First KOS module and coding conventions

Thomas Petazzoni*

15 Janvier 2005

Abstract

This documents explains how to create a simple, *Hello World* type, module in KOS, and give some recommandations regarding coding conventions.

Contents

1 Location **1**

2 Initialisation and cleanup routines **1**

3 Compilation **3**

4 Module features, exporting symbols **3**

5 Various coding conventions **6**

5.1 File naming 6

5.2 Function naming 6

1 Location

All modules in KOS are stored in the `modules/` directory, or in one of its subdirectories. The most important subdirectories are `x86/`, the directory that contains x86-specific code, `lib/`, the directory that contains various libraries and `fs/`, the directory that contains filesystems implementation.

So, to create the *Hello World* module, simply create a `helloworld/` subdirectory inside `modules/`.

2 Initialisation and cleanup routines

Each module may have one or more initialisation or cleanup routines. Currently, only initialisation routines are useful, since KOS is not able to unload modules.

*thomas.petazzoni@enix.org

Initialisation and cleanup routines are usually written in a C file that has the name of the module, in our case, it will be `helloworld.c`. This file should contain nothing except initialisation and cleanup routines, and symbol export directives (see **??**).

There are two kinds of initialisation routines:

- `init` routines, that are called before the interrupts are enabled in the operating system. There should be used only for the low-level modules like memory or task management;
- `post init` routines, called once the interrupts are enabled and the system is running threads;

For each type of initialisation routines, there are different levels that allows to order the initialisation of the various modules. All initialisation routines of level 0 are called before initialisation routines of level 1, etc. There are both initialisation levels for `init` and `post init` initialisation routines. Inside an initialisation level, the call order is defined by the order of the modules in the `MkVars` file (see 3).

As our *Hello World* module doesn't contain any low-level feature, we'll only use a `post init` initialisation routine, that we'll put at level 0:

Listing 1: Initialisation routine

```
__init_text static int post_init_module_level0
                          (kernel_parameter_t *kp)
{
   UNUSED(kp);

   printk("(Init␣module␣helloworld...␣Ok)\n");
   return 0;
}
```

The `post init` routines are usually named `post_init_module_level` appended with the level at which the routine will be executed. The `kernel_paramater_t` argument is passed to all modules, it contains various information concerning the configuration set up by the *loader*. These information are only useful for some modules.

The `__init_text` statement allows the kernel to frees the memory used by the initialisation code once initialisation is done. All global variables or functions used only during initialisation should use this statement. The `static` statement ensures that the function is not used anywhere outside of the current file.

The `UNUSED` macro is a simple macro that allows to avoid warnings about unused arguments. We activated a lot of warnings, including the ones about unused arguments. If you really know that you don't want to use an argument, mark it with `UNUSED`. `UNUSED` must be placed after all local variables declaration.

Now, we have to register this function as an initialisation routine, using the following statement:

Listing 2: Initialisation routine declaration

```
DECLARE_INIT_SYMBOL(post_init_module_level0, POST_INIT_LEVEL0);
```

That's enough to define an initialisation routine. We should also include `loader/mod.h` that contains the definitions for the macros used to declare initialisation routines, `lib/std/libstd.h` that contains the definition of the `printk` function and `kos/macros.h` that contains the definition of the `UNUSED` macro.

Listing 3: `helloworld.c` source code

```
#include <loader/mod.h>
#include <kos/macros.h>
#include <lib/std/stdlib.h>

__init_text static int post_init_module_level0
                           (kernel_parameter_t *kp)
{
   UNUSED(kp);

   printk("(Init␣module␣helloworld...␣Ok)\n");
   return 0;
}

DECLARE_INIT_SYMBOL(post_init_module_level0, POST_INIT_LEVEL0);
```

3 Compilation

In order to compile our module, we need to set up a simple `Makefile`, in the module directory:

Listing 4: `Makefile` for the *Hello World* module

```
OBJS=  helloworld.o

all: helloworld.ro

helloworld.ro: $(OBJS)

TOPSRCDIR=../..
include $(TOPSRCDIR)/modules/MkRules
```

The `OBJS` variables must contain the name of all object files that must be compiled in the module. `helloworld.ro` is the result of the compilation: it is the module itself, usable by the loader. Using that `Makefile`, you can now compile your module by simply typing `make`.

However, if you recompile KOS completely, your module won't be compiled. To do this, simply add `helloworld` inside the `SUBDIRS` variable at the top of the `modules/Makefile` file.

Now that your module is compiled, it has to be added to the floppy image of the operating system with the other modules. To do this, you have to edit the `MkVars` file at the main directory of KOS sources, and add the following line to the `MODULES` variable :

```
$(MODULES_DIR)/helloworld/helloworld.ro
```

KOS is ready to run your new module, using the instructions given in the *Compile, test and debug* KOS documentation. You should see the string printed through `printk` after the KOSlogo. The figure 1 shows what you should see. The red color of the message has been added afterwards, you should see it with the normal grey color.

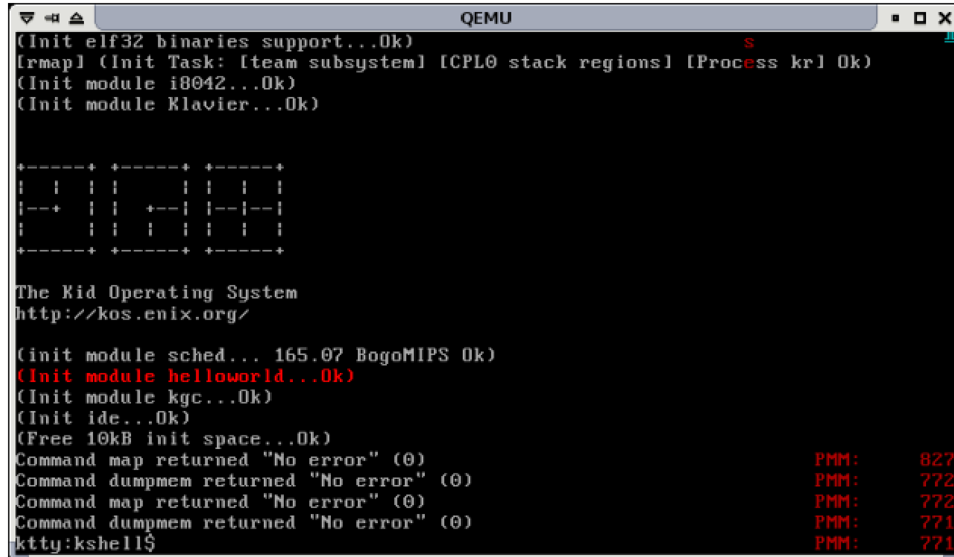

Figure 1: *Hello World* module running

4 Module features, exporting symbols

Your *Hello World* module is very simple for the moment. Let's add some features to it. The first one is an internal function of the module, which can be called only inside the module. This function will be called `hello_inside`. The second one is a function which is going to be exported, which means that other modules will be able to call it.

All module features should be implemented in C source files whose names start with an underscore. In our example, the code will be stored inside the `_helloworld.c` file.

Listing 5: The code of the *Hello World* module

```c
#include <lib/std/stdlib.h>

void hello_inside(void)
{
  printk(``Hello from the inside'');
}

void hello_outside(void)
{
  printk(``Hello from the outside'');
}
```

Now, we have to declare the prototypes of these functions in header files. The prototypes of exported functions and all public information (type definitions, macro definitions) must be placed in a header that has the name of the module, in our case `helloworld.h`. Other modules are allowed to include such a file.

Listing 6: Public include file `helloworld.h`

```
#ifndef __HELLOWORLD_H__
#define __HELLOWORLD_H__

void hello_outside();

#endif /* __HELLOWORLD_H__ */
```

The private function prototypes, type and macro definitions should be placed in a header file called `_helloworld.h`. It should not be included by other modules, but has to be included by the files of the `helloworld` module. It should contains a directive to include the public file `helloworld.h`.

Listing 7: Private include file `_helloworld.h`

```
#ifndef ___HELLOWORLD_H__
#define ___HELLOWORLD_H__

#include ``helloworld.h''

void hello_inside();

#endif /* ___HELLOWORLD_H__ */
```

Now, we can modify the `helloworld.c` file to use the `hello_inside` function and to export the `hello_outside` function:

Listing 8: `helloworld.c` source code

```
#include <loader/mod.h>
#include <kos/macros.h>
#include <lib/std/stdlib.h>
#include ``_helloworld.h''

__init_text static int post_init_module_level0
                            (kernel_parameter_t *kp)
{
   UNUSED(kp);

   printk("(Init␣module␣helloworld...␣Ok)\n");
   hello_inside();
   return 0;
}

EXPORT_FUNCTION(hello_outside);
```

```
DECLARE_INIT_SYMBOL(post_init_module_level0, POST_INIT_LEVEL0);
```

Since our module has several C source files, we need to modify the Makefile accordingly. The OBJS variable of the Makefile must be modified to:

Listing 9: OBJS variable of the Makefile

```
OBJS= helloworld.o _helloworld.o
```

5 Various coding conventions

5.1 File naming

- In each module, the file module.c should only contain initialisation and cleanup routines. It must contains *all* export symbol declarations ;
- In each module, the file module.h should only contain public function prototypes, type and macro definitions. By *public*, we mean *available to other modules*. For example, all exported functions must have their prototype defined in module.h as well as the types they use ;
- In each module _module.h should contain private function prototypes, type and macro definitions. This file should be included by all source files of the module, but not by any other module. It must include the public module.h so that all the source files of the module only have to include the private header ;
- The core of the module should be implemented in files whose names start with an underscore ;

5.2 Function naming

- All functions are named only with small letters and underscores. No capital letters are used ;
- All exported functions should not start with an underscore, and should start with a prefix common to the module, for example physmem_get_page or physmem_put_page ;
- All internal functions should start with an underscore ;
- All internal functions used only in the current file should be declared as static ;
- Except some specific functions, all functions should return a result_t. They should use return ESUCCESS to inform the caller that it was successfull or return -EMYERROR to inform the caller that something failed. All errors are defined in modules/kos/errno.h;