Programming micro-controllers - Arduino

Workshop on managing complex projects

Robert Benkoczi, C556
robert.benkoczi@uleth.ca

21-Jan-2024
U of Lethbridge - CS Club

Objectives

- Using the resources freely available online, the participants will develop a programming framework for an Arduino micro-controller that allows them to integrate complex behaviour.
- The participants will implement basic components of real-time programming projects (time constraints on processing).

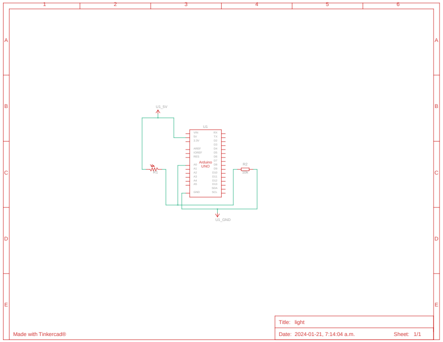
Prerequisites

- The participants know basic Arduino / C++ programming. They can write a basic Arduino program and load it.
- The participants are familiar with Object Oriented Programming techniques (classes, interfaces, abstract classes, and virtual functions).
- The participants know basic notions of electronic circuits (voltage, current, ground and 5V pins, resistive light sensors), and are familiar with the role of the pins of an Arduino micro-controller.

Micro-controllers

- Micro-controllers *control* the operation of various electro-mechanical systems (lights, motors, relays, etc.)
- Arduino = general purpose, programmable micro-controller.
- A micro-controller reads signals from sensors and generates signals to actuators (electro-mechanical systems)

The demo project for this workshop



- Input signals: resistive light sensor connected to an analog pin.
- Output signals: send the reading to the serial monitor.
- We will imagine that the complexity of the I/O increases and we will develop various software design techniques to deal with the increased complexity.

Basic structure of an Arduino program, v0.0

The setup

```
// general configuration parameters
// where is the light sensor connected
#define LS_PIN 0
// Speed of the serial interface
#define S_MON_SPEED 9600

void setup() {
  // put your setup code here, to run once:
  Serial.begin(S_MON_SPEED);
}
```

- Parameterize your hardware setup (named constants for input/output pins).

Basic structure of an Arduino program, v0.0 (c'ed)

The brain

```
void loop() {
  // put your main code here, to run repeatedly:
  // data / variables representing the state of the system controlled
  int ambient_light=0;

  // read input signals
  ambient_light = analogRead(LS_PIN);

  // generate output signals
  Serial.println(ambient_light);

  // possibly controll the "sampling rate"
  delay(500);
}
```

- Separate responsibilities according to the signals processed: read light sensor, write reading on separate lines.
- Maintain the state of the system: `int ambient_light`.
- Real-time programming I: process signals every x seconds.
  Only one `delay` statement / project, in the loop function.

Increase complexity in processing signals, v1.0

- Assign signals to classes.
- Decide how to represent system state: dedicated class/struct, or distributed among signal classes (I prefer a dedicated class).
- Simply create .h and .cpp (! not .cc) files in the project directory.
- Provide implementation in .cpp files. Include `#include <Arduino.h>` if using Arduino specific statements, such as `analogRead()`.

Example

- Signals processed by `MyLight` and `MyOutput` classes.

```
// object to processing input and maintain state of sensor
MyLight light;
// object processing the output
// pass a ptr to the light sensor object that maintains state in constructo
MyOutput output(&light);

void loop() {
  // put your main code here, to run repeatedly:
  // read input signals
  light.loop();
  output.loop();

  // possibly controll the "sampling rate"
  delay(500);
}
```

Example

- MyLight defined in light.h and light.cpp (also find better names, like MyLight.h, ...).

```
class MyLight {
 private:
  // state maintained by MyLight
  // (separate class to maintain state preferable)
  int ambient_light=0;

 public:
  // call to process signal
  void loop();

  // RET
  // reading from light sensor: 0..1023
  int getLight();
};
```

Example

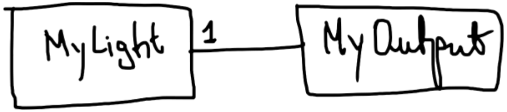
- and `MyOutput` in `out.h` and `out.cpp`.

```
class MyOutput {
 private:
  MyLight * light_sensor_ptr;
 public:
  // pass a pointer to the light sensor object whose data will be used
  MyOutput(const MyLight*);

  // call to process output signal
  void loop();
};
```

Notes

- The *loop()* method of the *signal* classes should terminate as quickly as possible. Do not `delay()` in the class member functions.
- Here is a UML class diagram. `https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/`

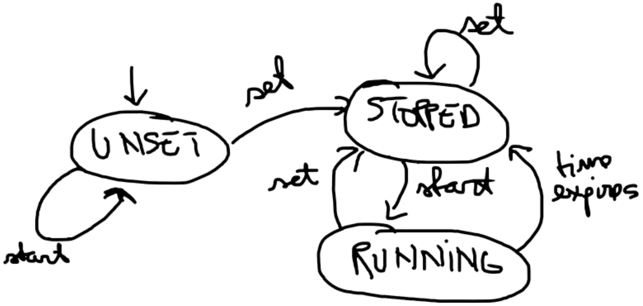


Exercises

1. Modify the light sensor class to configure it in the constructor with the analog pin to read data from.
2. Add a second light sensor to the project. Modify the code so that both sensors are read (hint: add a second MyLight object) and the output displays the pair of values read from the two sensors.
3. Change the design so that the state of the sensor(s) are maintained by a separate class. Design a suitable interface for this class so that it can be useful for both MyLight and MyOutput classes.

More complex *real-time* requirements; v1.1

- Often we have different real-time requirements for each signal. How do we execute computations with different real-time deadlines?
- Use timer classes, not `delay()`!
- A simple timer interface: set, start, checkState. Timer states: UNSET, RUNNING, STOPPED.



Example, v1.1

Suppose we want to read the light sensor every 100 ms, but output the average value every 2115 ms. What shall we do?

Example, v1.1

Suppose we want to read the light sensor every 100 ms, but output the average value every 2115 ms. What shall we do?

1. Add a new class to handle the state of the system, eg. the average value from the light sensor.

```
// comment this line to remove the debug messages
// #define __DEBUG

class MySystem {
 private:
  long int _sum;  // the sum of samples collected
  int _nr;  // number of samples collected

 public:
  MySystem();

  // adds sample to the sum of samples and counts the sample
  // Note: will overflow if too many samples!
  void addSample(int sample);
```

Example, v1.1

Suppose we want to read the light sensor every 100 ms, but output the average value every 2115 ms. What shall we do?

2. Connect the light sensor object and the output object to the `MySystem` object that now maintains the state of the system (adjust the constructors of `MyLight` and `MyOutput`, of course).

   In `light_1.1.ino`:

   ```
   #include "out.h"

   // object to maintain system state
   MySystem the_system;
   // object to processing input and maintain state of sensor;
   // pass the addr of the system object that maintains system state
   MyLight light(&the_system);
   // object processing the output
   ```

Example, v1.1

Suppose we want to read the light sensor every 100 ms, but output the average value every 2115 ms. What shall we do?

3. Process each of the signals according to the specified time deadlines.

```
void MyLight::loop() {
  if (sys_ptr != nullptr) {
    if (my_time.checkState() == MyTimer::STOPPED) {
      sys_ptr->addSample(analogRead(LS_PIN));
      my_time.start();  // don't forget to restart timer!
    }
  }
}

void MyOutput::loop() {
  // generate output signals
  if (sys_ptr != nullptr) {
    if (my_time.checkState() == MyTimer::STOPPED) {
      Serial.println(sys_ptr->getAverage());
      my_time.start();  // don't forget to start the timer
    }
  }
}
```

Example, v1.1

Suppose we want to read the light sensor every 100 ms, but output the average value every 2115 ms. What shall we do?

4. Don't forget to adjust the global (and unique) call to `delay()` in the topmost `loop()` function (its presence may save some battery) or remove it altogether.

Exercises

1. Incorporate the extensible light sensor object from the previous exercise (the one whose analog pin can be passed as a parameter) within framework v1.1.
2. Add a push button to start and stop the device. When stopped, the device does not read the light sensor, nor does it send data to the serial monitor. Use the builtin LED linked to digital pin 13 to indicate the state of the device (on or off).
   NOTE: timers are very useful to reliably process button presses!
3. Add the motor shield to the circuit and integrate your robot chassis. Start prototyping your line following solution.
   NOTE: compile and run often. Add features in **small** increments.

Extensions

Search among the many of the Arduino libraries available for components that can be used in your project. Examples:

- Library implementing co-routines, *Ace-Routine*.
  `https://www.arduino.cc/reference/en/libraries/aceroutine/`
  Use as an alternative to the signal classes demonstrated in this tutorial (note: when memory use is close to full, I noticed interference with String and JSon Arduino classes).
- Timers: arduino-timer (not tested)
  `https://github.com/contrem/arduino-timer`.
  Can define call-back functions to be executed at regular time intervals.
- Data structures: linked lists (not tested)
  `https://nkaaf.github.io/Arduino-List/html/index.html`
- Unit testing frameworks are available (not tested):
  - arduinounit
    `https://github.com/mmurdoch/arduinounit`
  - AUnit
    `https://github.com/bxparks/AUnit`