ECE 5780 Lab 3*

Due Wednesday March 16, 2016 at 11:30pm in ENGR 328 (320/400 points)†

Objectives

To learn how to program the microcontroller of an embedded device in a real-time operating system in C.

1 Logistics

You are to work in a team of 1-2 members in ENGR 328. Check out one LEGO Mindstorms (http://mindstorms.lego.com/en-us/Default.aspx) kit from the ECE store. You and your teammate should be able to independently describe the solutions. There is no formal report to turn in. Please upload your well-documented, well-structured code on Canvas after you have presented a demo to me on Wednesday March 16 at 11:30pm in ENGR 328.

2 LEGO Car Competition

The last part of this lab consists of building a robot car that can follow a track as fast as possible (see below). All groups have to show that their car is able to complete the challenge. The group with the fastest car wins, which implies eternal glory. The competition will take place during class on **Friday March 18** in ENGR 328.

3 LEGO Mindstorms

The LEGO Mindstorms Robotics Invention System consists of numerous LEGO pieces, the NXT unit (i.e., the brain), three motors, two touch sensors, an ultrasonic sensor, and a color sensor. The NXT unit is an autonomous programmable microcomputer (an Atmel 32-bit ARM7 processor, specifically, AT91SAM7S256) running at 48 MHz. The NXT brick can be used to control actuators such as motors and read inputs from the various sensors. The NXT brick also has an LCD display, which is useful for printing out information and debugging, as well as USB and Bluetooth communication ports. The NXT unit can easily be attached to the LEGO building blocks and pieces.

*Lab Assignment Courtesy of Martin Stigge and Wang Yi

†ECE 5780: 320 points, ECE 6780: 400 points

Instead of the standard firmware and default programming platform of the NXT, we will use nxtOSEK, which is a port of OSEK to the LEGO NXT platform. OSEK is a real-time operating system (RTOS) introduced by a consortium of mostly German car manufacturers and is widely used in industry. The NXT implementation uses ECRobot for low-level hardware access to sensors and motors. It also uses Newlib for standard C functions (similar to libc).

A program contains two parts.

- The program source code, e.g., `myprogram.c`
- The system's description in OIL format, e.g., `myprogram.oil`

The compilation toolchain first compiles the C file into an ARM binary and then generates the entire system's binary according to the description in the OIL file. This includes the definitions for all tasks, resources, event objects, etc...

4 Getting Started with nxtOSEK and the ECRobot API

All software necessary to work with OSEK on the NXT platform is installed on the lab machines in ENGR 328. To start the lab, you first need to upgrade the firmware on the NXT brick, since the original LEGO firmware does not support the nxtOSEK binaries.

1. Reset the NXT unit by pressing the reset button at the back of the NXT for more than 5 seconds while the NXT is turned on (see the manual that came with your kit for more information). The NXT will make an audible sound when it is in firmware update mode.
2. Download the Enhanced NXT firmware file from the course website and move it to the NeXTTool directory.
3. Start Cygwin and change the working directory to the NeXTTool directory.
4. Connect the NXT brick to your computer's USB port.
5. Type the following command to upload the Enhanced NXT firmware onto the NXT brick.

   ```
   $ ./NeXTTool.exe /COM=usb -firmware=lms_arm_nbcnxc_107.rfw
   ```

   Program upload may take a minute.
6. Remove the batteries from the NXT unit and re-insert them. Press the orange rectangle button to turn on the unit. The Enhanced NXT firmware has the same GUI as the LEGO standard firmware.

Here are some references that may prove to be useful

- nxtOSEK C API Reference (http://lejos-osek.sourceforge.net/ecrobot_c_api.htm)
- OSEK OS Specification (http://portal.osek-vdx.org/files/pdf/specs/deprecated/os221.pdf)
- OSEK OIL File Specification (http://portal.osek-vdx.org/files/pdf/specs/oil25.pdf)
- Newlib Reference Manual (http://sourceware.org/newlib/libc.html)
- Newlib Math Library Reference Manual (http://sourceware.org/newlib/libm.html)

Additional resources are also available on the course website and on the Internet.

5 Compiling and Uploading Your Program

To compile a program, you need to have a `.c` file, a `.OIL` file, and a Makefile (with the `TARGET` field set to the name of your program. You can try out the "Hello World" program under `nxtOSEK/samples_c/helloworld/`. To compile, type

```
$ make all
```

After a successful compilation, an `.rxe` file will be created. This is the binary that will run on the NXT brick. To upload it, connect the NXT unit to your computer USB port and type the following.

```
$ chmod u=+wrx rxeflash.sh
$ ./rxeflash.sh
```

The last command will automatically upload the `.rxe` file to your NXT brick. You can now disconnect the NXT unit from your computer and test out your code. **Important:** The NXT unit must be on for the upload process will fail.

6 Warmup

You will write a simple program to use the color sensor as a light sensor and display the current reading.

Obtain the skeleton C and OIL files from the course website. In the C file, we declare the following OSEK hooks.

```
void ecrobot_device_initialize() {}
void ecrobot_device_terminate() {}
void user_1ms_isr_type2() {}
```

These hooks are used to execute custom code when the device is initialized, shut down, and when a timer interrupt is called once every millisecond, respectively. You can ignore them for now but we will use them later for various purposes.

From the skeleton C file, you can see that a LightSensorTask is both declared and defined. Open `warmup.oil` to see the system description. In there, we describe the CPU we want to use, certain properties of the OSEK scheduler, and finally one task named "LightSensorTask". The task will start automatically at system startup ("AUTOSTART = TRUE"). Other properties are not important right now, but will be used later on in the lab.

Now, attach a color sensor to the NXT brick and fill in the rest of LightSensorTask. Your code should do the following.

- Display "Welcome to My World!"
- Use the color sensor as a light sensor and display the current light sensor A/D data repeatedly with a delay of 100 ms in between.

Note that your code should include an infinite loop so, technically, `TerminateTask()` will never be called. You should update the screen every second.

The following functions (and others) may come handy.

- `display_clear()`
- `display_string()`
- `display_update()`
- `ecrobot_get_nxtcolorsensor_light()`

- `systick_wait_ms()`

Hint 1: Consult the sample C files under `nxtOSEK/samples_c/` for many useful examples, especially the ones from the LEJOS-OSEK website (http://lejos-osek.sourceforge.net/samples.htm). You can also refer to the nxtOSEK C API reference (http://lejos-osek.sourceforge.net/ecrobot_c_api.htm).

Hint 2: Note that the color sensor needs to be initialized before usage. This should be done in the `ecrobot_device_initialize()` function. Similarly, you should deactivate the sensor in the `ecrobot_device_terminate()` function.

Hint 3: If your display is not showing anything, try using the `display_goto_xy()` function. You may have accidentally written your output to a place outside the screen.

Hint 4: If the compilation fails because of path errors (i.e., something is not found), check the `ecrobot.mak` and `tool_gcc.mak` files under `nxtOSEK/ecrobot`. You may also need to uncomment Line 79 in the `ecrobot.mak` file and set the appropriate path.

Compile your program and upload it to the NXT brick. Try measuring the light values of different surfaces.

7 Event-Driven Scheduling

In this part of the lab, you will learn how to program an event-driven schedule with nxtOSEK. The target application will be a LEGO car that drives forward as long as you press a touch sensor and it senses a table underneath its wheels with the help of a light sensor. Build a LEGO car that can drive on wheels. You may find inspiration in the manual included in the LEGO box or by looking at other designs on the Internet. Connect a touch sensor to one of the sensor input ports.

OSEK defines an event mechanism that can be used to schedule actions. Events may be generated by external sources, interrupt service routines (ISRs) or even other tasks. This allows tasks to immediately react to signals from various sources. With OSEK, events are defined per task. Only the "owner" task can wait for an event to occur. Events are stored in a data structure that needs to be reset after the occurrence of the event and this reset can only be done by the owner task. However, any task as well as ISRs may generate events for other tasks. Further, tasks may read the status of events of other tasks. You can find more information about the OSEK event mechanism in Chapters 7 and 13.5 of the OSEK OS manual (http://portal.osek-vdx.org/files/pdf/specs/deprecated/os221.pdf).

Each task includes an "event mask" which is a number of bits representing an event. A task can wait for an event using `WaitEvent()` and is suspended (i.e., blocked) until this event occurs (i.e., until the corresponding bit in its event mask is set). Meanwhile, the CPU is available for other tasks to execute. Tasks can wait for multiple events at once using an OR operation on the event's bits `WaitEvent(Event1 | Event2);`. After the event occurred, the owner task is responsible for clearing the corresponding bit in the event mask using `ClearEvent(Event1)`. If a task is waiting for multiple events at once, it may read its event mask to determine which of them occurred as follows.

```
EventMaskType eventmask = 0;
...
WaitEvent(Event1 | Event2);
GetEvent(MyTask, &eventmask);
if (eventmask & Event2) {
    /* Event 2 occured */
    ClearEvent(Event2);
    ...
}
```

Create a new program with a task "MotorControlTask" that does the following in an infinite loop.

- Wait for event "TouchOnEvent"
- Make the car move forward by activating the motors
- Wait for event "TouchOffEvent"
- Make the car stop

The actions should occur as soon as the user presses or releases the touch sensor button. Don't forget to declare the task using `DeclareTask()`.

In order for the system to recognize the events, these events need to be declared using `DeclareEvent()` right where you declare the task. In addition, create a new OIL file and specify the MotorControlTask as before with the following addition.

```
EVENT = TouchOnEvent;
EVENT = TouchOffEvent;
```

Since the events themselves need to be declared, add the following lines before the task definition.

```
EVENT TouchOnEvent { MASK = AUTO; };
EVENT TouchOffEvent { MASK = AUTO; };
```

The keyword "AUTO" tells OSEK to choose the appropriate bits. You could also specify the bit number yourself instead.

You now need to generate the touch sensor events. In general, these events should be generated by the appropriate ISRs that would be called when the corresponding interrupts are released. Unfortunately, the sensors on the NXT brick use polling; they need to be asked for their state again and again, instead of getting active themselves when something interesting happens. Our workaround for this is to create a small, second task that checks the sensors periodically (about every 10ms). If the state of the sensor has changed, it generates the appropriate event for the MotorControlTask. In order to do this, declare and implement a task "EventDispatcherTask". It should call the appropriate API function to read the touch sensor and compare it to its old state (a static variable may be useful for that). If the state has changed, the task should release the corresponding event:

```
   SetEvent(MotorControlTask, TouchOnEvent);
... or ...
   SetEvent(MotorControlTask, TouchOffEvent);
```

Just as before, put your code in an infinite loop with a delay at the end of the loop body. (We will use proper periodic tasks in the next part of the lab.) In order for the MotorControlTask to have priority over the EventDispatcherTask, make sure to assign a lower priority to the latter in the OIL file. Otherwise, the infinite loop containing the sensor reading would make the system completely busy, rendering it unresponsive.

Compile, upload, and test your program. Now, attach a light sensor to your car. The sensor should be somewhere in front of the wheel, close to the ground, and pointing downwards. Extend your program to also react to this light sensor. That is, the car should stop not only when the touch sensor is released, but also when the light sensor detects that the car is very close to the edge of a table. (You may need to play a little bit with your previous program in order to find a suitable value range.) The car should only start moving again when it is back on the table and the touch sensor is pressed (again).

Edge detection should happen in the EventDispatcherTask and be communicated to the MotorControlTask via the event mechanism. Use two new events for that purpose. Make sure you declare and define all the events properly in both the C and the OIL file.

There are some requirements. First, The EventDispatcherTask should create independent events from the touch and light sensors. Second, the logic used to determine whether to run or stop the car should

only happen in the body of the MotorControlTask. Third, the MotorControlTask must not directly read the sensors nor communicate with the EventDispatcherTask by means other than the event system. In other words, shared variables are not allowed. Fourth, the EventDispatcherTask should only generate events when states change. That is, only one event should be created when the touch sensor is pressed down and one when it is released. The same goes for the light sensor. Don't spam the event system with redundant information.

Hint: Some of the samples found on the LEJOS-OSEK website (http://lejos-osek.sourceforge.net/samples.htm) may be **very** useful.

Just for Fun: If you have time and/or are bored, try adding another task, say a DisplayTask. Explore setting different priorities for this task and observe the behavior of the system.

8 Periodic Scheduling

For this part of the lab, the objective is to make your car keep a constant distance from some object in front of it. In addition, the user may press the touch sensor to command the car to go backward.

We will create the following three periodic tasks.

- A task "MotorControlTask" that controls the motors and receives commands from the other tasks.
- A task "ButtonPressTask" that senses the state of the touch sensor and sends commands to the MotorcontrolTask.
- A task "DisplayTask" that displays some interesting information about the system state.

Note that there is no task that senses the distance yet. This will come later.

We start by defining a periodic task. Define the "MotorControlTask" in the OIL file as follows.

```
TASK MotorcontrolTask
{
  AUTOSTART = FALSE;
  PRIORITY = 1; /* Smaller value means lower priority */
  ACTIVATION = 1;
  SCHEDULE = FULL;
  STACKSIZE = 512; /* Stack size */
};
```

As can be seen from above, this task does not automatically start. We will release it again and again at regular intervals (hence it is a periodic task). In order to do so, we need two tools: (1) a counter that is increased every millisecond, and (2) an alarm that can activate the task every time the counter reaches a value of 50. This means that the task has a period of 50 ms.

The counter is defined as follows in the OIL file.

```
COUNTER SysTimerCnt
{
  MINCYCLE = 1;
  MAXALLOWEDVALUE = 10000;
  TICKSPERBASE = 1; /* One tick is equal to 1msec */
};
```

Further, an alarm is defined as follows.

```
ALARM cyclic_alarm
{
  COUNTER = SysTimerCnt;
  ACTION = ACTIVATETASK
```

```
    {
        TASK = MotorControlTask;
    };
    AUTOSTART = TRUE
    {
        ALARMTIME = 1;
        CYCLETIME = 50;
        APPMODE = appmode1;
    };
  };
```

Each time the specified COUNTER is increased by CYCLETIME, the given TASK is activated. To make this work, you need to do two things in your C program.

- Declare the counter in your C program using `DeclareCounter()`
- Since OSEK does not increment the counter for us, we need to do this ourselves. This is where the 1ms ISR hook is useful. Replace the empty definition of `user_1ms_isr_type2()` in your program with the following.

```
void user_1ms_isr_type2()
{
    SignalCounter(SysTimerCnt);
}
```

Before we implement the actual tasks, we need a way for them to communicate. We will use a data structure that is used by the sensing tasks to communicate to the MotorControlTask. Define a "driving command" structure as follows.

```
struct dc_t {
    U32 duration;
    S32 speed;
    int priority;
} dc = {0, 0, PRIO_IDLE};
```

Define priorities `PRIO_IDLE` and `PRIO_BUTTON` with values 10 and 20 using `#define`. In addition, write a function `change_driving_command(int priority, int speed, int duration)`, which can be used by tasks to update `dc`. A task can at any time overwrite the value(s) of this struct as long as its priority is higher than the current priority of the struct.

Using this struct, define the "MotorControlTask" to execute the driving command specified by `dc`. That is, if `dc.duration` is positive, the speed of the motors should be set according to `dc.speed`. Then, `dc.duration` will be decreased by the period of the MotorControlTask. If `dc.duration` is no greater than zero, `dc.priority` is set to `PRIO_IDLE` and the car stops moving.

As for the "ButtonPressTask", if the button of the touch sensor is pressed, the task should try to set the driving command to drive backward for 1000 ms with priority `PRIO_BUTTON` (the actual speed is up to you). This task has a period of 10 ms. Note that the body of a periodic task should not include an infinite loop since it is periodically activated. In addition, the body should end with `TerminateTask()`.

Finally, the "DisplayTask" outputs some useful information such as current speed, duration, etc... on the LCD. Its period is 100 ms.

A potential problem that may arise is that the struct `dc` is shared and used by several tasks that are reading from and writing to it. Use the resource concept of OSEK to achieve simple mutual exclusion. You can declare resources with `DeclareResource()` and use them with `GetResource()` and

`ReleaseResouce()`. Don't forget to define them in the OIL file as well. Consult the OIL specification (http://portal.osek-vdx.org/files/pdf/specs/oil25.pdf) for details.

Test your program. Right now, the car either stops or moves backward only. We will now incorporate distance measurement. Add an ultrasonic sensor pointing forward and connect it to a sensor port (leaving the light sensor attached may help later). Extend your program to include the "DistanceTask", which has a period of 100 ms and reads the distance between the car and an object in front of it. Using the sensor reading, the DistanceTask should try to set the driving command to a value that would make the car drive towards the object with a separation distance of around 20 cm. This means that the car should go forward if it is too far away, and backward if it is too close. As an appropriate speed for driving the car, the following strategy is quite elegant. Choose a speed proportional to the difference between the distance the car DOES have and the distance the car SHOULD have. This is also called a "proportional controller" or just "P controller" in control theory. If you have some knowledge about this field and are bored, you may try to implement a full PID controller. Define the priority `PRIO_DIST` used for writing to `dc` as a value that falls between the two already existing priorities.

The task displaying useful information on the screen should be extended to display even more useful information.

Test your program and make sure it has the following behavior. The car should try to keep a distance of 20 cm with respect to the object in front of it. When the touch button is pressed, the car should drive backward for one second, after which it should resume the distance-keeping (resulting in driving closer again).

9 Grand Finale

In the last part of the lab you will create a car that can follow a line that is drawn on the floor (not necessarily a straight one!). You may use any of the techniques you learned to define and schedule tasks, read sensor outputs and send commands to the motors. The line tracking should be done with the light sensor. If you are enrolled in ECE 6780, there are two additional requirements: (1) Do not hard-code the thresholds for the values of the light sensor. Instead, include a simple sensor calibration procedure at the beginning of your program so your car can adapt to different environments. (2) Your car does not have to drive backward, but do not assume a direction on the track. In other words, you car must work both clock- and counter-clockwise.

10 Checkoff

The TA will use the following criteria to grade your work.

10.1 Warmup (50 Points)

Task	Points
Code is well-structured and well-documented	5
The output looks nice	15
The color sensor is used as a light sensor and the readings appear correct	30
Total	50

10.2 Event-Driven Scheduling (100 Points)

Task	Points
Code is well-structured and well-documented	5
The system has the correct functionality	55
All four requirements are met	40
Total	100

10.3 Periodic Scheduling (100 Points)

Task	Points
Code is well-structured and well-documented	5
Mutual exclusion provided for shared resource	20
Car goes backwards when touch button is pressed	25
Car speeds up and slows down as appropriate with respect to object in front of it	30
Display provides useful information	20
Total	100

10.4 Grand Finale (150 Points)

Task	Points
Code is well-structured and well-documented	5
Auto light calibration working	45
Car stays on track	65
Car stays on track both directions (clockwise and counter-clockwise)	35
Total	150