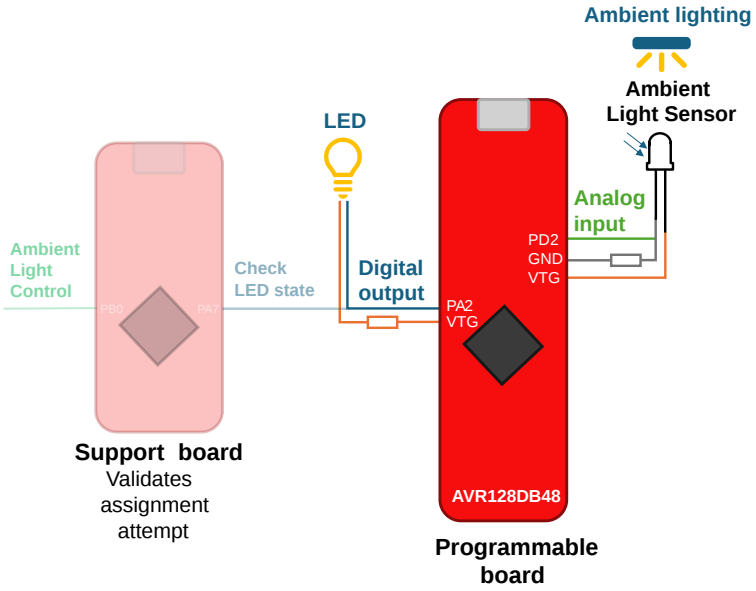
TDT4258 Lab 4 - Microchip Try

Assignment Goal

In this lab assignment, the goal is to create an application that reads the analog input from an analog ambient light sensor and uses the reading to toggle the LED when the ambient light level reaches a certain threshold. The objective is to achieve this while keeping the power consumption as low as possible. That is, when it's dark: turn the LED on, and when it's bright: turn the LED off.



The assignment consists of six tasks to guide you in building the application.

How to get points in Lab 4

In this lab you are trying to minimize energy consumption using different approaches. The different approaches result in vastly different power consumption. Tasks 3-5 implement those different approaches. To be awarded points for those tasks, your solution needs to show better (read: lower) power consumption than the indicated solutions on the leaderboard, which corresponds to the following limits:

- Task 3 busy-waiting: 1.3 mA,
- Task 3 polling: 700 µA,
- Task 4 interrupt-driven: 200 µA,
- Task 5 core independent operation: 150 µA.

The leaderboard numbers stem from a reference implementation, to which we added a bit of margin, such that you should be able to come up with a solution that outperforms the requirement for the respective tasks. For the hand-in on Blackboard, we ask you to submit your implementation of task 5. In addition to the file upload and the AI statement, you'll find some more text questions, where you can answer the questions raised in tasks 3-5.

Bonus

For the top four submissions on the leaderboard in lab 4 we have a small surprise in the last lecture!

About Microchip Try

Microchip Try allows you to remotely control hardware online. We'll be using the Microchip Try framework in this course. There are two modes available: a *Sandbox* for testing and debugging your code, which provides 2 minutes of hardware access and a serial terminal; and *Challenge Mode* for running tests on your code to verify it against the assignment goal and to compete against other participants.

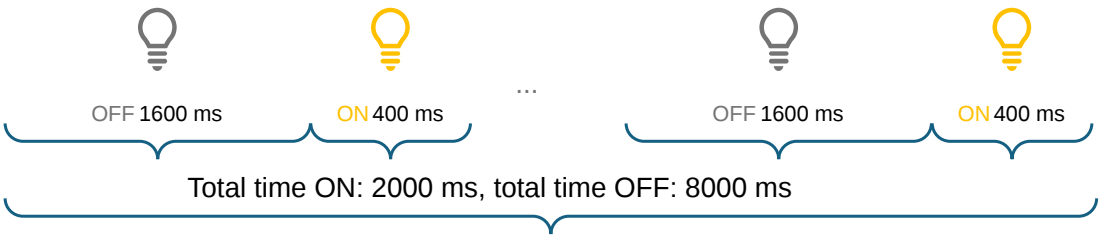
To program the microcontroller with your code, simply upload a hex file to the Microchip Try framework. See the *Preliminaries* tab on how to generate the hex file.

> 🕑 Please note that you'll be running your code on physical hardware, and as such you may experience queuing and varying wait times depending on the number of simultaneous users. If you upload a new hex file while in the queue, it will replace your previous file.

Sandbox

- Drag-and-drop programming
- Test your code
- Access to hardware for 2 minutes at a time
- Access to serial terminal to communicate with the microcontroller
- Adjust the ambient light level manually
- Run a predefined, known test sequence, blinking the ambient lights on and off to test your implementation
- Get a measure of the power consumption used by the microcontroller

To check if your code accomplishes the objective, you upload your program (as a hex file) to Try and then run a test sequence. The ambient lights then turn ON for 400 ms and then OFF for 1600 ms. This repeats five times for a total duration of 10 seconds.



The predefined sequence

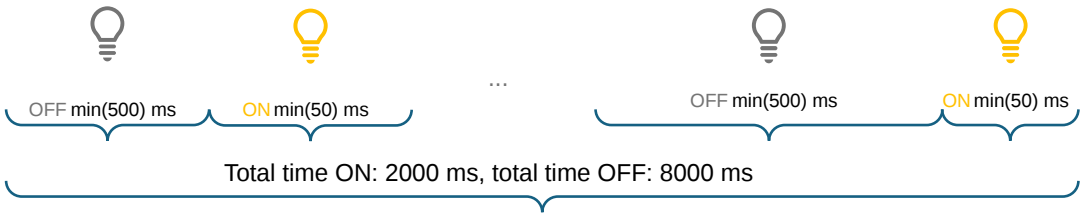
After the sequence has run, you get a response from Microchip Try letting you know whether your attempt was successful or not, i.e. whether you turned the LED on and off at the correct times according to the ambient light levels.

Challenge Mode

In Challenge mode you can test whether your code meets the objective. Specifically, your code should be able to detect whether the ambient lights are ON or OFF, and turn the LED OFF or ON accordingly. The power consumption of the microcontroller is measured as your code runs, with the goal of minimizing power consumption.

To verify your solution, upload your program (as a hex file). Your hex file is placed in a queue. When the hardware becomes available, the microcontroller is programmed with your implementation.

A randomized sequence is then generated with five alternating periods of lights ON and OFF, where the total ON time is 2000 ms (with each on period lasting at least 50 ms) and the total OFF time is 8000 ms (with each off period lasting at least 500 ms).



A randomized sequence

After the sequence has run, you will receive a response from Microchip Try, letting you know whether your attempt was successful. If you successfully turn the LED on and off at the correct times, you have the option to submit your attempt to the leaderboard.

You may try as many times as you like.

Note on Variations Between Hardware

Microcontrollers can exhibit varying power consumption due to differences in design and manufacturing processes. For the leaderboard, all participants use the same hardware instance to enable fairer performance comparisons. However, minor variations may still occur.

The hardware used in the Sandbox and the hardware used in the Challenge section are two separate instances of the same setup. As a result, you may observe variations in power consumption and performance when running the same code on different hardware instances.

Measuring very low currents—such as those drawn by microcontrollers in sleep modes— is inherently difficult. Currents measured in microamperes and less are easily affected by electrical noise, temperature changes, and limitations of measurement equipment. Small inaccuracies or interference can significantly impact the readings, making precise and repeatable measurements technically challenging. If you manage to reduce the power consumption enough to reach this limitation, you have done an excellent job and would need a more precise measurement setup to further refine your implementation.

Preliminaries

For developing the code, there are several options:

- **MPLAB Extensions for VS Code**: The current development environment for Microchip devices.
- **MPLAB X IDE**: The previous development environment for Microchip devices. Still supported, but not recommended.
- **Setting up your own environment**: There are alternatives to the options described above, but these are not covered here.

Using MPLAB Extensions for VS Code

> ▶️ Video guide series on YouTube

> ℹ️ This guide is written with Windows in mind, but there should not any breaking differences on Mac or Linux.
> On Mac, use the same VS Code shortcuts, but replace all Ctrl with Command
> e.g. Windows: *Ctrl+Shift+P*, Mac: *Command+Shift+P*

> 🐧 If you use Linux, I assume you know what you are doing

Installation and Dependencies for VS Code

1. Go to the XC compiler download page and download the MPLAB XC8 C-Compiler for your operating system.
2. Open the installer and follow the prompts. Refer to the XC8 installation guide for details.
   > ⚠️**Note**: Be sure to check 'Add xc8 to the PATH environment variable' in the Compiler Settings step of the setup.
3. Open VS Code.
4. Open the Extensions tab (Ctrl+Shift+X) and search for MPLAB and download "MPLAB".

Getting Started with VS Code

1. Create a **New Project** by either
   - Open the Welcome tab (Command Palette (*Ctrl+Shift+P*) `MPLAB: Welcome` to open) under Get Started with MPLAB, click on Create or Import MPLAB Project, then click Create New Project.
   - Open the Command Palette (*Ctrl+Shift+P*) and type `MPLAB: Create new project`.
2. Choose a project name and location, select `AVR128DB48` as device and `XC8` as the compiler.

Building the Project and Generating a Hex-file with VS Code

In order to run your code on the Microchip Try platform, you have to compile a **hex-file** from your program. MPLAB Extensions does not create hex-files by default, so we have to configure the project to do so.

You can configure you project in either of the following ways:

- Open the project Settings `MPLAB: Edit project properties (UI)`

    - Under Post-build steps, add an Item, paste the following: `avr-objcopy -O ihex %{TargetPath} %{TargetDirRelative}/%{TargetName}.hex` and click OK.

- Open `.vscode/<project-name>.mplab.json` and add the following to your configuration:

```
// .vscode/<project-name>.mplab.json
{
  ...
    "configurations": [
        {
            "name": "default",
            ...
            "postBuildSteps": [
                "avr-objcopy -O ihex %{TargetPath} %{TargetDirRelative}/%{TargetNam
            ]
            ...
        }
    ],
  ...
}
```

> ⚠️ If you did not add xc8 to the PATH environment variable during the compiler setup, this will not work. The simplest solution is to reinstall the compiler according to the earlier instructions. Alternatively, you can manually add the compiler binaries to your PATH variable or replace `avr-objcopy` with the absolute path to the binary.

You can now build your project in either of the following ways:

- Opening the Run Build Task menu (*Ctrl+Shift+B*) and selecting `project-name: default - Full Build`.
- Open the Command Palette (*Ctrl+Shift+P*) and typing `MPLAB CMake: Full Build`
- Click the build icon () in the upper right corner.

Confirm that you have (at least) the following files:

```
.
├── out
│   └── <project-name>
│       ├── default.elf
│       └── default.hex <-- this is the one you upload to Microchip Try
├── .vscode/
│   ├── <project-name>.mplab.json
│   └── settings.json
└── main.c
```

If you wish to do a clean build, open the Command Palette (*Ctrl+Shift+P*) and type `MPLAB CMake: Clean Build`.

Bug Reports and Feedback

We always strive to be better. Please report any bugs or suggestions for enhancements.

> 💡 If you experience conflicts with other extensions, try creating a profile in VS Code for the MPLAB Extensions.

Using MPLAB X IDE

How to download the IDE and create a project

1. Download the MPLAB X IDE. For Linux and Mac, follow the installation walkthrough for MPLAB X.
2. Open the downloaded installer.
3. Follow the prompts until the "Select Application" page. Here you only need to select `MPLAB X IDE (Integrated Development Environment)` and `8-bit MCUs`.
4. When the install is complete, check the box `XC compiler are not installed with the IDE [...]` or go directly to the download page for the XC8 compiler.
5. Download the MPLAB XC8 C-Compiler for your operating system.
6. Open the installer and follow the prompts. Refer to the XC8 installation guide for details.
7. Open MPLAB X and follow the instructions on how to create a new project. Select `AVR128DB48` as device and `XC8` as compiler.

How to create and add files to your project

Learn how to create and add files to your project.

How to build the project and generate the hex-file

To upload your code to Microchip Try, you'll need a hex-file of your program.

1. How to generate a production hex-file instructions at Microchip Developer Help.
2. Open your MPLAB X project folder in File Explorer, likely at `C:\Users\your-username\MPLABXProjects\<project-folder>` (for Windows users).
3. Locate your .hex-file in the folder `<project-folder>\dist\default\production\`.

Set up USART

Download `usart.h` and `usart.c` from this task description. The files implement a USART driver which may be useful while developing your code using the Microchip Try framework as we have provided you with a serial terminal.

Add the files to your project. Refer to the *Preliminaries* tab on the left on how to do this in your development environment.

Modify main.c

Modify your main.c file to include the usart.h file, by adding the following line at the top:

```
#include "usart.h"
```

In the `main()` function, test the driver by calling `USART3_Init()` to initialize the USART peripheral and `USART3_SendChar('A')` to send a character.

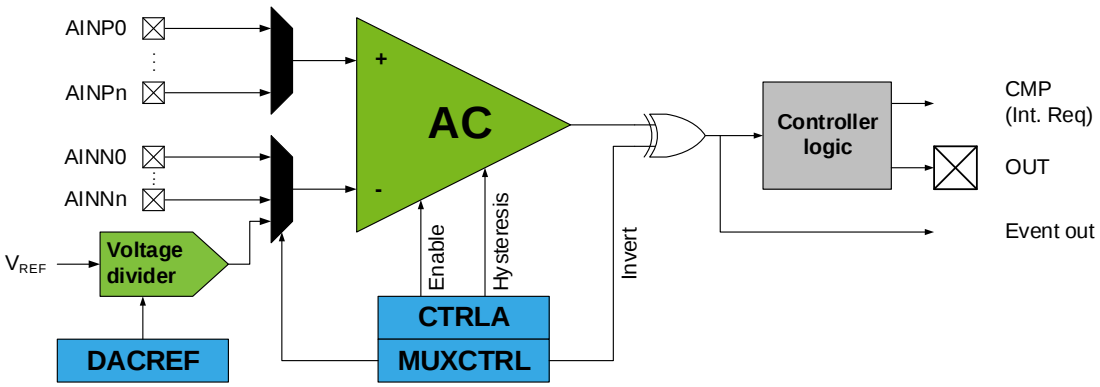
If you wish to, you can include the `<util/delay.h>` library in your main.c-file and use `_delay_ms()` to send a UART message every 1 second.

> ⚠️ The default clock frequency of the microcontroller is 4 MHz. This value is defined by `#define F_CPU 4000000UL` in `usart.h` and is used by the `<util/delay.h>` library and to calculate the baud rate of the USART communication. If you change the clock frequency of the microcontroller, remember to update the `F_CPU` value.

Generate the hex-file and upload your program to the sandbox in Microchip Try to run your code on the Curiosity Nano hardware.

Initialize the Analog Comparator

Read chapters 32.1 through 32.3 of the datasheet to get an overview of the Analog Comparator. Next, follow the initialization steps in chapter 32.3.1 to set up the peripheral, AC0. You'll need the table in chapter 3.1 to identify the correct positive input signal (`AINPn`) for pin PD2. The negative input signal should be set to 'DACREF'.



Step one, configuring the desired pin as an analog input, is provided below for the pin connected to the ambient light sensor:

```
void AC_init(){
    // Set pin PD2 (port D, pin 2) as an input
    PORTD.DIRCLR = PIN2_bm;
    // Disable digital input buffer and pull-up resistor for pin PD2
    PORTD.PIN2CTRL = PORT_ISC_INPUT_DISABLE_gc;
    // Remaining initialization steps...
}
```

The next steps in the initialization use the registers of the Analog Comparator. If you find these steps challenging, a pseudocode project with comments to guide you are provided.

> 📖 Reading the attached link to *Writing C-code for AVR MCUs* is highly recommended to get you started. Quickguide for initializing AVR peripherals

Setting voltage reference

To set the voltage reference of the analog comparator, add the following function to your project:

```
void VREF_init(void) {
    VREF.ACREF = VREF_REFSEL_1V024_gc;
}
```

The function sets the voltage reference of the analog comparator (AC) to 1.024 V, as described in Chapter 21.3.1 of the datasheet which explains the initialization steps of the Voltage Reference (VREF) peripheral.

Remember to call this function in your `main()` function.

Set the threshold value for AC to activate

The analog comparator compares two analog input voltages and outputs a signal level indicating when one of the inputs is higher than the other.

The hardware for this course is set up such that the positive input of the AC is connected to the ambient light sensor on pin PD2. For the negative input we'll use an internal reference (DACREF). The internal reference value is determined by the threshold for what is considered 'on' and 'off' for the ambient light levels. In this course, we'll define 0.1V as this threshold. Meaning, when the analog signal from the ambient light level sensor is below 0.1V, the AC output is low/'0' and the LED should be ON, and the LED should be OFF when the analog signal is above 0.1V and the AC output is high/'1'.

Use the equation below from 32.3.2.2 in the datasheet to calculate the DACREF value and add the following line to your `AC_init()` function: `AC0.DACREF = <calculated_DACREF>`. V_{REF} is the voltage reference of the AC set in the previous task, V_{DACREF} is the voltage level threshold discussed above, and the DACREF value should be a number between 0-255.

$$V_{DACREF} = \frac{DACREF}{256} \cdot V_{REF}$$

Reading the status of the Analog Comparator

Implement a function that checks if the signal from the analog sensor is above the set threshold. See Chapter 32.5.6 of the datasheet.

> 💡 Check the `CMPSTATE` bit field of the `STATUS` register

```
if(AC0.STATUS & AC_CMPSTATE_bm) {
  // ...
}
```

Setting up LED

Paste the following functions that turn and off the LED connected to pin PA2 into your main file.

```
void LED_init() {
    PORTA.DIRSET = PIN2_bm;
}
void set_LED_on(){
    // LED is active low. Set pin LOW to turn LED on
    PORTA.OUTCLR = PIN2_bm;
}
void set_LED_off(){
    // LED is active low. Set pin HIGH to turn LED off
    PORTA.OUTSET = PIN2_bm;
}
```

Remember to call the `LED_init()` function in your `main()`-function. You may remove the `USART3_Init()` function and other function calls to the USART peripheral if you don't use these for testing and debugging. If you remove the `#include "usart.h"` line, make sure that any necessary includes from `usart.h` are added to your main file instead.

Test turning ON and OFF the LED with the function you implemented in the previous task: When the signal from the analog sensor is above the set threshold turn the LED off, and turn the LED on when the signal is below the threshold.

Generate the hex-file and upload your program to the sandbox in Microchip Try.

Busy-waiting

Implement a busy-waiting scheme continuously checking if the ambient light sensor data is above the set threshold. When the sensor data is above the threshold (i.e. the sensor data indicates a low light environment) turn the LED ON. Otherwise, the LED should be OFF.

The structure of your main file should look something like this:

```
// #include ...
// #define ...

void AC_init() {
    // ...
}
void VREF_init() {
    // ...
}

void LED_init() {
```

```
    // ...
}

void set_LED_on() {
    // ...
}

void set_LED_off() {
    // ...
}

bool AC_above_threshold() {
    // Check the output of the Analog Comparator
}

int main() {
    AC_init();
    VREF_init();
    LED_init();

    while(1) {
        // Implement the busy-waiting scheme
    }

    return 0;
}
```

> 💡 Ensure unused digital I/O pins are not left floating. Enable internal pull-up resistors and disable the digital input buffer to minimize leakage and reduce power consumption. Refer to Chapters 18.3.2.3 and 18.3.2.4 in the datasheet for Pin Configuration and Multi-Pin Configuration.
>
> ```
> PORTx.PINCONFIG = PORT_ISC_INPUT_DISABLE_gc | PORT_PULLUPEN_bm;
> PORTx.PINCTRLUPD = 0xFF;
> ```

Generate the hex file and upload your program to the Challenge section in Microchip Try. Save the power consumption graph as an image for easy comparison.

Polling

Implement a polling scheme that periodically checks if the ambient light sensor data is above the set threshold. Put the device in sleep mode between the periodic checks. A timer driver using Timer/Counter A (TCA) is implemented for you with an interrupt service routine (ISR). The functions needed to put the device to sleep are also provided.

In this driver, the timer is configured to wake the device every 10 ms = 0.01 s. You can adjust this by updating the value of the `TCA0.SINGLE.PER` register according to the following formula:

$$T = \mathrm{PER} \cdot \frac{\mathrm{DIVn}}{f_{clk}} = \mathrm{PER} \cdot \frac{2}{4000000}$$

where T is the period in seconds, PER is the value stored in the register, DIVn is the prescaler value, and f_{clk} is the microcontroller clock frequency.

> ⚠️ Note that this formula assumes the clock frequency of the microcontroller is 4 MHz, if you have changed this, you'll need to update the formula accordingly.

`PER` is a 16-bit value that can be set to any value from 0 to 2^{16}. The prescaler is set to 2 in the driver, but it can be adjusted to any of the values specified in Chapter 23.5.1.

This is the relevant function from the driver to set up the timer. Note that we must explicitly enable standby mode (`TAC0.SINGLE.CTRLA` |= `TCA_SINGLE_RUNSTDBY_bm`) to keep it running while the CPU sleeps.

```
void TCA0_init() {
    // Set the period of the timer.
    // PER = period[s] * F_CPU / Prescaler = 0.01s * 4 000 000 Hz / 2
    TCA0.SINGLE.PER = 20000;
    // Enable timer overflow interrupt
    TCA0.SINGLE.INTCTRL = TCA_SINGLE_OVF_bm;
    // Run timer in standby mode, set prescaler to 2, enable timer
    TCA0.SINGLE.CTRLA = TCA_SINGLE_RUNSTDBY_bm | TCA_SINGLE_CLKSEL_DIV2_gc | TCA_SINGLE
}
```

Generate the hex file and upload your program to the Challenge section in Microchip Try. Save the power consumption graph as an image for easy comparison.

> 💡 A timer is really just a counter that increments by 1 every time it is triggered. We ask it to count to a given "goal" number at a particular pace, and give us a signal when it does so. How quickly it reaches this "goal" depends on how big that number is and how fast it counts. If we want a longer period, we have to ask it to count to a bigger number and/or ask it to count slower.
>
> The "goal" number is the period, `PER`. To increase the period, we can increase this number, but not beyond 2^{16}, as it is a 16-bit timer.
>
> How fast the timer counts depends on the microcontroller clock frequency, f_{clk}. If we want it to count slower, we can decrease the clock frequency, but that will also decrease the speed of everything else (which may or may not be desirable).
>
> We can also ask the timer to count only every *i*-th cycle. This is what the prescaler (sometimes referred to as clock divider) effectively does. For example, if we use a prescaler of 2 (like in the provided driver), the timer counts every 2 cycles and effectively runs at 2 MHz instead of 4 MHz —hence the name "clock divider".

Compare results

Review and compare the power consumption and discuss advantages and disadvantages of busy-waiting and polling. Please write your answer in the Submission Form on Blackboard.

Interrupt-Driven Approach

Update your initialization function for the analog comparator to enable interrupts when the analog sensor input goes above or below the threshold value. Make sure the AC is set to operate in standby mode. Then, implement the Interrupt Service Routine (ISR) for the AC. Use the ISR from the previous task as a guide. The interrupt vector for the comparator is `AC0_AC_vect`, and remember to clear the interrupt flag `AC_CMPIF_bm`. Since the analog comparator (AC) will now generate interrupts, you can remove the timer-interrupt from the polling scheme as it is no longer necessary.

See Chapter 32.3.4 about the available AC interrupt.

The LED should have the same behavior as in the previous task and the microcontroller should be set to sleep after the ISR runs.

Generate the hex file and upload your program to the Challenge section in Microchip Try.

Compare results

Review and compare the power consumption with those from the previous tasks. Identify advantages and disadvantages of using the interrupt-driven approach. Please write your answer in the Submission Form on Blackboard.

Core Independent Operation

Update your implementation to fulfill the requirements of the application without any intervention from the CPU by using the Event System. Ensure that the infinite while loop in your `main()` function is empty and that the solution does not use interrupts.

Refer to chapter 16 of the datasheet about the Event System. The Event Generator is the analog comparator and the event user is the pin connected to the LED. Remember to put the device to sleep.

Generate the hex-file and upload your program to the Challenge section in Microchip Try. Save the power consumption graph as an image for easy comparison.

Compare results

Review and compare your results with those from the last three exercises. Identify scenarios where an Event System is more effective than an interrupt-driven approach, and vice versa.

Lowering Power Consumption (Optional)

The final task is to lower the power consumption as much as possible. Use the methods from the datasheet, the lecture on low power techniques this Application Note, and other methods you can find to lower the power consumption even further.

If you wish to submit your attempt to the leaderboard, use the Challenge section in Microchip Try.

We'd really like your feedback!

If you have any feedback on the Microchip Try system, the assignment, the toolchain or anything else, please submit your feedback.

Support

If you have any questions, please use Piazza and the lab sessions.

If there are technical problems with the toolchain, you may contact the team at Microchip through the email address posted on Blackboard. Please note that replies should not be expected outside working hours.