Adding a driver may be quite easy if the driver class already exists. If not, create a new driver class and come back here.

We'll go through this guide assuming we are adding a new character device, we'll call it `pmtt`.

This device is an imaginary one with:

- a transmit register `TXR`,
- a receive register `RXR`,
- a status register `STR`, telling whether we can receive,
- an interrupt mask register `ITR`, masking send / receive interrupts.

Creating a new directory

We'll go to the correct directory in the source tree and create the directory for the device:

```
$ cd mutekh/device/driver/char
$ mkdir pmtt
$ cd pmtt
```

Creating the files

Now we can create the files. There are usually 4 files for a device:

- a `.config` for the BuildSystem,
- a `.h` for declaring the public functions,
- a `.c` implementing the actual driver,
- a `-private.h` containing the private definitions for the driver.

The `.config` file

Let's start with the `.config` file. It will contain:

```
%config CONFIG_DRIVER_CHAR_PMTT
desc Enable PMTT imaginary device
default undefined
provide CONFIG_DRIVER_TTY
%config end
```

Here we declare this driver implements a TTY, i.e. a device with a send/receive interface.

The `.h` file

Now we may define the external interface of our device.

Interface function prototypes are defined in two headers:

- `hexo/device.h` defines the class-agnostic functions: init, cleanup and irq,
- `device/`*class*`.h` defines the class-specific functions. for character devices, there is only a `request` function to implement.

In order to enforce the correct prototype, even in case of evolution, it is mandatory to use the proper macros `DEV_INIT`, `DEV_CLEANUP`, `DEV_IRQ` and `DEVCHAR_REQUEST`.

```
#ifndef DRIVER_TTY_PMTT_H
#define DRIVER_TTY_PMTT_H

#include <device/char.h>
#include <hexo/device.h>

/* The device constructor & destructor */
DEV_INIT(pmtt_init);
DEV_CLEANUP(pmtt_cleanup);

/* The request handler */
DEVCHAR_REQUEST(pmtt_request);

/* The IRQ handler */
DEV_IRQ(pmtt_irq);

#endif
```

The private header

Here we can define the internal constants, like the header offsets and the register meanings.

All the internally-used structures may also be defined here. Driver instances may have an internal state, we'll define a `struct pmtt_state_s` containing the current pending request.

```
#ifndef DRIVER_TTY_PMTT_PRIVATE_H
#define DRIVER_TTY_PMTT_PRIVATE_H

/* Register offsets */
#define PMTT_TXR 0
#define PMTT_RXR 1
#define PMTT_STR 2
#define PMTT_ITR 3

/* Status register bits */
#define PMTT_STR_RXREADY 1
#define PMTT_STR_TXREADY 2

struct pmtt_state_s
{
   struct dev_char_rq_s *current_request;
};

#endif
```

The implementation file

The drivers should be non-blocking and return as soon as possible. A callback (defined in the request structure) has to be called, either from the IRQ handler or the request function, telling the upper layer the about status of the request. Moreover, this driver relies on the ICU driver class in order to receive interrupts.

First we have common header and declarations:

```
#include "pmtt.h"
#include "pmtt-private.h"

#include <device/icu.h>
```

```
#include <hexo/types.h>
#include <hexo/device.h>
#include <device/driver.h>
#include <hexo/iospace.h>
#include <hexo/alloc.h>
#include <hexo/interrupt.h>
```

Let's begin with the device constructor. Here we have to:

- Fill the device structure with a static and read-only pointer-to-function table,
- Allocate internal state for the driver and put it in the `drv_pv` field of the device,
- Register to the interrupt controller unit,
- Initialize the peritheral.

The function table is:

```
static const struct driver_s pmtt_drv =
{
    .class = device_class_char,
    .f_init = pmtt_init,
    .f_cleanup = pmtt_cleanup,
    .f_irq = pmtt_irq,
    .f.chr = {
        .f_request = pmtt_request,
    }
};
```

When a device is created, the calling code must fill the device structure `icudev` and `irq` fields. The driver is responsible for registering to the ICU.

```
DEV_INIT(pmtt_init)
{
   struct pmtt_state_s *state = mem_alloc(sizeof(struct pmtt_state_s), MEM_SCOPE_SYS);

   if ( state == NULL )
     return ENOMEM;

   state->current_request = NULL;

   dev->drv = &pmtt_drv;

   dev->drv_pv = state;

   DEV_ICU_BIND(dev->icudev, dev, dev->irq, pmtt_irq);

   return 0;
}
```

The destructor is quite straightforward:

```
DEV_CLEANUP(pmtt_cleanup)
{
   struct pmtt_context_s *pv = dev->drv_pv;

   DEV_ICU_UNBIND(dev->icudev, dev, dev->irq);

   mem_free(pv);
}
```

For character devices, the upper layer may decide to continue with the request or not, thus the callback has to be called any time the status changes. See trunk/mutekh/drivers/include/device/char.h for complete reference.

For the sake of simplicity, this driver won't be able to handle concurrent requests. Therefore we'll return directly from handler if there is already a pending request.

```
DEVCHAR_REQUEST(pmtt_request)
{
   struct pmtt_context_s *pv = dev->drv_pv;

   if (rq->size == 0) {
      rq->error = 0;
      rq->callback(dev, rq, 0);
   }

   if (pv->current_request != NULL) {
      rq->error = EBUSY;
      rq->callback(dev, rq, 0);
   }

   pv->current_request = rq;

   switch (rq->type)
   {
   case DEV_CHAR_READ:
      // Enable RX irq
      cpu_mem_write_32(dev->addr[0] + PMTT_ITR, PMTT_RXREADY);
      break;

   case DEV_CHAR_WRITE:
      // Enable TX irq
      cpu_mem_write_32(dev->addr[0] + PMTT_ITR, PMTT_TXREADY);
      break;
    }
}
```

Then we can define the IRQ handler, where all the device logic takes place:

```
DEV_IRQ(pmtt_irq)
{
   struct pmtt_context_s *pv = dev->drv_pv;

   struct dev_char_rq_s *rq = pv->current_request;

   size_t len = 0;

   switch (rq->type)
   {
   case DEV_CHAR_READ:
      while ( rq->size && (cpu_mem_read_32(dev->addr[0] + PMTT_STR) & PMTT_RXREADY) ) {
         *(rq->data) = cpu_mem_read_32(dev->addr[0] + PMTT_RXR);
         rq->size--;
         rq->data++;
         len++;
      }
      break;

   case DEV_CHAR_WRITE:
      while ( rq->size && (cpu_mem_read_32(dev->addr[0] + PMTT_STR) & PMTT_TXREADY) ) {
         cpu_mem_read_32(dev->addr[0] + PMTT_TXR, *(rq->data));
         rq->size--;
         rq->data++;
         len++;
      }
      break;
   }
```

```
      // At last, we can consider the request as finished if the callback says so,
      // or if the size is 0.
      if ( rq->callback(dev, rq, size) || rq->size == 0 ) {
         pv->current_request = NULL;
         cpu_mem_write_32(dev->addr[0] + PMTT_ITR, 0);
      }

      // Tell the ICU we processed the IRQ.
      return 1;
   }
```

Possible improvements

As an exercise to the reader, this driver could be improved to:

- buffer the incoming characters even if no request is pending,
- handle concurrent requests,
- correctly lock the device to avoid race-conditions when multiple processors access the same device.