-forme
  - Arbre des périphériques
- Pour les deux, comparaison de chaînes de caractères
- Les deux mécanismes sont souvent présents dans le code des pilotes

Plan

Le bus I^2C

Présentation

- *Inter-Integrated Circuit*
- Conçu en 1982 par *Philips Semiconductor*
- Utilisé pour connecter, sur de courtes distances et à bas débit, des périphériques à un microprocesseur ou à un microcontrôleur
  - Exemples : capteur de température, accéléromètre, mémoire flash...

Squelette d'un pilote de périphérique I²C
Exemple

- Périphérique I²C Seiko Instruments S-35390A
- Horloge temps réel (*real-time clock*, RTC)
- `/drivers/rtc/rtc-s35390a.c`, version 4.18

rtc-s35390a.c (simplifié)

```
static const struct i2c_device_id s35390a_id[] = {
        { "s35390a", 0 },
        { }
};
MODULE_DEVICE_TABLE(i2c, s35390a_id);

static const struct of_device_id s35390a_of_match[] = {
        { .compatible = "s35390a" },
        { .compatible = "sii,s35390a" },
        { }
};
MODULE_DEVICE_TABLE(of, s35390a_of_match);

static int s35390a_probe(struct i2c_client *client,
                         const struct i2c_device_id *id) { /* ... */ }
static int s35390a_remove(struct i2c_client *client) { /* ... */ }

static struct i2c_driver s35390a_driver = {
        .driver         = {
                .name   = "rtc-s35390a",
                .of_match_table = of_match_ptr(s35390a_of_match),
        },
        .probe          = s35390a_probe,
        .remove         = s35390a_remove,
        .id_table       = s35390a_id,
};
module_i2c_driver(s35390a_driver);
```

rtc-s35390a.c (simplifié)

Association statique

```
static const struct i2c_device_id s35390a_id[] = {
        { "s35390a", 0 },
        { }
};
MODULE_DEVICE_TABLE(i2c, s35390a_id);

static struct i2c_driver s35390a_driver = {
        .id_table       = s35390a_id,
        /* ... */
};
```

- Cette structure sert à l'association statique
  - Chaque entrée contient deux champs : une chaîne de caractères et un entier
  - La dernière entrée doit être vide
  - Elle est associée au pilote grâce au champ `id_table` de la structure `i2c_driver`
  - La macro `MODULE_DEVICE_TABLE` permet d'exporter ces informations vers l'espace utilisateur

rtc-s35390a.c (simplifié)

Association avec un arbre des périphériques

```
static const struct of_device_id s35390a_of_match[] = {
        { .compatible = "s35390a" },
        { .compatible = "sii,s35390a" },
        { }
};
MODULE_DEVICE_TABLE(of, s35390a_of_match);

static struct i2c_driver s35390a_driver = {
        .driver         = {
                .of_match_table = of_match_ptr(s35390a_of_match), /* ... */
        }, /* ... */
};
```

- Cette structure sert à l'association à l'aide d'un arbre des périphérique
  - Chaque entrée contient notamment un champ `compatible`
  - La dernière entrée doit être vide
  - Elle est associée au pilote grâce au champ `of_match_table` de la structure `i2c_driver`
  - La macro `MODULE_DEVICE_TABLE` permet d'exporter ces informations vers l'espace utilisateur

rtc-s35390a.c (simplifié)

Fonction probe

```
static int s35390a_probe(struct i2c_client *client,
                         const struct i2c_device_id *id) { /* ... */ }

static struct i2c_driver s35390a_driver = {
        .probe          = s35390a_probe,
        /* ... */
};
```

- Appelée lorsqu'un périphérique supporté est détecté
- Arguments
  - `client` : représente le périphérique I^2C
  - `id` : pointeur vers l'entrée du tableau `s35390a_id` correspondant au périphérique (`NULL` si c'est l'arbre des périphérique qui a été utilisé)
- Retourne `0` si tout se passe bien, un nombre négatif en cas d'erreur

rtc-s35390a.c (simplifié)

Fonction probe

- Il est possible de récupérer les informations privées
  - Champ `driver_data` de la structure `i2c_device_id`
  - Champ `data` de la structure `of_device_id`

```
if (client->dev.of_node) {
    /* Device tree */
    const struct of_device_id *of_id =
        of_match_device(s35390a_of_match, &client->dev);
    variant = of_id->data;      /* Champ data de of_device_id */
} else {
    /* Association statique */
    variant = id->driver_data;  /* Champ driver_data de i2c_device_id */
}
```

rtc-s35390a.c (simplifié)

Fonction remove

```
static int s35390a_remove(struct i2c_client *client) { /* ... */ }

static struct i2c_driver s35390a_driver = {
        .remove         = s35390a_remove,
        /* ... */
};
```

- Appelée lors de la « déconnexion » du périphérique
  - Déchargement du module
  - Modification de l'arbre des périphériques

rtc-s35390a.c (simplifié)

Déclaration du pilote

```
static struct i2c_driver s35390a_driver = {
        .driver         = {
                .name   = "rtc-s35390a",
                .of_match_table = of_match_ptr(s35390a_of_match),
        },
        .probe          = s35390a_probe,
        .remove         = s35390a_remove,
        .id_table       = s35390a_id,
};
module_i2c_driver(s35390a_driver);
```

- `name` : nom du pilote (le même que le nom du module)
- La macro `module_i2c_driver` crée automatiquement les fonctions d'initialisation et de nettoyage

Plan

Le bus I²C

Transmissions

- Toujours à l'initiative d'un maître
- Adresses sur 7 bits
- Début d'un échange
  - Maître : `START` + 7 bits d'adresse + `R(1)/W(0)`
  - Esclave (si présent) : `ACK`
- Si écriture (maître vers esclave)
  - Maître : un octet de données
  - Esclave : `ACK` ou `NACK`...
- Si lecture (esclave vers maître)
  - Esclave : un octet de données
  - Maître : `ACK` (poursuite) ou `NACK` (fin)...
- Maître : `STOP`

API de base maître

Écriture (maître vers esclave)

```
int i2c_master_send(const struct i2c_client *client,
                    const char *buf, int count);
```

- `client` : structure représentant le périphérique esclave
- `buf` : pointeur vers le tampon contenant les données à transmettre à l'esclave
- `count` : nombre d'octets à transmettre
- Retourne le nombre d'octets transmis ou un code d'erreur négatif

API de base maître

Lecture (esclave vers maître)

```
int i2c_master_recv(const struct i2c_client *client,
                    char *buf, int count);
```

- `client` : structure représentant le périphérique esclave
- `buf` : pointeur vers le tampon où seront stockées les données reçues
- `count` : nombre d'octets à recevoir
- Retourne le nombre d'octets reçus ou un code d'erreur négatif

API maître

Transfert de messages

```
int i2c_transfer(struct i2c_adapter *adap, struct i2c_msg *msg
                 int num);
```

- `adap` : structure représentant le contrôleur I²C (peut être récupérée à partir du client : `client->adapter`)
- `msgs` : tableau de messages
- `num` : nombre de messages
- Retourne le nombre de messages transférés ou un code d'erreur négatif

API maître

Transfert de messages

```
struct i2c_msg {
        __u16 addr;          /* slave address */
        __u16 flags;
        __u16 len;           /* msg length */
        __u8 *buf;           /* pointer to msg data */
};
#define I2C_M_RD 0x0001      /* read data, from slave to master
```

- `addr` : adresse de l'esclave (peut être récupérée à partir du client : `client->addr`)
- `flags` : drapeaux (le plus important étant `I2C_M_RD` pour faire une lecture)
- `len` : nombre de données à lire ou écrire
- `buf` : pointeur vers le tampon contenant les données à écrire ou, en cas d'écriture, où seront stockées les données lues

À vous de jouer

- Vous allez écrire le début d'un pilote de périphérique pour un accéléromètre I^2C
- Ce périphérique est émulé par QEMU