Echtzeitdatenverarbeitung

Dokumentation Bearbeiten

Gruppenmitglieder:	Charlotte Friedemann(7011467)
	Johannes Theiner(7010923)
Semester:	Wintersemester 2020/21
letzte Änderung:	2. Januar 2021

1 Tasks

Um einen konsistenten Startpunkt zu erhalten werden bei allen Tasks zu Beginn sämtliche vom entsprechenden Task verwendeten Aktoren auf Ausgangsposition zurückgesetzt.

1.1 Drehteller

Nach dem Zurücksetzen vom Drehteller und des Auswerfers am Eingang fährt der Drehteller zuerst fünf Runden, bei denen der Auswerfer am Ausgang bedingungslos betätigt wird, um Teile die sich vielleicht noch in der Anlage befinden aus dieser zu entfernen.

Um eine Synchronisierung zu erlangen wird der Drehteller erst aktiv, wenn Prüfer, Bohrer und Auswerfer mit einer Nachricht auf die Status Mailbox signalisiert haben das sie mit ihren Aktionen fertig sind.

Liegt auf mindestens einem Sensor(Eingang, Tester oder Bohrer) ein Teil wird der Drehteller aktiviert und erst wieder deaktiviert, wenn dieser wieder in Position ist. Nun wird den anderen Tasks über Nachrichten signalisiert das diese aktiv werden können und der Prozess beginnt erneut.

1.2 Prüfer

Sobald der Prüfer aktiv werden darf(Nachricht auf Status Mailbox) wird überprüft ob ein Teil auf dem Sensor liegt. Liegt ein Teil auf dem Sensor, fährt der Prüfer aus und das Testergebnis wird dem Bohrer über eine Nachricht mitgeteilt. Nun wird der Prüfer eingefahren und der Drehteller kann wieder aktiv werden. Der Prozess beginnt nun wieder von vorne.

1.3 Bohrer

Wird ein Werkstück durch den Sensor erkannt wird abhängig von der Nachricht des Prüfers der Bohrer angeschaltet, heruntergefahren und das Werkstück festgehalten. Nach einer kurzen Wartezeit wird der Bohrer wieder nach oben gefahren, ausgeschaltet und das Werkstück losgelassen. Nachdem der Auswerfer über ein zu erwartendes Teil benachrichtigt wurde, wird die Kontrolle wieder an den Drehteller übergeben.

1.4 Auswerfer

Da für den Auswerfer keine Sensoren existieren sendet der Bohrer den Status seines Sensors per Nachricht an den Auswerfer, der auf Basis dieser auslöst.

1.5 Diagramme

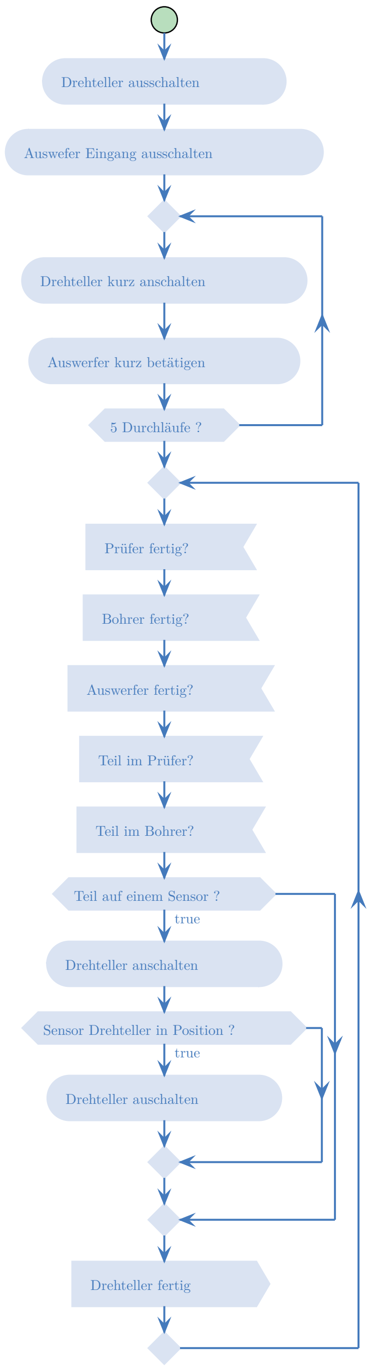

Abbildung 1: SDL Diagramm Drehteller

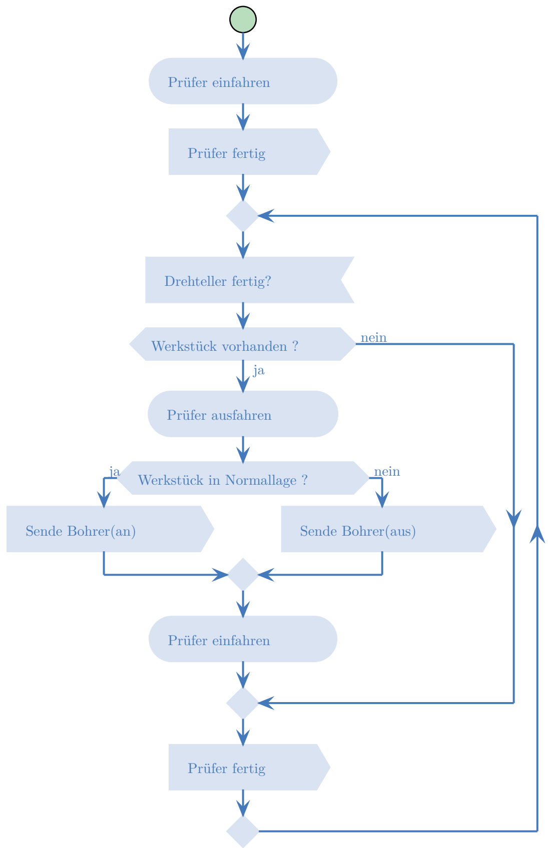

Abbildung 2: SDL Diagramm Prüfer

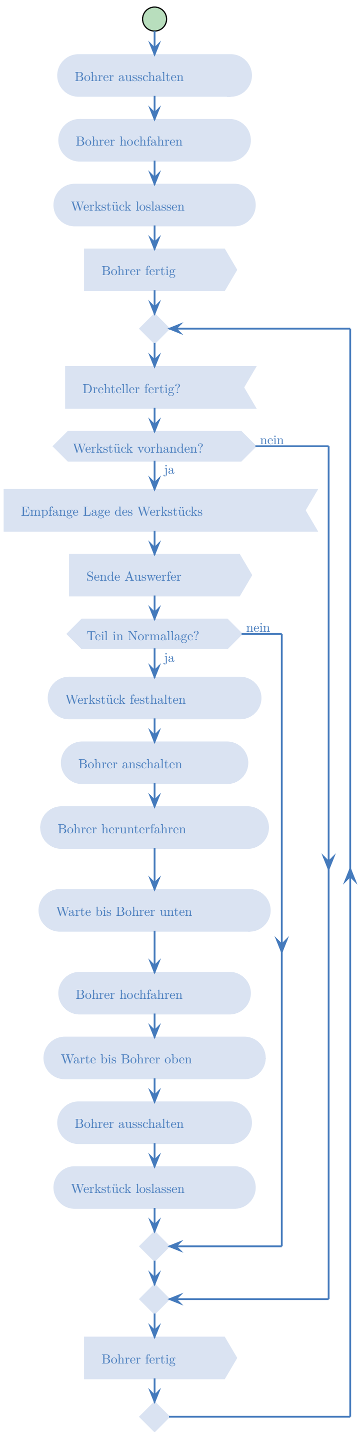

Abbildung 3: SDL Diagramm Bohrer

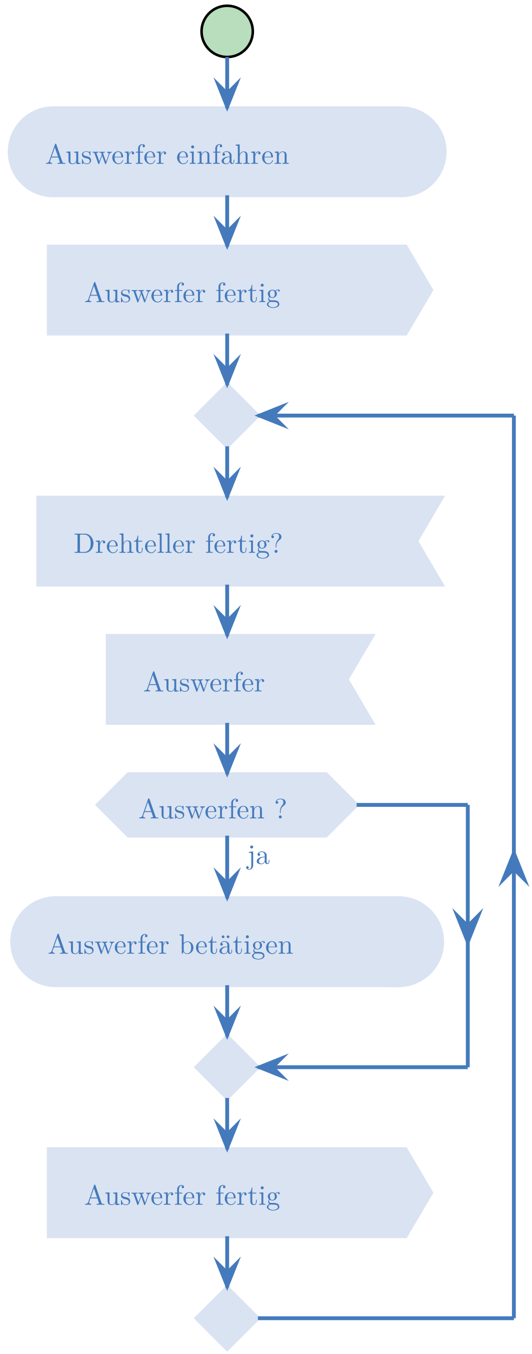

Abbildung 4: SDL Diagramm Auswerfer

2 Hilfsfunktionen

Zur Vereinfachung des Leseflusses und der besseren Nachvollziehbarkeit des Codes werden wiederkehrende Programmteile in Hilfsfunktionen ausgelagert. Passiert während des Durchlaufens der einzelnen Funktionen ein Fehler, so wird die erste hier beschriebene Hilfsfunktion aufgerufen.

2.1 void end(bool fail)

Bei Aufruf dieser Hilfsfunktion werden sämtliche Taks, Mailboxen, Modbus-Verbindungen und Semaphoren deinitialisiert und der Timer gestoppt. Falls der Übergabeparameter `fail == true` ist, so wird eine entsprechende Nachricht auf der Konsole ausgegeben.

2.2 int readAll(int type, int *result)

Das Auslesen aller zur Verfügung stehenden Bits, abhängig vom Typ der Verbindung, erfolgt mithilfe der nicht Thread-sicheren Funktion readAll. Dabei wird dieser Funktion sowohl der Verbindungs-Typ (`DIGITAL_IN, DIGITAL_OUT, ANALOG_IN, ANALOG_OUT`), als auch ein Pointer auf den Speicherplatz übergeben, an welchem das Ergebnis gespeichert wird. Bei erfolgreichem Durchlaufen dieser Funktion wird eine 0 zurückgegeben, ansonsten eine 1.

2.3 int readData(int part)

Im Gegensatz zur Hilfsfunktion `readAll()` liest die Funktion readData nur ein einzelnes Bit aus und gibt dieses zurück. Damit die Richtigkeit der Daten während des Auslesens gewährleistet werden kann, ist diese Funktion mithilfe einer Semaphore Thread-sicher gemacht worden. Nach dem Auslesen aller Bits des Modbus wird der Wert der Funktion `readAll()` mit der in `part` übergebenen Bitmaske verundet und somit isoliert. Dieses einzelne Bit wird nun in `result` gespeichert und die Semaphore wieder freigegeben. Bei fehlerfreiem Aufruf der `readAll()` -Funktion wird das spezifizierte einzelne Bit zurückgeben, falls nicht, so wird in die `end()` -Hilfsfunktion gesprungen und alle Tasks, Mailboxen, etc deinitialisiert und gestoppt.

2.4 void disable(int actor)

Diese Funktion deaktiviert den im Übergabeparameter spezifizierten Aktor. Da auch in diesem Fall die Daten aus dem Modbus während des Lesezugriffs nicht verändert werden dürfen ist diese Funktion Thread-sicher. Zur Kommunikation mit dem jeweiligen Aktor wird die gesamte Bitfolge über *readAll()* ausgelesen. Ist dies ohne Fehler geschehen, so wird die empfangene Bitfolge mit der negierten Bitmaske des anzusprechenden Aktors verundet. Diese neue, sich nur in einem Bit unterscheidende Bitfolge wird auf den Modbus geschrieben und die Semaphore freigegeben.

2.5 void sendMail(MBX * mailbox, int msg)

Das Senden einer blockierenden Nachricht (`msg`) an eine über mailbox spezifierte Mailbox wird mithilfe dieser Funktion realisiert.

2.6 void sendMailNonBlocking(MBX * mailbox, int msg)

Im Unterschied zur Funktion `sendMail()` wird hier eine nicht-blockierende Nachricht (msg) an die in mailbox spezifizierte Mailbox gesendet.

2.7 int receiveMail(MBX * mailbox)

Diese Hilfsfunktion fasst das Erhalten einer blockierenden Nachricht an der im Übergabeparameter spezifizierten Mailbox zusammen und gibt den Nachrichteninhalt zurück.

2.8 int receiveMailNonBlocking(MBX * mailbox)

Analog zu `receiveMail()` wird auch hier einen Nachricht an der im Übergabeparamter spezifizierten Mailbox erhalten und deren Inhalt zurückgegeben, jedoch ist diese nicht blockierend.

2.9 void sleepMs(int ms)

Da bei der Kommunikation der Tasks untereinander oder der Abarbeitung an den einzelnen Stationen teilweise Pausen benötigt werden, ist diese Funktion unerlässlich. Dabei wird mithilfe des Übergabeparameters festgelegt, für wie viele Millisekunden ein Tasks pausieren soll.

3 Quellcode

```c
#include <rtai_mbx.h>
#include <rtai_sched.h>
#include <rtai_sem.h>
#include <sys/rtai_modbus.h>

//======================
//Author: Charlotte Friedemann(7011467), Johannes Theiner(7010923)
//Description: Praktikum EZDV Gruppe A5(Bearbeiten 2)
//Created: 19.10.2020
//Finished: 30.11.2020
//======================

#define SENSOR_PART_TURNTABLE 1<<0
#define SENSOR_PART_DRILL 1<<1
#define SENSOR_PART_TESTER 1<<2
#define SENSOR_DRILL_UP 1<<3
#define SENSOR_DRILL_DOWN 1<<4
#define SENSOR_TURNTABLE_POS 1<<5
#define SENSOR_PART_TEST 1<<6

#define ACTOR_DRILL 1<<0
#define ACTOR_TURNTABLE 1<<1
```

```c
#define ACTOR_DRILL_DOWN 1<<2
#define ACTOR_DRILL_UP 1<<3
#define ACTOR_PART_HOLD 1<<4
#define ACTOR_TESTER 1<<5
#define ACTOR_EXIT 1<<6
#define ACTOR_ENTRANCE 1<<7

MODULE_LICENSE("GPL");

static RT_TASK turntable_task, drill_task, tester_task, output_task;
static MBX turntable_status_mbx, tester_status_mbx;
static MBX output_status_mbx, drill_status_mbx;
static MBX output_data_mbx, drill_data_mbx, turntable_data_mbx;
static SEM semaphore;

int connection;

char node[] = "modbus-node";

/**
 * deinitialize all tasks, mailboxes, modbus connections & semaphores
 * @param fail should a error message be printed.
 */
void end(bool fail) {
    rt_modbus_disconnect(connection);
    rt_task_delete(&turntable_task);
    rt_task_delete(&tester_task);
    rt_task_delete(&drill_task);
    rt_task_delete(&output_task);

    rt_mbx_delete(&turntable_status_mbx);
    rt_mbx_delete(&tester_status_mbx);
    rt_mbx_delete(&drill_status_mbx);
    rt_mbx_delete(&output_status_mbx);
    rt_mbx_delete(&turntable_data_mbx);
    rt_mbx_delete(&drill_data_mbx);
    rt_mbx_delete(&output_data_mbx);

    rt_sem_delete(&semaphore);

    stop_rt_timer();

```

```
    if(fail) {
        rt_printk("an error has occurred.\n");
        rt_printk("module needs to be restarted.\n");
    }
}

/**
 * read all bits of the specified type
 * this function is *not* thread safe.
 * @param type (DIGITAL_IN, DIGITAL_OUT, ANALOG_IN, ANALOG_OUT)
 * @param result value to write the result to
 * @return status code(0: success, 1: failure)
 */
int readAll(int type, int *result) {
    int res = rt_modbus_get(connection, type, 0,
                            (unsigned short *) result);
    return res;
}

/**
 * read a single bit for the specified part
 * this function is thread safe.
 * @param part bitmask to read the sensor/actor
 * @return read bit.
 */
int readData(int part) {
    int value = 0;
        rt_sem_wait(&semaphore);
        //the semaphore needs to be locked before this code executes,
        // so we can't adhere to the ISO standard here.
    int code = readAll(DIGITAL_IN, &value);
    int result = (part & value);
    rt_sem_signal(&semaphore);
    if(code) end(true);
    return result;
}

/**
 * disable a specified actor of the system.
 * this function is thread safe.
 * @param actor bitmask of the actor to disable
 */
```

```
void disable(int actor) {
        int value = 0;
    rt_sem_wait(&semaphore);
    if(readAll(DIGITAL_OUT, &value)) end(true);
    else {
            int result = rt_modbus_set(connection, DIGITAL_OUT, 0,
                                value &= ~actor);
            rt_sem_signal(&semaphore);
            if (result) end(true);
    }
}

/**
 * enable one specified actor of the system.
 * this function is thread safe.
 * @param actor bitmask of the actor to enable
 */
void enable(int actor) {
        int output = 0;
    rt_sem_wait(&semaphore);
    if (readAll(DIGITAL_OUT, &output)) end(true);
    else {
            int result = rt_modbus_set(connection, DIGITAL_OUT, 0,
                                output |= actor);
            rt_sem_signal(&semaphore);
            if(result) end(true);
    }

}

/**
 * send a blocking message to the specified mailbox
 * @param mailbox where should the message be send to ?
 * @param msg message to send
 */
void sendMail(MBX * mailbox, int msg) {
    int mbxStatus = rt_mbx_send(mailbox, &msg, sizeof(int));
    if(mbxStatus == EINVAL) end(true);
}

/**
 * send a non blocking message to the specified mailbox
```

```
 * @param mailbox where should the message be send to ?
 * @param msg message to send
 */
void sendMailNonBlocking(MBX * mailbox, int msg) {
    int mbxStatus = rt_mbx_send_if(mailbox, &msg, sizeof(int));
    if(mbxStatus == EINVAL) end(true);
}

/**
 * receive a blocking message
 * @param mailbox mailbox to receive the message
 */
int receiveMail(MBX * mailbox) {
    int msg = 0;
    int mbxStatus = rt_mbx_receive(mailbox, &msg, sizeof(int));
    if(mbxStatus == EINVAL) end(true);
    return msg;
}

/**
 * receive a non blocking message
 * @param mailbox mailbox to receive the message
 */
int receiveMailNonBlocking(MBX * mailbox) {
    int msg = 0;
    int mbxStatus = rt_mbx_receive_if(mailbox, &msg, sizeof(int));
    if(mbxStatus == EINVAL) end(true);
    return msg;
}

/**
 * sleep for x milliseconds
 * @param ms specified time to sleep in milliseconds.
 */
void sleepMs(int ms) {
    rt_sleep(ms * nano2count(1000000));
}

/**
 * turntable task
 */
static void turntable(long data) {
```

```
        int times = 0;
        rt_printk("started turntable task\n");

        //reset everything first
    disable(ACTOR_TURNTABLE);
    disable(ACTOR_ENTRANCE);

    //throw out all parts that might be on the table.
    do {
            enable(ACTOR_TURNTABLE);
            sleepMs(1000);
            disable(ACTOR_TURNTABLE);
            sleepMs(500);
            enable(ACTOR_EXIT);
            sleepMs(500);
            disable(ACTOR_EXIT);
            times++;
    }while (times < 5);

    //start processing
    while (1) {
            //receive status mail from: tester, drill & output
        receiveMail(&turntable_status_mbx);
        receiveMail(&turntable_status_mbx);
        receiveMail(&turntable_status_mbx);
        //always read from mailbox to make sure that no overflow occurs.
        int msg1 = receiveMailNonBlocking(&turntable_data_mbx);
        int msg2 = receiveMailNonBlocking(&turntable_data_mbx);
        //if a part is on any of the sensors
        if(readData(SENSOR_PART_TURNTABLE) || msg1 || msg2) {
            enable(ACTOR_TURNTABLE);
            sleepMs(100);
                        if (!readData(SENSOR_TURNTABLE_POS)) {
                                do {
                                        sleepMs(10);
                                } while (readData(SENSOR_TURNTABLE_POS) == 0);
                        }
                disable(ACTOR_TURNTABLE);
            sleepMs(500);
        }
        //send status mails
        sendMail(&tester_status_mbx, 1);
```

```
        sendMail(&drill_status_mbx, 1);
        sendMail(&output_status_mbx, 1);
    }
}

/**
 * tester task
 */
static void tester(long data) {
    rt_printk("started tester task\n");

    //reset everything first
        disable(ACTOR_TESTER);
        sendMail(&turntable_status_mbx, 1);
        //start processing
    while (1) {
        receiveMail(&tester_status_mbx);
            if(readData(SENSOR_PART_TESTER)) {
            //turntable should turn on the next turn.
                    sendMailNonBlocking(&turntable_data_mbx, 1);
                    enable(ACTOR_TESTER);
            sleepMs(500);
            //should the drill be active on the next turn ?
                    if(readData(SENSOR_PART_TEST)) {
                            sendMailNonBlocking(&drill_data_mbx, 1);
                    }else {
                            sendMailNonBlocking(&drill_data_mbx, 0);
                    }
                    disable(ACTOR_TESTER);
            }
            sendMail(&turntable_status_mbx, 1);
    }
}

/**
 * drill task
 */
static void drill(long data) {
        rt_printk("started drill task\n");
        //reset everything first
    disable(ACTOR_DRILL);
    disable(ACTOR_PART_HOLD);
```

```
    disable(ACTOR_DRILL_DOWN);

    enable(ACTOR_DRILL_UP);
    if (!readData(SENSOR_DRILL_UP)) {
        do {
                sleepMs(10);
        } while (readData(SENSOR_DRILL_UP) == 0);
    }
    disable(ACTOR_DRILL_UP);
    sendMail(&turntable_status_mbx, 1);
    //start processing
    while (1) {
        receiveMail(&drill_status_mbx);
        //if part is in drill
            if(readData(SENSOR_PART_DRILL)) {
            //turntable should be active on the next turn.
                    sendMailNonBlocking(&turntable_data_mbx, 1);
            //read test result from mailbox.
                    if(receiveMailNonBlocking(&drill_data_mbx)) {
                            enable(ACTOR_PART_HOLD);
                            enable(ACTOR_DRILL);
                            enable(ACTOR_DRILL_DOWN);
                            if (!readData(SENSOR_DRILL_DOWN)) {
                                    do {
                                            sleepMs(10);
                                } while (readData(SENSOR_DRILL_DOWN) == 0);
                            }
                sleepMs(1000);
                                disable(ACTOR_DRILL);
                                disable(ACTOR_DRILL_DOWN);
                                enable(ACTOR_DRILL_UP);
                                if (!readData(SENSOR_DRILL_UP)) {
                                        do {
                                                sleepMs(10);
                                        } while (readData(SENSOR_DRILL_UP) == 0);
                                }
                                disable(ACTOR_DRILL_UP);
                                disable(ACTOR_PART_HOLD);
                    }
            //should the output be active on the next turn ?
                    sendMailNonBlocking(&output_data_mbx, 1);
            }
```

```
            sendMail(&turntable_status_mbx, 1);
    }
}

/**
 * output task
 */
static void output(long data) {
        rt_printk("started output task\n");
        //reset everything first
    disable(ACTOR_EXIT);
    sendMail(&turntable_status_mbx, 1);
    //start processing
    while (1) {
        receiveMail(&output_status_mbx);
        //should the output be activated
            if(receiveMailNonBlocking(&output_data_mbx)) {
                    enable(ACTOR_EXIT);
            sleepMs(500);
                    disable(ACTOR_EXIT);
            }
            sendMail(&turntable_status_mbx, 1);
    }
}

static int __init
example_init(void) {
    rt_set_oneshot_mode();
    start_rt_timer(0);
    modbus_init();

    rt_printk("init: started\n");
    if ((connection = rt_modbus_connect(node)) == -1) {
        rt_printk("init: could not connect to %s\n", node);
        return -1;
    }

    rt_sem_init(&semaphore, 1);

    if (rt_task_init(&turntable_task, turntable, 0, 1024, 0, 0, NULL)) {
        rt_printk("turntable: cannot initialize task\n");
        end(true);
```

```
    }
    if (rt_task_init(&drill_task, drill, 0, 1024, 0, 0, NULL)) {
       rt_printk("drill: cannot initialize task\n");
        end(true);
    }
    if (rt_task_init(&output_task, output, 0, 1024, 0, 0, NULL)) {
        rt_printk("output: cannot initialize task\n");
         end(true);
    }
    if (rt_task_init(&tester_task, tester, 0, 1024, 0, 0, NULL)) {
        rt_printk("tester: cannot initialize task\n");
        end(true);
    }

    if (rt_mbx_init(&turntable_status_mbx, sizeof(int))) {
            end(true);
    }
    if (rt_mbx_init(&tester_status_mbx, sizeof(int))) {
        end(true);
    }
    if (rt_mbx_init(&drill_status_mbx, sizeof(int))) {
        end(true);
    }
    if (rt_mbx_init(&output_status_mbx, sizeof(int))) {
        end(true);
    }
    if (rt_mbx_init(&output_data_mbx, sizeof(int))) {
        end(true);
    }
    if (rt_mbx_init(&drill_data_mbx, sizeof(int))) {
            end(true);
    }
    if (rt_mbx_init(&turntable_data_mbx, sizeof(int))) {
            end(true);
    }

    rt_task_resume(&turntable_task);
    rt_task_resume(&drill_task);
    rt_task_resume(&output_task);
    rt_task_resume(&tester_task);
    rt_printk("loaded module Bearbeiten2\n");
    return (0);
```

```
}

static void __exit
example_exit(void) {
    end(false);
    rt_printk("module Bearbeiten2 unloaded\n");
}

module_exit(example_exit)
module_init(example_init)
```