Systèmes embarqués 4

J.-M Friedt

FEMTO-ST/département temps-fréquence

jmfriedt@femto-st.fr

transparents à `jmfriedt.free.fr`

6 décembre 2016

Interfaces de communication

- Le noyau accède au matériel et garantit l'intégrité des accès
- L'espace utilisateur (plusieurs processus) accède aux ressources au travers des requêtes au noyau
- Interface "classique" : `/dev` où "tout" est fichier [1]
- Interface uniforme en terme de **périphérique** représenté : `/sys/class` (PWM, IIO, ...) et `/sys/bus` (PCI, USB, SPI ...) : pas de nouvel IOCTL pour chaque périphérique de la même classe
- communication par chaînes ASCII
- contient à la fois les canaux de données (ex-read/write) et de configuration (ex-ioctl)

1. cependant, chaque ioctl etait unique à chaque périphérique, donc 2 DACs devaient être reprogrammés selon les ioctl spécifiques

Plateforme/pilote

- des pilotes sont chargés en mémoire et sont associés à certains périphériques (VID :PID en USB, nom en SPI)
- les pilotes sont appelés par une plateforme, potentiellement plusieurs fois, avec la configuration matérielle acquise lors du boot ou de l'insertion du périphérique,
- $\Rightarrow$ appels **évènementiels** aux pilotes
- `platform` est le périphérique générique : `/sys/devices/platform`

Plateforme

- créée en fonction de la description du périphérique : pour une plateforme générique (/sys/bus/platform/devices/gpio-simple*)

```
static struct platform_driver gpio_simple_driver = {
      .probe = gpio_simple_probe,
      .remove = gpio_simple_remove,
      .driver = { .name = "gpio-simple",
      },
};
```

alors que pour une interface SPI

```
static struct spi_driver ad5624r_driver = {
        .probe = ad5624r_probe,
        .remove = ad5624r_remove,
        .driver = { .name = "ad5624r",
                    .owner = THIS_MODULE,
                  },
        .id_table = ad5624r_id,
};
```

$\Rightarrow$ définition des fonctions appelées lors du chargement (probe) et déchargement (remove)

Plateforme

Enregistrement de la plateforme auprès du noyau :

```
static struct platform_device *pdev;

static int __init gpio_simple_init(void)
{ret = platform_driver_register(&gpio_simple_driver);
 pdev = platform_device_register_simple("gpio-simple", 0, NULL, 0);
}

static void __exit gpio_simple_exit(void)
{platform_device_unregister(pdev);
 platform_driver_unregister(&gpio_simple_driver);
}

module_init(gpio_simple_init)
module_exit(gpio_simple_exit)
```

fait appel aux méthodes probe (`insmod`) ou remove (`rmmod`)
⇒ création de `/sys/bus/platform/drivers/gpio-simple`

Communication avec le pilote

Exemple de définition des points d'entrée de chaque instance du pilote :

```
static DEVICE_ATTR(value1, 0440, gpio_simple_show1, NULL); // methode show
static DEVICE_ATTR(value2, 0440, gpio_simple_show2, NULL); // methode show
static DEVICE_ATTR(value3, 0220, NULL, gpio_simple_read1); // methode store
```

se traduit par

```
# ls  /sys/bus/platform/drivers/gpio-simple/gpio-simple.0/
driver  driver_override  modalias  power  subsystem  uevent  value1
value2  value3
```

dont les actions sont par exemple

```
int gpio_simple_show1(struct device *dev, struct device_attribute *attr, \
  char *buf)
{ return sprintf(buf, "Hello World 1\n");}
```

Utilisation intensive des macros :
`linux-headers.../include/linux/device.h` indique

```
#define DEVICE_ATTR(_name, _mode, _show, _store) \
 struct device_attribute dev_attr_##_name = __ATTR(_name, _mode, _show, _st
```

⇒ création de la variable `dev_attr_value` appelée par
`device_create_file(&pdev->dev, &dev_attr_value);` (init) et
`device_remove_file(&pdev->dev, &dev_attr_value);` (exit)

IIO

IIO : Industrual Input/Outputs. (2009)

Pourquoi un nouvel environnement de communication ?

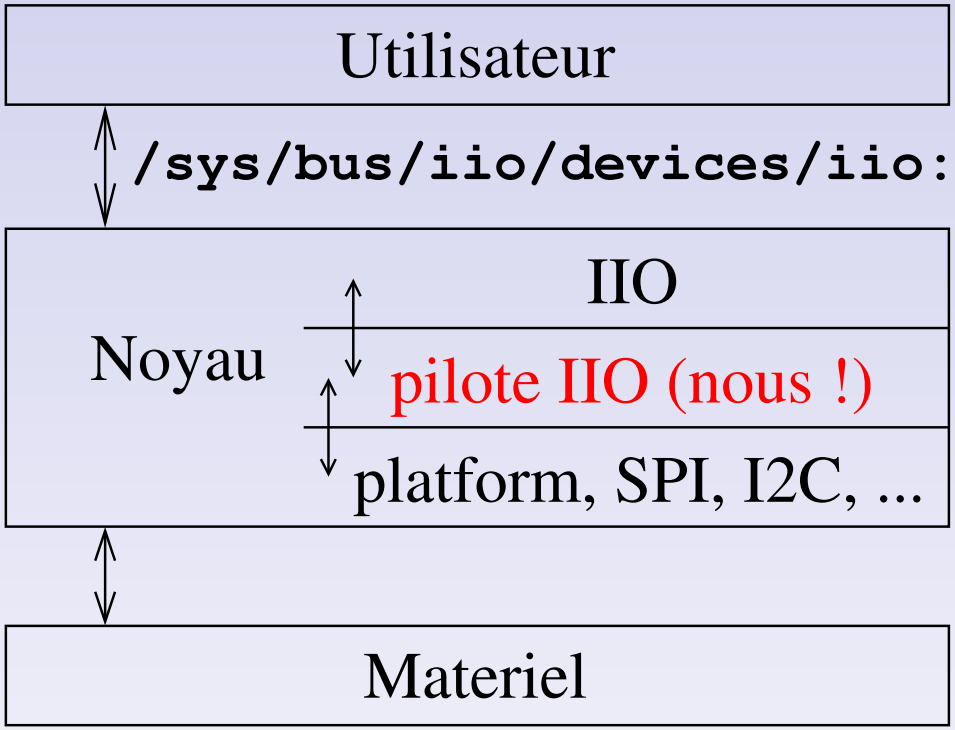
- IIO fournit les interfaces que tout capteur accessible par Linux doit respecter (ADC, DAC, accéléromètre, magnétomètre ...),
- échange d'informations en format ASCII au lieu de binaire, plus accessible au shell ou aux langages scriptés (Python)
- API entre l'interface vue de l'utilisateur et les pilotes qui accèdent au matériel

```
static int ad5624r_spi_write(struct spi_device *spi,
                             u8 cmd, u8 addr, u16 val, u8 len)
{
     u32 data;
     u8 msg[3];

     data = (0 << 22) | (cmd << 19) | (addr << 16) | (val << (16 - len));
     msg[0] = data >> 16;
     msg[1] = data >> 8;
     msg[2] = data;

     return spi_write(spi, msg, 3);
}
```

Définition des interfaces



`modprobe industrialio`

pour éviter

```
[   81.594608] iio_simple_dummy: Unknown symbol iio_device_alloc (err 0)
[   81.594621] iio_simple_dummy: Unknown symbol iio_device_free (err 0)
[   81.594634] iio_simple_dummy: Unknown symbol iio_device_unregister (err
[   81.594644] iio_simple_dummy: Unknown symbol iio_device_register (err 0)
```

Chargement du pilote ...

- En l'absence d'auto-détection, description de la structure matérielle de la carte par devicetree ou script.fex
- définition d'un pilote (qui n'est *pas* appelé automatiquement)

```
static const struct platform_device_id ad5624r_id[] = {
        {"ad5624r3", ID_AD5624R3},
        {"ad5644r3", ID_AD5644R3},
        {"ad5664r3", ID_AD5664R3}, // identifiant du peripherique
...

static struct spi_driver ad5624r_driver = {
        .driver = {
                    .name = "ad5624r",
                    .owner = THIS_MODULE,
                   },
        .probe = ad5624r_probe,
        .remove = ad5624r_remove,
        .id_table = ad5624r_id,    // ICI on dit quel composant
                                   // n\'ecessite ce pilote
};
```

... qui est appelé par la plateforme

- définition d'une plateforme qui fait appel au pilote au moment de sa première initialisation

```
static struct platform_device plat_gr_device = {
    .name = "ad5624r3",     // plateforme qui fait appel au pilote
// MEME NOM QUE DANS static struct platform_driver ad5624r_driver = {
```

- la méthode probe du pilote est appelée autant de fois que le périphérique est présent
- ainsi, le probe accepte des arguments que init ne peut pas recevoir.

```
insmod composant.ko
insmod dummy_platform.ko
ls -l /sys/bus/iio/devices/iio\:device0/
cat /sys/bus/iio/devices/iio\:device0/name
```

Initialisation du pilote

```
static const char *iio_dummy_part_number = "iio_dummy_part_no";
static int iio_dummy_probe(int index)
{  struct iio_dev *indio_dev;
   indio_dev = iio_device_alloc(sizeof(*st));
   iio_dummy_init_device(indio_dev);
   indio_dev->name = iio_dummy_part_number;
...
   indio_dev->channels = iio_dummy_channels;


...
   ret = iio_device_register(indio_dev);
}
```

et affichage des diverses étapes de chargement du pilote

```
[  156.678734] Probe
[  156.678765] Init device
```

qui se traduit par

```
# cat /sys/bus/iio/devices/iio\:device0/name
iio_dummy_part_no
```

Fichiers dans chaque interface

```
#define AD5624R_CHANNEL(_chan, _bits) { \
        .type = IIO_VOLTAGE, \
        .indexed = 1, \
        .output = 1, \
        .channel = (_chan), \
        .info_mask_separate = BIT(IIO_CHAN_INFO_RAW), \
        .info_mask_shared_by_type = BIT(IIO_CHAN_INFO_SCALE), \
...
```

Le pilote décrit les fonctionnalités du matériel (entrée/sortie, canaux, étallonage ...)

```
#define DECLARE_AD5624R_CHANNELS(_name, _bits) \
        const struct iio_chan_spec _name##_channels[] = { \
                AD5624R_CHANNEL(0, _bits), \
                AD5624R_CHANNEL(1, _bits), \
                AD5624R_CHANNEL(2, _bits), \
                AD5624R_CHANNEL(3, _bits), \
}

static DECLARE_AD5624R_CHANNELS(ad5624r, 12);
static DECLARE_AD5624R_CHANNELS(ad5644r, 14);
static DECLARE_AD5624R_CHANNELS(ad5664r, 16);
```

Fichiers dans chaque interface

```
#define AD5624R_CHANNEL(_chan, _bits) { \
        .type = IIO_VOLTAGE, \
        .indexed = 1, \
        .output = 1, \
        .channel = (_chan), \
        .info_mask_separate = BIT(IIO_CHAN_INFO_RAW), \
        .info_mask_shared_by_type = BIT(IIO_CHAN_INFO_SCALE), \
...
```

Le pilote décrit les fonctionnalités du matériel (entrée/sortie, canaux, étallonage ...)

```
#define DECLARE_AD5624R_CHANNELS(_name, _bits) \
        const struct iio_chan_spec _name##_channels[] = { \
                AD5624R_CHANNEL(0, _bits), \
                AD5624R_CHANNEL(1, _bits), \
                AD5624R_CHANNEL(2, _bits), \
                AD5624R_CHANNEL(3, _bits), \
}
```

et les points d'accès associés seront créés :

```
static const struct ad5624r_chip_info ad5624r_chip_info_tbl[] = {
 [ID_AD5624R3] = {.channels = ad5624r_channels,
                  .int_vref_mv = 1250, },
```

Définition des interfaces

Création de /sys/bus/iio/devices/iio\:device0 contenant

```
dev                          in_accel_x_raw          in_voltage0_raw
in_voltage3-voltage4_raw     out_voltage0_raw        uevent
in_accel_x_calibbias         in_sampling_frequency   in_voltage0_scale
in_voltage-voltage_scale     power
in_accel_x_calibscale        in_voltage0_offset      in_voltage1-voltage2_raw
name                         subsystem
```

Interfaces définies par la description du composant : iio_chan_spec [2]

```
static const struct iio_chan_spec iio_dummy_channels[] = {
{ .type = IIO_VOLTAGE,
  .indexed = 1,
  .channel = 0, // Channel has a numeric index of 0
  .info_mask_separate = // liste des infos fournies par le canal
  BIT(IIO_CHAN_INFO_RAW) | BIT(IIO_CHAN_INFO_OFFSET) | BIT(IIO_CHAN_INFO_SC
  .info_mask_shared_by_dir = BIT(IIO_CHAN_INFO_SAMP_FREQ), // sampling_freq
  .scan_index = voltage0,
  .scan_type = { // Description of storage in buffer
               .sign = 'u', /* unsigned */
               .realbits = 13, /* 13 bits */
               .storagebits = 16, /* 16 bits used for storage */
               .shift = 0, /* zero shift */
...
```

Liste des informations (RAW, OFFSET...) se retrouve dans write

2. $SOURCES/include/linux/iio/iio.h

Écriture

Fonction appelée lors de la lecture ou écriture des données :

```
static const struct iio_info composant_info = {
        .write_raw = composant_write_raw,
        .read_raw = composant_read_raw,
        .driver_module = THIS_MODULE,
};
```

qui fait appel aux fonctions de bas-niveau :

```
static int composant_write_raw(struct iio_dev *indio_dev,
   struct iio_chan_spec const *chan, int val, int val2, long mask)
{struct composant_state *state = iio_priv(indio_dev);
 int ret;
 switch (mask) {              // nature de la transaction
    case IIO_CHAN_INFO_RAW:  // vvv recupere nbre de bits dans chan
      if (val >= (1 << chan->scan_type.realbits) || val<0) return -EINVAL;
      ret = ad5624_spi_write(state, chan->channel, val);
    default: ret = -EINVAL;
   }
  return ret;
}
```

Unique fonction d'écriture + nature de la transaction sélectionnée par `info` [3] exploitant la fonction vue initialement (`ad5624r_spi_write()`)

3. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/drivers/iio/dac/ad8801.c?id=refs/tags/v4.9-rc8

Lecture

Remplit *val, *val2 et renvoie la nature de la transaction

```
static int composant_read_raw(struct iio_dev *indio_dev,
  struct iio_chan_spec const *chan, int *val, int *val2, long info)
{struct composant_state *state = iio_priv(indio_dev);
 switch (info) {
  case IIO_CHAN_INFO_RAW:
    *val = state->dac_cache[chan->channel];
    return IIO_VAL_INT;              // valeur entiere immediate renvoyee
  case IIO_CHAN_INFO_SCALE:
    *val = state->vrefh_mv - state->vrefl_mv;
    *val2 = 8;
    return IIO_VAL_FRACTIONAL_LOG2; // mantisse et exposant
  case IIO_CHAN_INFO_OFFSET:
    *val = state->vrefl_mv;
    return IIO_VAL_INT;
  default:
    return -EINVAL;
 }
 return -EINVAL;
}
```

Conclusion

- Perte de la généricité des appels par /dev ...
- ... au profit d'une uniformité des appels systèmes par périphérique/bus
- Nouveau niveau d'abstraction avec description du pilote et de la plateforme
- Interfaces séparant données et configuaration
- Échanges de données sous forme ASCII

Ressources :

- documentation de IIO dans les sources du noyau [4]
- M. Ripard, *IIO, a new kernel subsystem*, FOSDEM 2012 [5]
- L.-P. Clausen, *Using the Linux IIO framework for SDR – A hardware abstraction layer*, FOSDEM 2015 [6]

4. lxr.free-electrons.com/source/drivers/staging/iio/Documentation/
5. https://archive.fosdem.org/2012/schedule/event/693/127_iio-a-new-subsystem.pdf
et http://free-electrons.com/blog/fosdem2012-videos/ pour la vidéo
6. https://archive.fosdem.org/2015/schedule/event/iiosdr/

Mise en pratique

Consulter les exemples proposés dans

```
[...]/linux-headers-*/drivers/staging/iio$ ls iio_*
iio_dummy_evgen.c  iio_simple_dummy_buffer.c  iio_simple_dummy_events.c
iio_dummy_evgen.h  iio_simple_dummy.c         iio_simple_dummy.h
```

1. Proposer un exemple dans lequel deux instances d'un pilote sont chargées par une module noyau.
2. Proposer un exemple dans lequel un compteur est incrémenté à chaque lecture.
3. Proposer un exemple dans lequel la valeur renvoyée par une lecture est celle transmise au cours d'une écriture : penser à protéger lecture et écriture par un *mutex*.

Pour chaque problème, effectuer la démonstration **sur PC** puis sur carte **Olinuxino A13micro**.