

Drivers: KFile

KFile interface: when object oriented programming (in C!) is useful in embedded

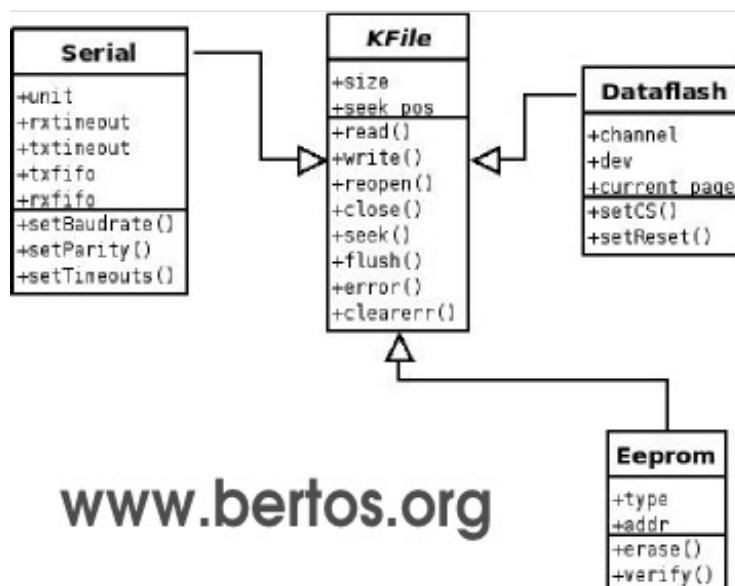
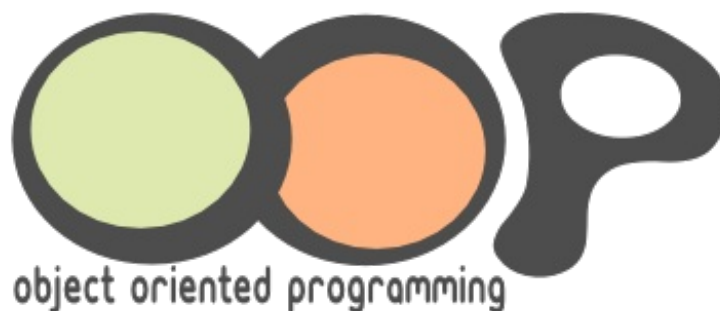
Do not be tricked by the headline, this is not otherworldly.

But let's start with order: **what problem do we want to solve?**

We have the typical problem of exchanging data with a peripheral: a RS232 serial, a SPI, an EEPROM, a flash and so on.

In our embedded operating system (<http://www.bertos.org>) we often create drivers that transfer data.

We usually code in C but very often we use the object oriented approach that is not only related to high level languages like C++, nor necessarily to a lack of resources. If used in the right way it elegantly solves a broad range of problems linked with the creation of common and reusable interfaces that allow you to save development time and precious memory space.



Back to the problem, the fastest and most efficient way seems to be implementing a range of specific functions that simply read/write from the device.

Taking the serial driver as an example, we could have:

```

ser_init - init serial
ser_read - read
  
```

```
ser_write - write
ser_close - close serial
```

Then, for instance, we could implement the SPI driver. Finally we will find:

```
spi_init - init SPI
spi_read - read
spi_write - write
spi_close - close SPI
```

HAVE YOU NOTICED ANYTHING STRANGE?

These functions semantically do the same things. OK, the implementation will be different, but generally speaking we will always need a function to initialize the device, one for reading, one for writing and, potentially, one for de-initializing the peripheral. If we were to develop a protocol that uses the serial, we would code a call to the various `ser_read`, `ser_write`, from inside the protocol.

And what if I have to use the same protocol with the SPI? You would say that renaming the calls would be enough and even fast, would you not? Sure, but in this way I would be forced to write **two** versions of the same protocol, one with the `ser_*` calls and one with `spi_*` ones.

In the event of using the same protocol with both drivers in the same application they would **double** the memory space required!

There is something wrong. This architecture forces to rewrite a customized version of all high-level code that uses low-level drivers, even if they are similar.

WOULD IT NOT BE MORE CONVENIENT TO ALWAYS HAVE THE SAME FUNCTIONS FOR THIS KIND OF STUFF?

After all, what we want to do is just read and write, why do we have to worry about **where** and **how**?

If for example I implement a bootloader, why does the code need to know that I am reading the firmware from a serial? The bootloader code is generic: it reads the new firmware somewhere, computes a checksum or similar and then reprograms the microcontroller flash. It would be reusable in other situations as well (it could read the firmware by radio, by ethernet, by a buffer memory, etc...), but in this way it is bound to a specific driver. I can not reuse it without changing it and making another version.

We should find a way to abstract from the driver we are using, and use a generic interface instead. It would be very handy to have a number of generic access functions to be used when we need to interface with a driver. We would always use them, the application code would always remain generic and would be reused in many situations, saving development time and memory space.

Those who are familiar with Object Oriented Programming (OOP) principles, will figure out what I am talking about: to have an abstract base class that describes an interface. For those not familiar with OOP, this "generic interface", is nothing but a series of "function prototypes" that describe how to perform some tasks. These functions are however only abstract and are not implemented. Or at least they are not implemented in one way only.

Each driver could implement these functions described in a generic way to really perform what they need. In OOP it is said that the driver could "derive" from this base "class" and implement the various methods.

BUT HOW IS THIS DONE IN C? THERE ARE NO CLASSES!

But structures do exist, and they are very similar. They can contain data, and also pointers. And in particular, function pointers. So if we define types for generic function pointers suited to our purposes, we can bundle them together in a struct ... and our interface is finally born!

```

typedef struct KFile
{
    OpenFunc_t open;

    ReadFunc_t read;

    WriteFunc_t write;

    CloseFunc_t close;
} KFile;

```

We named it **KFile** because it, in fact, describes a file, in the generic meaning of "information container". Actually the definition we have in our RTOS is more complete (you can view it here: <http://dev.bertos.org/browser/trunk/bertos/kern/kfile.h?rev=1798>) but for the sake of comprehension even this version is good enough.

BUT TAKE ONE STEP BACK, HOW ARE THESE FUNCTION POINTERS DEFINED?

For instance the type for the reading function, `ReadFunc_t`, might be this:

```

typedef size_t (*ReadFunc_t) (struct KFile *fd, void *buf, size_t size);

```

This function will return the number of bytes read.

Let's analyse the input parameters:

- ▶ **buf** is the buffer where read data will be put.
- ▶ **size** is the length we want to read.
- ▶ **fd** is a pointer to the current `KFile` structure needed as context.

OK, this interface is nice, but what is it for? How do I use it? And what is the purpose of `fd` parameter?

For using this interface we define a number of (small) handy inline functions which will be those we will really call on in the code that uses the interface.

For the read interface, we might have:

```

inline size_t kfile_read(struct KFile *fd, void *buf, size_t size)
{
    return fd->read(fd, buf, size);
}

```

And so on for other interface members.

Why are they necessary? Could I not call `fd->read` directly?

Yes I could. But with this stub function we have a single point in the code where all read functions pass. This is very handy for inserting debug information, statistics and/or special checks. For instance, in our RTOS, all of these stubs contain a check, in which the member called was not `NULL`.

Does this added "indirection" (as they say in jargon), waste CPU cycles or flash space?

No it does not. Being inline and single line makes the compiler not really perform a call for `kfile_read`, and then for `fd->read`, but this one is placed directly into the code without any waste of resources.

At this point we have defined a set of abstract interface functions and we know how to call them in the code that will use those.

Yes, but how can I write a driver that implements this interface?

We start, for example, with a serial driver. Usually it requires several things in order to work. For example we need to know which serial we want to open (often they are more than one). Without taking that into account, it will be virtually always necessary to have a buffer for the received characters (usually under interrupt) and other things like that.

We could call this the serial "status". Where do we place it? Usually in static variables of a module so they can be accessed by all the functions that work on the serial. Then we will need to write these functions that will perform the actions we already know (read, write, etc...). If there is more than one serial, we will have a set of twin functions that differ only because they work on different serial ports: ser0_read, ser1_read, ser2_read, etc...

These functions are virtually identical, except they work on different serials' "status".

But, if instead of leaving this status as a global variable, we place it in a structure, we will already see some benefits:

```
typedef struct Serial
{

    /** Physical port number */
    unsigned int unit;

    /**
     * \name Transmit and receive FIFOs.
     *
     * Declared volatile because handled asynchronously by interrupts.
     *
     * \{
     */
    FIFOBuffer txfifo;
    FIFOBuffer rxfifo;
    /* \} */

    /** ...other members... */
} Serial;
```

Here, to simplify, the status is represented by the port id number of the serial I want to talk to, and by the FIFO rx/tx buffers.

Then we have to change the functions in such a way that they do not implicitly and directly access the status any more, except through an **explicit** passing of this structure. For instance the serial read function prototype might be:

```
size_t ser_read(Serial *ser, void *buf, size_t len);
```

The latter, instead of directly accessing the status and the hardware registers needed, may take them according to the "unit" parameter present in the Serial structure passed by pointer. In this way, I write the serial access functions **only once** saving development time, debugging time, and microcontroller flash memory space!

Perhaps you have not noticed yet, but most of the job is done, and implementing the KFile interface is now very easy. Yes, because if we add the function pointers which we want to call to the data of the serial status, the job will

be done. There is no need to define prototypes, we have already done this when we have defined the KFile structure. Furthermore we could use it directly as **first member** (this is important, later you will understand why):

```
typedef struct Serial
{
    /* This driver implements a KFile interface */
    KFile fd;

    /** Physical port number */
    unsigned int unit;

    /* ...other members... */
} Serial;
```

Then we can define an "init" function that sets all the parameters needed:

```
int ser_init(Serial *ser, unsigned int unit, ...)
{
    /* Set the port number we want to open */
    ser->unit = unit;

    /* Assign the KFile interface functions to call */
    ser->fd->read = ser_read;
    ser->fd->write = ser_write;
    ...
}
```

The functions assigned in the last two rows are the ones that **truly** perform the job, and their prototype will be the same as defined in the KFile interface (do you remember ReadFunc_t?). They may also be static in the serial module, since no one outside the C file will call them directly by name.

Now the work is finished, the serial driver can be used from any KFile function! How is this possible?

Do you remember that the KFile field inside the Serial structure should be the **first**?

If I pass a Serial structure to the kfile_read() function, since the first part of Serial is really a KFile structure, it will find what it is looking for in the memory.

kfile_read() will call fd->read that being assigned to ser_read, will perform really what you want!

Here you can understand the need of the **fd** parameter that each KFile function take as first argument: it is the serial driver status that low level functions like ser_read need to know what to work on.

Once drivers are written according to KFile interface, accessing them is totally abstract and generic.

An example code might be:

```
/* Declare the serial context */
Serial ser0;

/* Initialize serial 0 */
ser_init(&ser0, );

/* Read from serial 0 */
```

```
kfile_read(&ser0.fd, buf, len);
```

As you can see I chose to pass the KFile member pointer, which is inside the Serial status, to kfile_read, in order to avoid a cast. Alternatively it could be done with:

```
/* Read from serial 0 */  
kfile_read((KFile *)&ser0, buf, len);
```

These two methods are equivalent, only if the KFile member inside the driver is the first, although this way is a bit "dirtier".

OK, everything is beautiful, but does this waste resources?

Not necessarily. If you look well, for every driver that implements the KFile interface, we only have an overhead given by the KFile structure itself. It contains only few function pointers so the RAM used is in the order of magnitude of ten bytes per driver. Another potentially "negative" factor is that driver function calls are not performed directly, but indirectly, using function pointers: the CPU cycles only slightly more every time we call a function of hundreds/thousands CPU cycles.

OK, but for which drivers can I use this interface?

Many. If the interface you use is well designed and is sufficiently generic, there are no limits. In BeRTOS we use it for practically everything: accessing serial, SPI, EEPROM, Dataflash, etc...

Moreover, when the driver needs special operations, it is always possible to define specific functions that operate on specific drivers only: on the serial we will need a function to set the baudrate, for the EEPROM we need one that allows to move in the address space and specify at which address we want to read/write to, and so on.

OK, the most important question, what is all this for?

Here we are at the crucial point. With this approach, we will get fantastic benefits, with relatively few drawbacks:

If more possible instances of the same driver do exist:

4 identical serials, 8 identical memories, etc... **I write the driver only once** instead of 4 or 8 times. Although I could copy-and-paste, the bugs increase dramatically. When you paste, you have to change status and register references in every place you use them (you will always miss something, and it will be very difficult to spot). Furthermore, when you find a bug in a driver you have to remember to apply the (modified accordingly) fix to the driver copies. But above all you will waste 4 or 8 times the memory space needed!

Many different drivers can be controlled by the same functions!

There will be no need to rewrite a specific printf for every driver you have, **only one** which writes on a KFile is enough. Once the interface is defined you can call your printf on serials, SPI, EEPROM, displays without adding a line of code, saving a great deal of time and space. Implementing in the driver the few low level functions defined in the KFile interface, grants you access to all generic KFile functions (see here: <http://dev.bertos.org/browser/trunk/bertos/kern/kfile.c?rev=1886>) and all high-level protocols and applications that you you will write (or that you will find already written in BeRTOS).

You can switch between drivers on the fly without modifying the application code.

If you have an EEPROM and you have to switch to a flash, KFile interface ensures that your application, which is used to write on an EEPROM, will magically write on a flash with no modifications. If you write protocols, you can implement them in an abstract way on a KFile and choose the physical layer on the fly: using serial, SPI or even Ethernet makes no difference when there is a KFile!

Not only can you switch drivers on the fly, but you can also do it at runtime!

If we then prepare a configuration protocol (or a hardware mechanism) to know the board revision number, it is possible to use the very same binary firmware file for all board revisions, even if the hardware changes between board revisions (therefore infinitely simplifying production procedures, firmware upgrades, avoiding confusion for wrong versions, etc...). On start-up the firmware can just configure itself (via a protocol or "discovering" on which board it is running) connecting, according to configuration, a hardware peripheral, or another with the KFile interface.

DEVELOPER'S TRAC

On the developers website you will find all the information to contribute to BeRTOS development community, milestones, the possibility to open tickets to the community, the changelog and all the details on every feature. dev.bertos.org

PREMIUM SUPPORT

By registering to Develer's advanced support services you will have online assistance, phone assistance and the possibility to report bugs directly to our BeRTOS engineering team. All services guarantee reply in 24 working hours. bertos.org/supporto/premium/

CERTIFIED TRAINING

Customized training, onsite teaching, embedded application debug and deploy assistance, peer review on source code, training classes for single programmers or working teams. bertos.org/supporto/formazione/



BeRTOS website by [Develer](#).