ECE 5780/6780 Lab 3

1 Objective

To give hands-on experience developing a real-time embedded system.

2 Logistics

You are to work in teams of 1 or 2 members. Check out one LEGO Mindstorms kit from the ECE store per team. You and your teammate should be able to independently describe all solutions. This lab consists of 3 different projects with 3 demonstrations and submissions.

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.

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 can be found in the LegoPackage.zip file. Install it on a computer in ENLAB 255 or your personal laptop or PC. 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.

Additional resources are available on Canvas 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 “HelloWorld" 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

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:

- Use the color sensor as a light sensor and read the current light sensor A/D data repeatedly with a delay of 100 ms in between.
- Update the display every 500 ms with:

```
Welcome to My World!
Name: <your name>
Light Sensor: <average value since last update>
Count: <sensor read count since start>
```

Note that your code should include an infinite loop so, technically, TerminateTask() will never be called.

The following functions (and others) may come in 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/, the LEJOS-OSEK website (http://lejos-osek.sourceforge.net/samples.htm). and the nxtOSEK C API reference (http://lejos-osek.sourceforge.net/ecrobot_c_api.htm) for useful examples.

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.

Submission: (1) Demonstrate your working light sensor to the instructor at the specified time and (2) submit your well-structured, well-documented C code on Canvas.

Rubric:

Task	Points
Code is well-structured and well-documented	25
Display looks professional	25
Light sensor is functional	50
Total	**100**

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.

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 using `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.

Submission: (1) Demonstrate your working Lego car to the instructor at the specified time and (2) submit your well-structured, well-documented C and oil files on Canvas.

Rubric:

Task	**Points**
Code is well-structured and well-documented	25
Touch sensor is functional	50
Light sensor is functional	50
Motors are functional	50
Car starts and stops appropriately	25
Total	**200**

8 Competition

In the last part of the lab you will create an autonomous vehicle that follows the course laid out on the floor of ENLAB 255. You will receive points for (1) successfully completing the course and (2) completing with a better time than the other entries. The course rules are as follows:

1. The vehicle shall follow a course marked with black electrical tape
2. Start and Finish lines will be marked with cross-wise strip of red tape. Time starts when any part of the car crosses the Start line. Time ends when the entire vehicle has crossed the Finish line.
3. The course can include straight lines, curves, sharp angles, and crossings
4. Sections of the course may involve "dashed", rather than solid tape.
5. There will be one obstacle blocking the course. The vehicle must detect the obstacle, navigate around or over it, and get back on course.

Submission: (1) Demonstrate your working Lego car to the instructor at the time of the competition and (2) submit your well-structured, well-documented C and oil files on Canvas.

Rubric:

Task	**Points**
Code is well-structured and well-documented	25
Car can follow straight solid line	25
Car can follow dashed line	25
Car can negotiate curves	25
Car can negotiate sharp angles	25
Car can negotiate obstacle	25
Car completes the entire course	100
Time (relative to other cars)	50
Total	**300**