Technical Brief 20110127 from Missing Link Electronics:

Hardware Parameter Access from Application Software

We present a technique for accessing hardware device paramenters and control values based on the Open Source GNU/Linux standardized concept of the `sysfs` virtual filesystem. Exemplary code for a motor control device demonstrates how to easily access a hardware device's registers from application software and/or user scripts to facilitate hardware integration or system debugging.

— Technical Brief 20110127 —

Interfacing between hardware and software plays an important role, for example, in rapid prototyping of control systems. Application software access to the real world by reading sensor values or by driving actuators is a basic requirement for embedded systems. By following defacto standards from the GNU/Linux Open Source world the Missing Link Electronics "Soft" Hardware Platform can quickly and safely link together the hardware and the software domains.

Hardware can have many control parameter values or status indicators which are not required during normal operation but may greatly facilitate debugging or realizing special applications like test and measurement equipment and data acquisition systems. However, to mitigate development risks a system's interfaces for hardware access should not be affected by this extra debug and diagnostic functionality such that existing application software can be left unchanged.

Communication between hardware and software is handled using so-called memory mapped hardware device registers, as shown in FIG. 1.

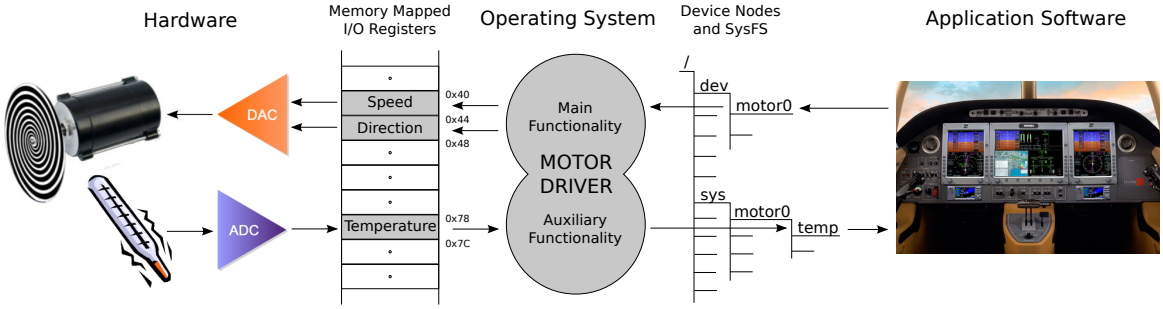

Figure 1: GNU/Linux Methods for Hardware Access

These hardware device registers are part of the hardware peripheral devices and blend into the memory map visible by the system's processor. When software accesses a memory region assigned to a particular hardware device, it thereby can exchange data with that particular hardware peripheral device. Under GNU/Linux, device drivers normally operate in a particular domain – the so-called kernel-space – while all user application software operates in the restricted domain – the so-called user-space.

Therefore, a "bridge" is needed to allow application software developers to easily and non-intrusively access certain hardware device registers. In short, a reliable method is needed for exporting hardware device register's from the device driver level in the kernel-space to the user-space (FIG. 2).

This technical brief will demonstrate how to implement such methods for linking hardware and software. It bases on programming normal file IO which is described in chapter 14 of the Linux Device Drivers programming guide [LDD3].

The MLE Linux follows Open Source GNU/Linux defacto standards where the kernel is in charge of providing device drivers, for access to the hardware and to hide hardware details from application software. Typically, the kernel provides unified methods of access to hardware peripheral devices mostly via so-called character device nodes or block device nodes. This way the kernel acts as a Hardware Abstraction Layer and user-space application software can be independent of the underlying hardware platform and independent of the exact hardware devices used.

While hiding hardware details in general is a good thing (and a very important feature of modern operating systems like GNU/Linux) there are situations where one wants more insight to the device driver and the hardware device's internals. This is the case during hardware / software co-debugging as well as for implementing application specific systems, used as test and measurement equipment, for example.

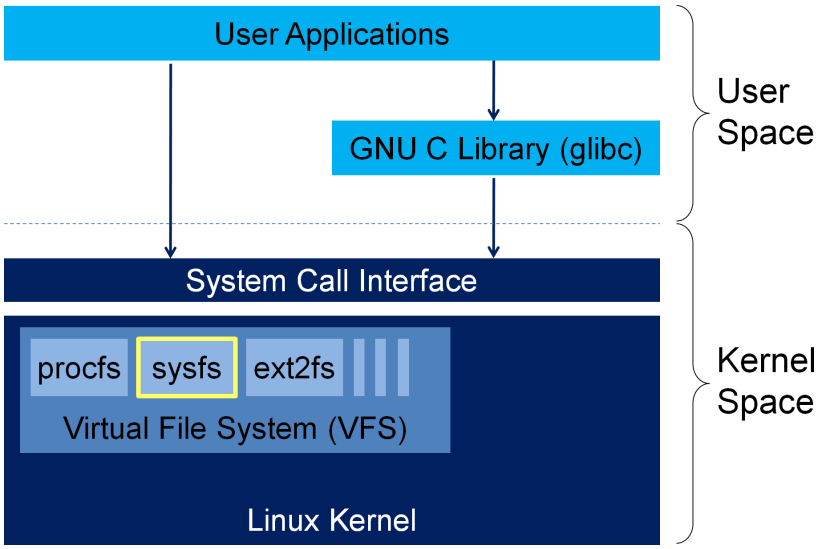

Figure 2: GNU/Linux User and Kernel Space

To implement this hardware / software access path, one should not change the standard interfaces which the kernel provides, because then the concept of a Hardware Abstraction Layer may brake, possibly resulting in major re-writes of existing application software. That's why at MLE we recommend to separate access to hardware parameter values and status indicators from the main device access methods.

The following will describe – in an exemplary fashion – the concepts for achieving this separation. As an example, we use the motor device with the corresponding Linux device driver from FIG. 1. This example's main functionality is to control the revolutions per minutes (RPM) of the motor and, therefore, the device driver provides a so-called Linux character device `/dev/motor0` to control RPM. In addition to the RPM control, our exemplary motor has a temperature sensor attached to it. While the device driver can read the current temperature values by accessing the provided memory mapped hardware registers, the question now is: How should this temperature value be passed to user-space applications?

Fundamentally, there exist five different ways to implement the extra access path:

1. Map the hardware device's hardware memory into user-space.
2. Insert an additional Linux character device for the temperature values.
3. Utilize the `ioctl` system call associated with the motor's existing Linux device.
4. Insert a node into `/proc` via the virtual filesystem `procfs`.
5. Insert a node into `/sys` via the virtual filesystem `sysfs`.

The first option for accessing hardware from user-space is straight forward: Map the IO memory of the respective hardware portion into user-space and access it like any other variable from within the application software. While this is a fast and very direct method of access, it is error prone and lacks any hardware abstraction. On top of this one would have to access hardware device registers via memory address offsets which is much more cumbersome and leads to harder to maintain application software code.

The second option, an additional Linux character device, may be a good choice depending on the type of data one wants to exchange with the device driver and/or the hardware. However, there is the challenge to identify corresponding device nodes. And, it may take a huge effort to parse the data, especially if many different bits of information are to be obtained. This access path is appropriate for large amounts of data with similar context, purpose and structure, but not for simple hardware parameter access.

The third option, the `ioctl` system call is intended to get and set parameters of a particular device, for example, small portions of data. As a result the `ioctl` system call is hard to use from command line or shell scripts and thus not ideal in rapid prototyping.

According to the GNU/Linux coding standard, the fourth option, access via the virtual filesystem `procfs` should only be used for access to process data (with minor exceptions for historical reasons). `procfs` is not intended for access to system details and, to stay compliant with defacto standards, one should never use it to expose device driver details to user-space.

The last option, access via the virtual filesystem `sysfs` is the preferred approach in the Open Source GNU/Linux community. It was invented to reflect the internal structure of a GNU/Linux system's devices and their relationship to each other. It was meant to export attributes for hardware peripheral devices to bridge the different layers: kernel-space and user-space.

Additionally, `sysfs` can reflect hierarchical structures like sub-system layers. In short, it is the perfect concept to expose hardware device parameters to user-space applications, because it was build for that sole purpose. In many cases, `sysfs` should be preferred over `ioctl` system calls because it is easier to use in command-line shell scripts and easier to maintain in application software.

The `sysfs` virtual filesystem's directory hierarchy is tied together by so-called `kobject` kernel object structures. These kernel objects can have parential relationships between each other and export so-called attribute file node for access from user-space. These attribute

file nodes can be accessed by standard read and write system calls just like any other file in the `sysfs` virtual filesystem. Typically, it is mounted under `/sys`.

Reading a file node from the `sysfs` virtual filesystem triggers the execution of a corresponding show-function inside the kernel, while writing to a file node herein executes a corresponding store-function. Each attribute file node represents a single hardware device parameter and – according to the defacto standards – shall contain one and only one single human readable value.

Using exemplary C source code fragments for our motor control hardware device with an integrated temperature sensor, we will demonstrate how to implement a `sysfs` file node. Thereby, our motor temperature can be accessed via the `sysfs` virtual filesystem. The following code snippets are part of a Linux kernel module and extend the device driver's functionality to provide the `sysfs` file nodes including access to the device's hardware registers.

For simplicity, we will create an attribute file node directly inside the directory `/sys` (normally, one would use sub-directories to reflect the hierarchy of the connected hardware peripherals). We also omit any comments and error checking to keep the example short. See [mle_ds_addac.c], the sourcecode for our Delta-Sigma analog to digital and digital to analog converters, for a more complete example.

Each `kobject` kernel object for an attribute file node has a `ktype` associated with it, which links it to call-back functions to be called when the kernel object is no longer needed, or to be called when the related `sysfs` entries are accessed via read or write from applications. The `ktype` structure also holds a list of attributes handled by the kernel object and which are exported via the `sysfs` virtual filesystem. As a result, a typical `sysfs` attribute file node has five components: C code structure as the handle for the data type, and C code functions for creation and initialization of the attribute file node, to be called as the kernel's store-function, to be called as the kernel's show-function, and to tear down and release the attribute file node.

```
static struct kobj_type motor_kobj_ktype = {
        .release = motor_kobj_release,
        .sysfs_ops = &(struct sysfs_ops){
                .show = motor_kobj_show,
                .store = motor_kobj_store
        },
        .default_attrs = (struct attribute*[]){
                &(struct attribute){
                        .name = "temperature",
                        .owner = THIS_MODULE,
                        .mode = S_IRUGO|S_IWUGO
                },
                NULL
        }
};
```

After allocating memory for the kernel object and initializing it to zero, a call to the C code function `kobject_init` will associate that kernel object with its' `ktype`. A call to the C code

function `kobject_add` will create a sub-directory within the `/sys` directory with the given name and will place file nodes for the kernel object's attributes within that sub-directory:

```
struct kobject *kobj = (struct kobject*)kzalloc(sizeof(struct kobject), GFP_KERNEL);
kobject_init(&kobj, &motor_kobj_ktype);
kobject_add(&kobj, NULL, "motor0");
```

As a result, there will be an entry `/sys/motor0` in the `sysfs` virtual filesystem.

The show- and store-functions are called when a read, respectively a write, to the attribute's file node occurs. When the attribute structure is passed to this functions, one can use the attribute's name as a convenient handle. Because this example has only one single attribute, "temperature", we do not check the name here. We get the value to write as a C code character string stored in `buffer`, convert it to an integer number and write this integer number into the predefined hardware device register. For a realworld application this register is mapped to the kernel's virtual address space. We assume this has already by done and `temperature_hw_register` is the registers virtual address. See [mle_ds_addac.c] for an example how to do this.

```
static ssize_t motor_kobj_store(struct kobject *kobj,
                                struct attribute *attr,
                                char *buffer,
                                size_t size)
{
        iowrite32be(simple_strtoul(buffer, NULL, 10), temperature_hw_register);
        return size;
}
```

Reading a hardware device register via `sysfs` is nearly the same, except the dataflow goes in the opposite direction:

```
static ssize_t motor_kobj_show(struct kobject *kobj,
                               struct attribute *attr,
                               char *buffer)
{
        snprintf(buffer, PAGE_SIZE, "%d_C\n", ioread32be(temperature_hw_register));
        return strlen(buffer)+1;
}
```

When the kernel object is no longer used, a call to the C code function `kobject_del` does the opposite of the C code function `kobject_add`: It removes all related entries from the sub-directory in the `sysfs` virtual filesystem. Afterwards, calling the C code function `kobject_put` will decrease the kernel's usage counter of the kernel object, eventually triggering a call of the release function, if the kernel object is no longer used by any other kernel code. This is how a release function can be implemented:

```
static void motor_kobj_release(struct kobject *kobj)
{
        kfree(kobj);
}

kobject_del(&kobj);
kobject_put(&kobj);
```

After loading the `sysfs` kernel module there will be a sub-directory `/sys/motor0` containing a file node named `temperature`.

We can test the proper functioning by using the UNIX `cat` command to read the file node:

```
$ cat /sys/motor0/temperature
43 C
```

The GNU/Linux `sysfs` virtual filesystem concept provides an easy-to-use interface for both kernel-space and user-space programs to exchange small amounts of data in the form of attributes, hardware device parameters or status information.

One should be aware of the fact, that `sysfs` is not real-time capable and any sequence of events may not be exactly synchronized to the data transfered on the device driver's main access path. It is, however, well suited to obtain information from peripheral hardware and the device driver and does not interfere with the standard device access methods or device operation.

Linking hardware and software via the `sysfs` virtual filesystem is therefore ideal for debugging purposes or to implement special application specific devices without breaking the Hardware Abstraction Layer or compatibility with standard software.

References

[LDD3] Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman:
Linux Device Drivers, Third Edition,
O'Reilly Media, February 2005.
http://lwn.net/Kernel/LDD3

[mle_ds_addac.c] *MLE driver for Delta-Sigma ADC and DAC*,
http://www.mlecorp.com/files/sources/mle_ds_addac.c