```
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("MY I3C Master IP");

struct class *my_i3c_class = NULL;
struct device *my_i3c_device = NULL;
struct my_i3c_device_data my_device_data;

// process xfer frame
void my_i3c_master_xfer_start_locked(struct my_i3c_master *master) {
   struct my_i3c_xfer *xfer = master->xferqueue.cur;
   my_i3c_master_set_interrupt_mask(master, MSTATUS_COMPLETE | MSTATUS_MCTRLDONE);

   /* Process commands list stored in xfer (private transfert) */
   writel(cmd->cmd_ctrl, master->regs + MCTRL_REG);
   for (...) {
         writel(payload[idx], master->regs + MWDATAB_REG);
         my_i3c_master_trigger_cmd(master, MSTATUS_TXNOTFULL)) // wait if Tx is full
      }
   }
}

void my_i3c_master_xfer_end_locked(struct my_i3c_master *master, u32 isr) {
   // enable interruption for command completion or error/warning */
   my_i3c_master_set_interrupt_mask(master, ~MSTATUS_COMPLETE & ~MSTATUS_MCTRLDONE);
   xfer->status = readl(master->regs + MSTATUS_REG);
   if (xfer->cmds[0].rnw && (isr & MSTATUS_RXPEND)) {
      for (status = ...; idx...);
          status = readl(master->regs + MSTATUS_REG), idx++)
         payload[idx] = readl(master->regs + MRDATAB_REG);
      }
   }
   complete(&xfer->comp);

   xfer = list_first_entry_or_null(&master->xferqueue.list, struct my_i3c_xfer, node);
   master->xferqueue.cur = xfer;
   my_i3c_master_xfer_start_locked(master);
}

void my_i3c_master_xfer_queue(struct my_i3c_master *master, struct my_i3c_xfer *xfer) {
   init_completion(&xfer->comp);
   spin_lock_irqsave(&master->xferqueue.lock, flags);
   if (master->xferqueue.cur) {
      list_add_tail(&xfer->node, &master->xferqueue.list);
   } else {
      master->xferqueue.cur = xfer;
      my_i3c_master_xfer_start_locked(master);
   }
   spin_unlock_irqrestore(&master->xferqueue.lock, flags);
}

void my_i3c_master_xfer_unqueue(struct my_i3c_master *master,struct my_i3c_xfer *xfer)
{
   unsigned long flags;

   spin_lock_irqsave(&master->xferqueue.lock, flags);
   if (master->xferqueue.cur == xfer) {
      status = readl(master->regs + MSTATUS_REG);
      my_i3c_master_update_bus_status(master, status);
      my_i3c_master_write_exit/stop(master);
   } else {
      list_del_init(&xfer->node);
   }
   spin_unlock_irqrestore(&master->xferqueue.lock, flags);
}
```

```c
int my_i3c_master_bus_init(struct i3c_master_control *m) {
   struct my_i3c_master *master = to_my_i3c_master(m);
   struct i3c_device_info info = {};
   /* configure fgpa */
   my_i3c_master_set_config(master);
   status = readl(master->regs + MSTATUS_REG);
   my_i3c_master_update_bus_status(master, status);
   /* Get an address for the master. */
   ret = i3c_master_get_free_addr(m, 0);
   writel((ret & GENMASK(6, 0)) << 1, master->regs + MDYNADDR_REG);
   ret = i3c_master_set_info(&master->base, &info);
   return 0;
}

int my_i3c_master_ccc_cmd_get(struct my_i3c_master *master, struct i3c_ccc_cmd *cmd) {
   struct my_i3c_xfer *xfer = my_i3c_master_xfer_alloc(master, 1);
   struct my_i3c_cmd *ccmd;
   ccmd = xfer->cmds; ccmd->rx_buf = cmd->dests[0].payload.data; ccmd->cmd_ctrl = ...;
   my_i3c_master_xfer_queue(master, xfer);
   wait_for_completion(&xfer->comp))
   my_i3c_master_xfer_free(xfer);
   return ret;
}

int my_i3c_master_send_ccc_cmd(struct i3c_master_control *m, struct i3c_ccc_cmd *cmd) {
   struct my_i3c_master *master = to_my_i3c_master(m); ...
   my_i3c_master_ccc_cmd_get(master, cmd); ...
}

int my_i3c_master_priv_xfers(struct i3c_dev_desc *dev, struct i3c_priv_xfer *xfers, n)
{
   struct i3c_master_control *m = i3c_dev_get_master(dev);
   struct my_i3c_master *master = to_my_i3c_master(m);
   struct my_i3c_cmd *ccmd = NULL;
   struct my_i3c_xfer *current_xfer;
   unsigned i, ret = 0;

   for (i = 0; i < n; i++) {
      current_xfer = my_i3c_master_xfer_alloc(master, 1);
      ccmd = &current_xfer->cmds[0];
      ccmd->rnw = xfers[i].rnw;
      ccmd->cmd_ctrl = MCTRL_SEOM_MASK...;
      if (ccmd->rnw) { ccmd->rx_buf = xfers[i].data.in;  }
      else        { ccmd->tx_buf = xfers[i].data.out; }
      my_i3c_master_xfer_queue(master, current_xfer);
      wait_for_completion(&current_xfer->comp)
      xfers[i].err = my_i3c_master_get_err(&current_xfer->cmds[0]);
      my_i3c_master_xfer_free(current_xfer);
   }
   return current_xfer->ret;
}


int my_i3c_master_do_daa(struct i3c_master_control *m) {
   struct my_i3c_master *master = to_my_i3c_master(m);
   ...
   xfer = my_i3c_master_xfer_alloc(master, 1);
   my_i3c_master_xfer_queue(master, xfer); // exec DAA request and read the 8 bytes
   wait_for_completion(&xfer->comp);
   i3c_master_get_free_addr(m, last_addr + 1); // get free address fpr slave
   writel(last_addr, master->regs + MWDATAB_REG); // Write the new DA for the slave
   ...
   ret = readl(master->regs + MDATACTRL_REG);
   if (ret & DATACTRL_RXEMPTY) my_i3c_master_xfer_free(xfer);
   for (i = 0; i < slot; i++) i3c_master_add_i3c_dev_locked(m, addrs[i]);
   i3c_master_defslvs_locked(&master->base);
   return 0;
}

void my_i3c_master_ibi_hj(struct work_struct *work) {
   // Receive HotJoin
   switch (master->hj.mode) {
   case my_i3c_HOTJOIN_ACK:
      writel(MCTRL_IBIRESP_ACKNOBYTE, master->regs + MCTRL_REG);break;
   case my_i3c_HOTJOIN_MANUAL:
      writel(my_i3c_HOTJOIN_ACK);break;
   }
   my_i3c_master_write_stop(master);
   send_sig_info(SIGIBI, &info, my_device_data.task); // send signal to user
   err = i3c_master_do_daa(&master->base);
}

int my_i3c_master_ibi_read(struct my_i3c_master *master, u8 addr)
{
   for(status = readl(master->regs + MCTRL_REG), idx=0; (status & ...); idx++){
      read_data[idx] = readl(master->reg + MRDATAB_REG);}

   memcpy(my_device_data.ibi_event->data, read_data, len);
   send_sig_info(SIGIBI, &info, my_device_data.task); // send signal to user
}

/* Called after an IBIWON interruption to process the IBI
 * (Detect if it's a HotJoin Or Data IBI) */
int my_i3c_master_ibi_demux(struct my_i3c_master *master, unsigned status)
{
   err = my_i3c_master_trigger_cmd(master, MSTATUS_COMPLETE); // wait COMPLETE status
   switch (MSTATUS_IBITYPE(status)) {
   case MSTATUS_IBITYPE_IBI:
      err = my_i3c_master_ibi_read(master, master->ibi.addr); // read incoming data
      break;
   case MSTATUS_IBITYPE_HJ:
      queue_work(master->base.wq, &master->hj.hjwork); // pure hot plug event
      break;
   case MSTATUS_IBITYPE_NONE:
   default: break;
   }
   return err;
}

irqreturn_t my_i3c_master_interrupt(int irq, void *data)
{
   struct my_i3c_master *master = data;

   u32 interrupt_status = readl(master->regs + MINTMASKED_REG);
   status = readl(master->regs + MSTATUS_REG);
   /* Request by a slave to send a start & allow the reception of IBI */
   if (interrupt_status & MSTATUS_SLVSTART) {
      // send a Start to the bus to offer the possibility to the slave of process IBI
      queue_work(master->base.wq, &master->ibi.slvreq_work);
   } else if (interrupt_status & MSTATUS_IBIWON) {
      my_i3c_master_ibi_demux(master, status); // Rx from slave or hot plug
   } else {
      spin_lock(&master->xferqueue.lock);
      my_i3c_master_xfer_end_locked(master, interrupt_status); // xfer completion
      spin_unlock(&master->xferqueue.lock);
   }
   return IRQ_HANDLED;
```

```c
}

/* Bloc containing function for i3c char device */
int my_i3c_chardev_open(struct inode *inode, struct file *file) {
   if (!atomic_dec_and_test(&my_device_data.available)) {
      atomic_inc(&my_device_data.available);
      return -EBUSY;
   }
   return 0;
}

int my_i3c_tool_write_i3c_buffer(struct my_i3c_master *master,
                i3c_cmd_param *cmd_params)
{
   struct i3c_dev_desc *i3cdev;
   struct i3c_priv_xfer *xfers = kmalloc(sizeof(struct i3c_priv_xfer), GFP_KERNEL);
   i3cdev = my_i3c_tool_search_i3c_dev(master, cmd_params->dyn_addr);
   err = i3c_device_do_priv_xfers(i3cdev->dev, xfers, 1);
   kfree(xfers);
   return err;
}

int my_i3c_tool_read_i3c_buffer(struct my_i3c_master *master, i3c_cmd_param
*cmd_params, u8 *read_buffer)
{
   struct i3c_dev_desc *i3cdev;
   struct i3c_priv_xfer *xfers = kmalloc(sizeof(struct i3c_priv_xfer), GFP_KERNEL)
   xfers->data.in = read_buffer;
   i3cdev = my_i3c_tool_search_i3c_dev(master, cmd_params->dyn_addr);
   err = i3c_device_do_priv_xfers(i3cdev->dev, xfers, 1);
   kfree(xfers);
   return err;
}

int my_i3c_chardev_release(struct inode *inode, struct file *file) {
   atomic_dec(&my_device_data.available);
   return 0;
}

ssize_t my_i3c_chardev_read(struct file *file, char *buffer, size_t len, loff_t
*offset) { ... }
ssize_t my_i3c_chardev_write(struct file *file, char *buffer, size_t len, loff_t
*offset) { ... }

long my_i3c_chardev_ioctl(struct file *file, unsigned int cmd,unsigned long arg) {
   struct my_i3c_master *master = my_device_data.master;

   switch (cmd) {
   case my_i3c_CMD_CONF:
      memcpy(&master->conf, (i3c_config *)arg, sizeof(master->conf));
      my_i3c_master_set_config(master);
      break;
   case my_i3c_CMD_DAA:
      err = i3c_master_do_daa(&master->base);
      break;
   case my_i3c_CMD_WI3C:
      err = my_i3c_tool_write_i3c_buffer(master, &cmd_params);
      break;
   case my_i3c_CMD_RI3C:
      err = my_i3c_tool_read_i3c_buffer(master, &cmd_params, buffer);
      break;
   case my_i3c_CMD_WR3C:
      err = my_i3c_tool_write_i3c_buffer(master, &cmd_params);
      err = my_i3c_tool_read_i3c_buffer(master, &cmd_params, buffer);
      break;
   case my_i3c_CMD_STOP/EXIT:
      err = my_i3c_master_write_stop/exit(master);
      break;
   case my_i3c_CMD_GET_EVENT:
      err = copy_from_user(&ibi_event, (struct ibi_event *)arg,
                           sizeof(struct ibi_event)); break;
   case my_i3c_CMD_GET_DEVICE_LIST:
      list_for_each_entry_safe (...) {devices_info[idx++] = i3cdev->info;  }
      break;
   case my_i3c_CMD_CCC:
      err = my_i3c_master_send_ccc_cmd(&master->base, (i3c_cmd_ccc_param *)arg);
      break;
   case my_i3c_CMD_SET_HJ_MODE: ... ;break;
   default: err = -EPERM;
   }

   if (cmd == my_i3c_CMD_...) {
      err = copy_to_user(cmd_params.read_data, buffer, cmd_params.buffer_len);
   }
   return err;
}

// Definition of actions linked to the char device exposed to the userspace
struct file_operations fops = {
   .open = my_i3c_chardev_open,
   .read = my_i3c_chardev_read,
   .write = my_i3c_chardev_write,
   .unlocked_ioctl = my_i3c_chardev_ioctl,
   .release = my_i3c_chardev_release,
};

/*   -------------------- BLOC END --------------------------- */
const struct of_device_id my_i3c_master_of_match[] = {
   .compatible = "my-i3c-nfcc", // Link with device_tree definition of I3C entry
};
MODULE_DEVICE_TABLE(of, my_i3c_master_of_match);

int my_i3c_driver_remove(struct platform_device *op) {
   struct my_i3c_master *master = platform_get_drvdata(op);
   i3c_master_unregister(&master->base);
   device_destroy(my_i3c_class, MKDEV(major_number, 0));
   unregister_chrdev(major_number, DEVICE_NAME);
   return 0;
}

const struct i3c_master_control_ops my_i3c_master_ops = {
   .bus_init = my_i3c_master_bus_init, // initialize an I3C bus
   .do_daa = my_i3c_master_do_daa, // Dynamic Address Assignment: send DAS command and
then add all devices discovered
   .send_ccc_cmd = my_i3c_master_send_ccc_cmd, // Send a CCC command.
   .priv_xfers = my_i3c_master_priv_xfers, // Do private I3C SDR transfers.
};

// initialisation of i3c driver ( hardware link and userspace link)
int my_i3c_driver_probe(struct platform_device *pdev)
{
   major_number = register_chrdev(0, DEVICE_NAME, &fops); // Initializing the cdev
   my_i3c_device = device_create(my_i3c_class, MKDEV(major_number, 0), DEVICE_NAME);
   master = devm_kzalloc(&pdev->dev, sizeof(*master), GFP_KERNEL); // alloc master
   res = platform_get_resource(pdev, IORESOURCE_MEM, 0); // Get reg @ from device tree
   master->regs = devm_ioremap_resource(&pdev->dev, res);

   spin_lock_init(&master->xferqueue.lock); INIT_LIST_HEAD(&master->xferqueue.list);
   master->hj.mode = my_i3c_HOTJOIN_ACK;
   INIT_WORK(&master->hj.hjwork, my_i3c_master_ibi_hj);

   /* Get interrupt line from device tree */
   my_device_data.irq = irq_of_parse_and_map(pdev->dev.of_node, 0);
   devm_request_irq(&pdev->dev, my_device_data.irq,my_i3c_master_interrupt, 0,
DEVICE_NAME, &master);
   platform_set_drvdata(pdev, master); dev_set_drvdata(my_i3c_device, master);
   ret = i3c_master_register(&master->base, &pdev->dev, &my_i3c_master_ops, false);
   my_device_data.ibi_event = devm_kzalloc(&pdev->dev, sizeof(...), GFP_ATOMIC);

   /* enable the SLVSTART interruption catch to detect slave IBI request */
   my_i3c_master_ibi_upd_rules(IBIRULES_NOBYTE | IBIRULES_MSB0);
   master->ibi.enable = 1; // Enable IBI interruption & response

   my_device_data.master = master;
   atomic_set(&my_device_data.available, 1); // availability control a driver open call
}

struct platform_driver my_i3c_driver = {
   .probe = my_i3c_driver_probe,
   .remove = my_i3c_driver_remove,
   .driver = {
         .name = DEVICE_NAME,
         .of_match_table = of_match_ptr(my_i3c_master_of_match),
         .owner = THIS_MODULE
   },
};

int __init my_i3c_module_init(void) {
   my_i3c_class = class_create(THIS_MODULE, CLASS_NAME); // Register the device class
   retval = platform_driver_register(&my_i3c_driver);
   return retval;
}

void __exit my_i3c_module_exit(void) {
   class_unregister(my_i3c_class);
   class_destroy(my_i3c_class);
   platform_driver_unregister(&my_i3c_driver);
}

module_init(my_i3c_module_init);
module_exit(my_i3c_module_exit);
```