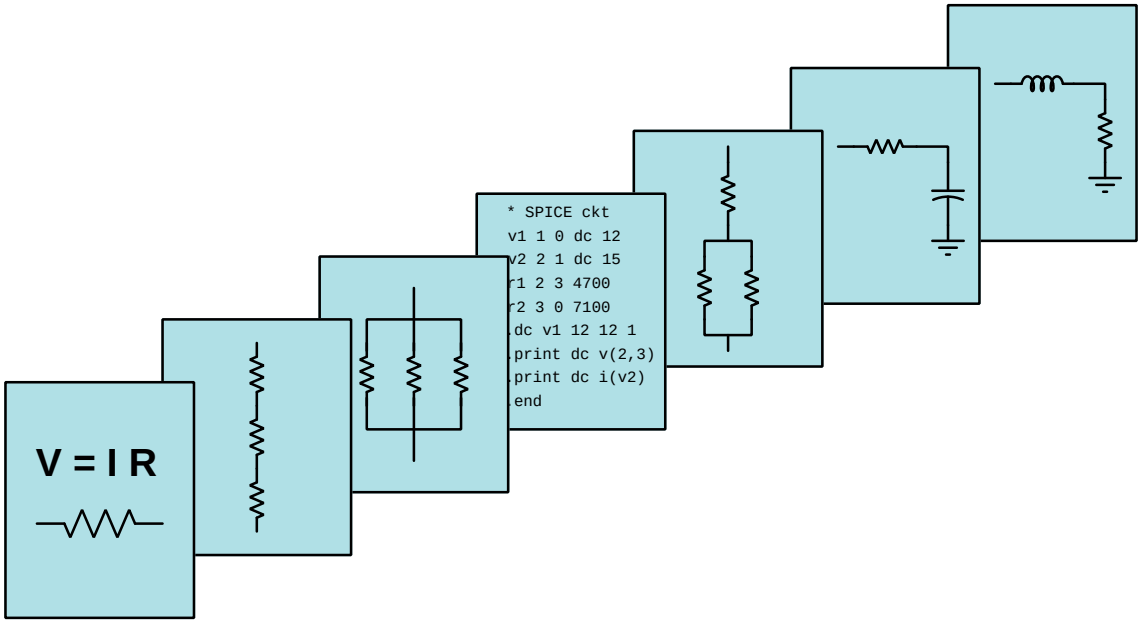
Modular Electronics Learning (ModEL) project



Programmable Control Devices

Last update = 9 November 2024

Contents

Chapter 1

Introduction

1.1 Recommendations for students

Microcontrollers and Programmable Logic Controllers (PLCs) are two different types of digital computer designed for the similar purpose of real-time control.

Important concepts related to both microcontrollers and PLCs include **programming**, **code**, **discrete** versus **analog** signals, **I/O**, **comments**, **execution order**, **ladder diagrams**, **AND** versus **OR** versus **NOT** logic functions, **open** versus **closed** contacts, **series** versus **parallel** connections, **high** versus **low** logic states, and **conditional** instructions.

Here are some good questions to ask of yourself while studying this subject:

- How may we identify the binary bit values for a counter circuit, from its timing diagram?
- How do microcontrollers differ from PLCs in form?
- How do microcontrollers differ from PLCs in function?
- How do microcontrollers differ from PLCs in terms of programming languages?
- What is the "normal" status of a switch contact?
- What are some of the different types of timer functions?
- How do we implement AND versus OR logic using wired relay contacts?
- What is a *routine* in a computer program, also known as a *subroutine* or a *function*?
- What is the purpose of a *pullup resistor*?
- What is the purpose of a *pulldown resistor*?
- What is *pseudocode*, and how does it differ from a standardized programming language?

1.2 Challenging concepts related to programmable devices

The following list cites concepts related to this module's topic that are easily misunderstood, along with suggestions for properly understanding them:

- **Normal status of a switch** – to say that a switch is "normally open" or "normally closed" refers to its electrical state *when at rest*, and not necessarily the state you will typically find the switch in. The root of the confusion here is the word *normal*, which most people take to mean "typical" or "ordinary". In the case of switches, though, it refers to the zero-stimulation status of the switch as it has been manufactured. In the case of PLCs it refers to the status of a "virtual switch" when its controlling bit is 0.

1.3 Recommendations for instructors

This section lists realistic student learning outcomes supported by the content of the module as well as suggested means of assessing (measuring) student learning. The outcomes state what learners should be able to do, and the assessments are specific challenges to prove students have learned.

- **Outcome** – Demonstrate effective technical reading and writing

  Assessment – Students present their outlines of this module's instructional chapters (e.g. Case Tutorial, Tutorial, Historical References, etc.) ideally as an entry to a larger Journal document chronicling their learning. These outlines should exhibit good-faith effort at summarizing major concepts explained in the text.

  Assessment – Students explain how each of the pseudocode programs execute in the Tutorial chapter's examples.

- **Outcome** – Apply Boolean logic to the writing of real controller programs

  Assessment – Write code for a microcontroller to implement AND, OR, NOT, NAND, and NOR functions.

  Assessment – Write code for a PLC to implement AND, OR, NOT, NAND, and NOR functions.

- **Outcome** – Independent research

  Assessment – Locate microcontroller datasheets, user manuals, and/or programming reference books and properly interpret some of the information contained in those documents including number of I/O pins, clock frequency(ies), maximum output current ratings, programming languages supported, special built-in functions (e.g. UART serial communication, timers, event counters), etc.

  Assessment – Locate PLC I/O module datasheets and properly interpret some of the information contained in those documents including number of I/O channels, voltage and current limitations, sourcing versus sinking capability, etc.

Chapter 2

Case Tutorial

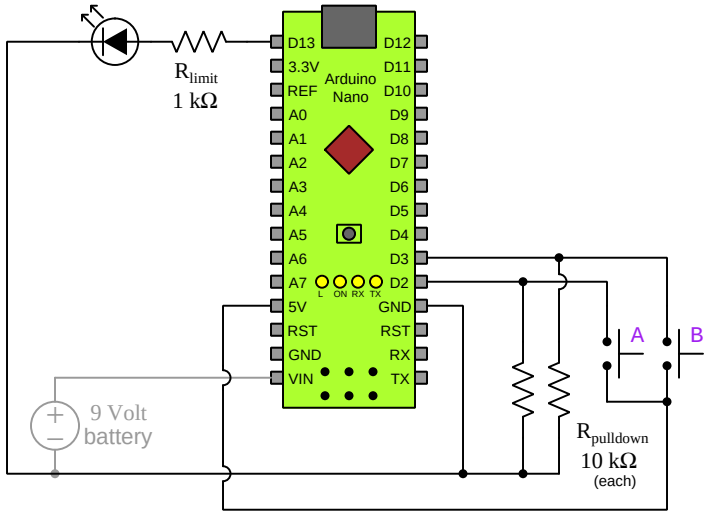
The idea behind a *Case Tutorial* is to explore new concepts by way of example. In this chapter you will read less presentation of theory compared to other Tutorial chapters, but by close observation and comparison of the given examples be able to discern patterns and principles much the same way as a scientific experimenter. Hopefully you will find these cases illuminating, and a good supplement to text-based tutorials.

These examples also serve well as challenges following your reading of the other Tutorial(s) in this module – can you explain *why* the circuits behave as they do?

2.1 Example: Arduino 2-input AND logic function

This example was tested on an Arduino Nano containing the Atmel M328P microcontroller IC:

Schematic diagram



Sketch listing

```
int A, B;

void setup() {
  pinMode(2, INPUT);
  pinMode(3, INPUT);
  pinMode(13, OUTPUT);
}

void loop() {
  A = digitalRead(2);
  B = digitalRead(3);

  // Implementing an AND function
  if (A == HIGH && B == HIGH)
    digitalWrite(13, HIGH);

  else
    digitalWrite(13, LOW);
}
```

Functional behavior – one LED representing the output of an AND logic function.

- $R_{pulldown}$ provides definite "low" logic states to input pins `D2` and `D3` when their respective switches are open. If not for these resistors, the logical state of each input would be uncertain when its switch was open because it would be electrically "floating". The resistance value used here may be quite large, as its only purpose is to quickly drain off any stored electrical charge from the gates of the microcontroller's internal MOSFET transistors when the switch opens. Given the trivial amount of capacitance existing between the gate terminal and channel of a typical MOSFET (typically measured in *pico*-Farads), even a pulldown resistor value in the hundreds of kilo-Ohms results in a very rapid time constant ($\tau = RC$) sufficient for this manual pushbutton application.

- R_{limit} serves to limit the amount of current sunk by each LED when energized. Using a 1 kΩ resistor limits current well below the 20 mA rating for output pins on the Arduino Nano.

- The line `int A, B;` line at the very top of the Sketch program declares two *integer variables*, which are simply dedicated spaces in the microcontroller's memory where whole-numbers values will be stored, one of those spaces referred to by the name `A` and the other by the name `B`.

- The `pinMode()` instructions set the "direction" of the three I/O pins used for this application, pins `D2` and `D3` configured as inputs and pin `D13` as an output. By locating these instructions within the `setup` portion of the Sketch program, they are only executed once upon power-up.

- The `loop` portion of the Sketch program executes over and over again, just as the name "loop" implies. Looping is important for our program to repeatedly check the inputs for any new states and update the output accordingly. If not for looping, the program would not be responsive to any new logical states applied to the input pins, but would only check them at one moment in time and then never again!

- Each `digitalRead()` instruction reads the digital logic state of the specified pin. By setting `A` equal to the scanned state of pin `D2` and `B` equal to the scanned state of pin `D3` we have two variables in the microcontroller's memory which may be easily referenced for making decisions.

- The `if()` instruction checks for a condition of both `A` *and*[1] `B` being "high" which simply means numerical values of 1 each. If this condition is met, the `digitalWrite()` instruction then sets pin `D13` to a "high" digital logic state to turn on the LED as an AND logic gate would if both its inputs were "high".

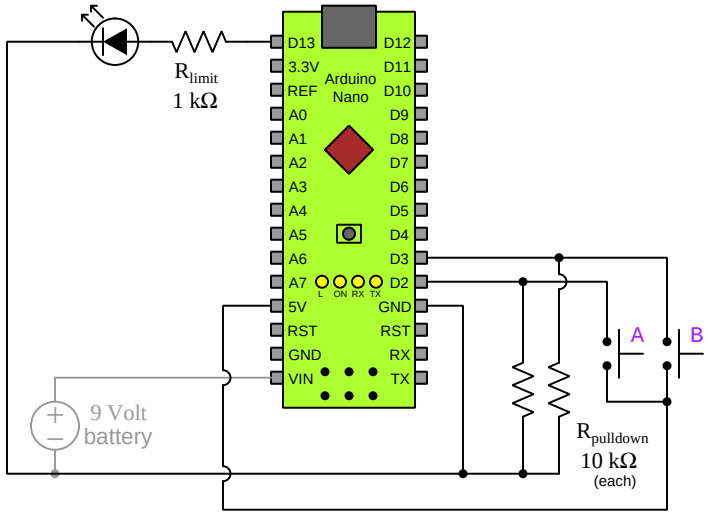
- The `else` instruction provides default action in the event that the `if()` instruction's conditions are *not* met. In this program, for any condition where both `A` and `B` are not both "high" we default to pin `D13` being cleared to a "low" digital logic state, just as the output of an AND gate would go "low" for any condition other than both its inputs being "high".

[1] The `&&` operator specifies a logical AND relationship in the Sketch programming language.

2.2 Example: Arduino 2-input OR logic function

This example was tested on an Arduino Nano containing the Atmel M328P microcontroller IC:

Schematic diagram



Sketch listing

```
int A, B;

void setup() {
  pinMode(2, INPUT);
  pinMode(3, INPUT);
  pinMode(13, OUTPUT);
}

void loop() {
  A = digitalRead(2);
  B = digitalRead(3);

  // Implementing an OR function
  if (A == HIGH || B == HIGH)
    digitalWrite(13, HIGH);

  else
    digitalWrite(13, LOW);
}
```

Functional behavior – one LED representing the output of an OR logic function.

- $R_{pulldown}$ provides definite "low" logic states to input pins `D2` and `D3` when their respective switches are open. If not for these resistors, the logical state of each input would be uncertain when its switch was open because it would be electrically "floating". The resistance value used here may be quite large, as its only purpose is to quickly drain off any stored electrical charge from the gates of the microcontroller's internal MOSFET transistors when the switch opens. Given the trivial amount of capacitance existing between the gate terminal and channel of a typical MOSFET (typically measured in *pico*-Farads), even a pulldown resistor value in the hundreds of kilo-Ohms results in a very rapid time constant ($\tau = RC$) sufficient for this manual pushbutton application.

- R_{limit} serves to limit the amount of current sunk by each LED when energized. Using a 1 kΩ resistor limits current well below the 20 mA rating for output pins on the Arduino Nano.

- The line `int A, B;` line at the very top of the Sketch program declares two *integer variables*, which are simply dedicated spaces in the microcontroller's memory where whole-numbers values will be stored, one of those spaces referred to by the name `A` and the other by the name `B`.

- The `pinMode()` instructions set the "direction" of the three I/O pins used for this application, pins `D2` and `D3` configured as inputs and pin `D13` as an output. By locating these instructions within the `setup` portion of the Sketch program, they are only executed once upon power-up.

- The `loop` portion of the Sketch program executes over and over again, just as the name "loop" implies. Looping is important for our program to repeatedly check the inputs for any new states and update the output accordingly. If not for looping, the program would not be responsive to any new logical states applied to the input pins, but would only check them at one moment in time and then never again!

- Each `digitalRead()` instruction reads the digital logic state of the specified pin. By setting `A` equal to the scanned state of pin `D2` and `B` equal to the scanned state of pin `D3` we have two variables in the microcontroller's memory which may be easily referenced for making decisions.

- The `if()` instruction checks for a condition of either `A` *or*[2] `B` being "high" which simply means numerical values of 1. If this condition is met, the `digitalWrite()` instruction then sets pin `D13` to a "high" digital logic state to turn on the LED as an OR logic gate would if either of its inputs were "high".

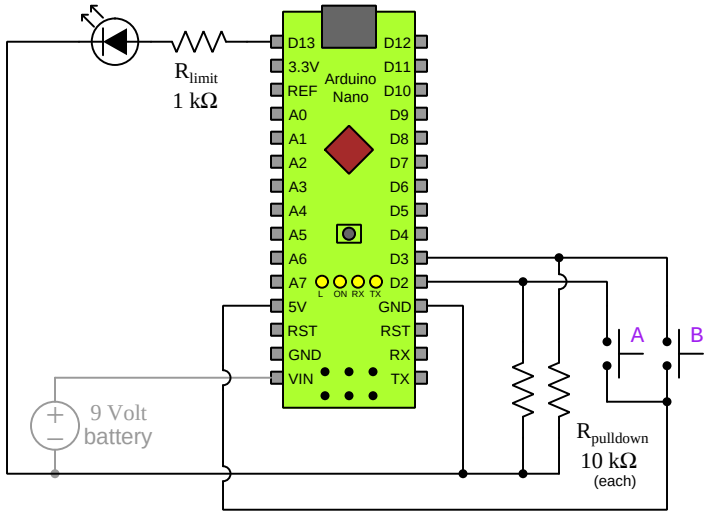
- The `else` instruction provides default action in the event that the `if()` instruction's conditions are *not* met. In this program, for any condition where neither `A` nor `B` are "high" we default to pin `D13` being cleared to a "low" digital logic state, just as the output of an OR gate would go "low" if neither of its inputs were "high".

[2]The `||` operator specifies a logical OR relationship in the Sketch programming language.

2.3 Example: Arduino 2-input XOR logic function

This example was tested on an Arduino Nano containing the Atmel M328P microcontroller IC:

Schematic diagram



Sketch listing

```
int A, B;

void setup() {
  pinMode(2, INPUT);
  pinMode(3, INPUT);
  pinMode(13, OUTPUT);
}

void loop() {
  A = digitalRead(2);
  B = digitalRead(3);

  // Implementing an XOR function
  if (A != B)
    digitalWrite(13, HIGH);

  else
    digitalWrite(13, LOW);
}
```

Functional behavior – one LED representing the output of an Exclusive-OR logic function.

- $R_{pulldown}$ provides definite "low" logic states to input pins `D2` and `D3` when their respective switches are open. If not for these resistors, the logical state of each input would be uncertain when its switch was open because it would be electrically "floating". The resistance value used here may be quite large, as its only purpose is to quickly drain off any stored electrical charge from the gates of the microcontroller's internal MOSFET transistors when the switch opens. Given the trivial amount of capacitance existing between the gate terminal and channel of a typical MOSFET (typically measured in *pico*-Farads), even a pulldown resistor value in the hundreds of kilo-Ohms results in a very rapid time constant ($\tau = RC$) sufficient for this manual pushbutton application.

- R_{limit} serves to limit the amount of current sunk by each LED when energized. Using a 1 kΩ resistor limits current well below the 20 mA rating for output pins on the Arduino Nano.

- The line `int A, B;` line at the very top of the Sketch program declares two *integer variables*, which are simply dedicated spaces in the microcontroller's memory where whole-numbers values will be stored, one of those spaces referred to by the name `A` and the other by the name `B`.

- The `pinMode()` instructions set the "direction" of the three I/O pins used for this application, pins `D2` and `D3` configured as inputs and pin `D13` as an output. By locating these instructions within the `setup` portion of the Sketch program, they are only executed once upon power-up.

- The `loop` portion of the Sketch program executes over and over again, just as the name "loop" implies. Looping is important for our program to repeatedly check the inputs for any new states and update the output accordingly. If not for looping, the program would not be responsive to any new logical states applied to the input pins, but would only check them at one moment in time and then never again!

- Each `digitalRead()` instruction reads the digital logic state of the specified pin. By setting `A` equal to the scanned state of pin `D2` and `B` equal to the scanned state of pin `D3` we have two variables in the microcontroller's memory which may be easily referenced for making decisions.

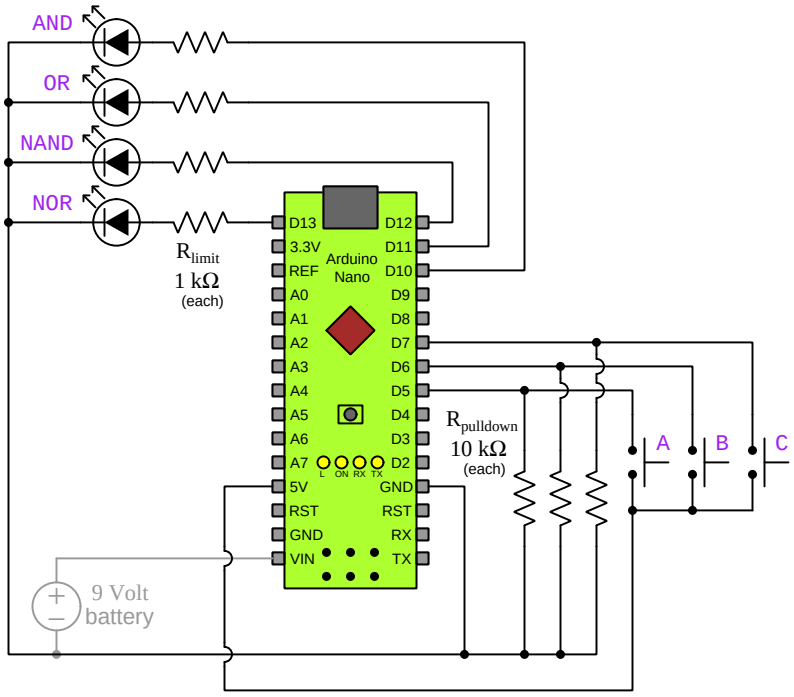
- The `if()` instruction checks for a condition of `A` *differing* in state from `B`. If this condition is met, the `digitalWrite()` instruction then sets pin `D13` to a "high" digital logic state to turn on the LED as an XOR logic gate would if it input states were unequal.

- The `else` instruction provides default action in the event that the `if()` instruction's conditions are *not* met. In this program, for any condition `A` and `B` share the same state we default to pin `D13` being cleared to a "low" digital logic state, just as the output of an XOR gate would go "low" if its inputs were the same.

2.4 Example: Arduino 3-input logic functions

This example was tested on an Arduino Nano containing the Atmel M328P microcontroller IC:

Schematic diagram



An optional 9 Volt battery provides electrical energy so that the Arduino may run independent of any other computer. When plugged into a programming computer's USB port, the Arduino draws power from that computer and requires no battery of its own.

Functional behavior – four LEDs for four different logic function outputs: AND, OR, NAND, and NOR.

- $R_{pulldown}$ provides definite "low" logic states to input pins `D5`, `D6`, and `D7` when their respective switches are open. If not for these resistors, the logical state of each input would be uncertain when its switch was open because it would be electrically "floating". The resistance value used here may be quite large, as its only purpose is to quickly drain off any stored electrical charge from the gates of the microcontroller's internal MOSFET transistors when the switch opens. Given the trivial amount of capacitance existing between the gate terminal and channel of a typical MOSFET (typically measured in *pico*-Farads), even a pulldown resistor value in the hundreds of kilo-Ohms results in a very rapid time constant ($\tau = RC$) sufficient for this manual pushbutton application.
- R_{limit} serves to limit the amount of current sunk by each LED when energized. Using a 1 kΩ resistor limits current well below the 20 mA rating for output pins on the Arduino Nano.

Sketch listing

```
int A, B, C;  // Lettered variables for the three inputs

void setup() {
 pinMode(10, OUTPUT); // sets digital pin 10 as an output
 pinMode(11, OUTPUT); // sets digital pin 11 as an output
 pinMode(12, OUTPUT); // sets digital pin 12 as an output
 pinMode(13, OUTPUT); // sets digital pin 13 as an output
 pinMode(5, INPUT);   // sets digital pin 5 as an input
 pinMode(6, INPUT);   // sets digital pin 6 as an input
 pinMode(7, INPUT);   // sets digital pin 7 as an input
}

void loop() {
  A = digitalRead(5);
  B = digitalRead(6);
  C = digitalRead(7);

  // 3-input AND function
  if (A == 1 && B == 1 && C == 1)
    digitalWrite(10, 1);

  else
    digitalWrite(10, 0);

  // 3-input OR function
  if (A == 1 || B == 1 || C == 1)
    digitalWrite(11, 1);

  else
    digitalWrite(11, 0);

  // 3-input NAND function
  digitalWrite(12, !digitalRead(10));

  // 3-input NOR function
  digitalWrite(13, !digitalRead(11));
}
```

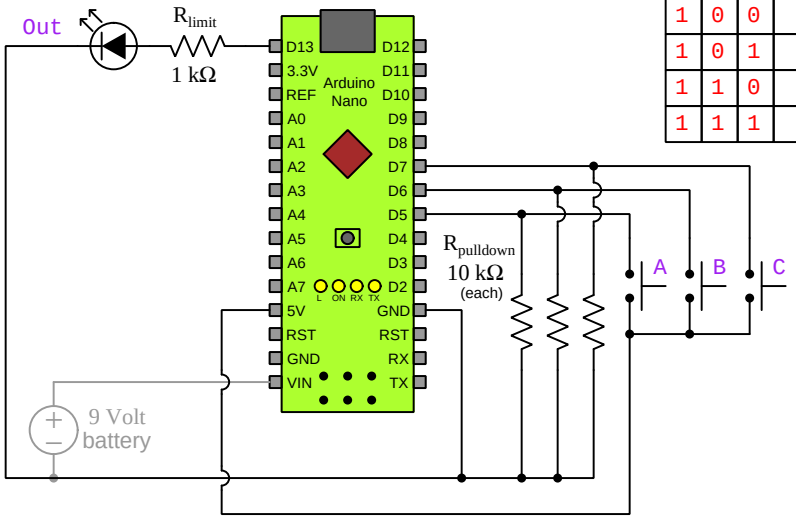
Note the use of the Boolean operators `&&` for AND, and `||` for OR. Note as well how the NAND and NOR functions simply read the AND and OR output states, respectively, and invert them using the `!` Boolean complement operator.

2.5 Example: Arduino SOP truth table implementation

This example was tested on an Arduino Nano containing the Atmel M328P microcontroller IC:

Schematic diagram

A	B	C	Out
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0



An optional 9 Volt battery provides electrical energy so that the Arduino may run independent of any other computer. When plugged into a programming computer's USB port, the Arduino draws power from that computer and requires no battery of its own.

Functional behavior – Boolean SOP expression $\overline{A}B\overline{C} + A\overline{B}C + AB\overline{C}$

- $R_{pulldown}$ provides definite "low" logic states to input pins `D5`, `D6`, and `D7` when their respective switches are open. If not for these resistors, the logical state of each input would be uncertain when its switch was open because it would be electrically "floating". The resistance value used here may be quite large, as its only purpose is to quickly drain off any stored electrical charge from the gates of the microcontroller's internal MOSFET transistors when the switch opens. Given the trivial amount of capacitance existing between the gate terminal and channel of a typical MOSFET (typically measured in *pico*-Farads), even a pulldown resistor value in the hundreds of kilo-Ohms results in a very rapid time constant ($\tau = RC$) sufficient for this manual pushbutton application.
- R_{limit} serves to limit the amount of current sunk by the LED when energized. Using a 1 kΩ resistor limits current well below the 20 mA rating for output pins on the Arduino Nano.

Sketch listing

```
int A, B, C;

void setup() {
 pinMode(13, OUTPUT); // sets digital pin 13 as an output
 pinMode(5, INPUT);   // sets digital pin 5 as an input
 pinMode(6, INPUT);   // sets digital pin 6 as an input
 pinMode(7, INPUT);   // sets digital pin 7 as an input
}

void loop() {
  A = digitalRead(5);
  B = digitalRead(6);
  C = digitalRead(7);

  // Evaluating (A' B C') term
  if (A == 0 && B == 1 && C == 0)
    digitalWrite(13, 1);

  // Evaluating (A B' C) term
  else if (A == 1 && B == 0 && C == 1)
    digitalWrite(13, 1);

  // Evaluating (A B C') term
  else if (A == 1 && B == 1 && C == 0)
    digitalWrite(13, 1);

  else
    digitalWrite(13, 0);
}
```

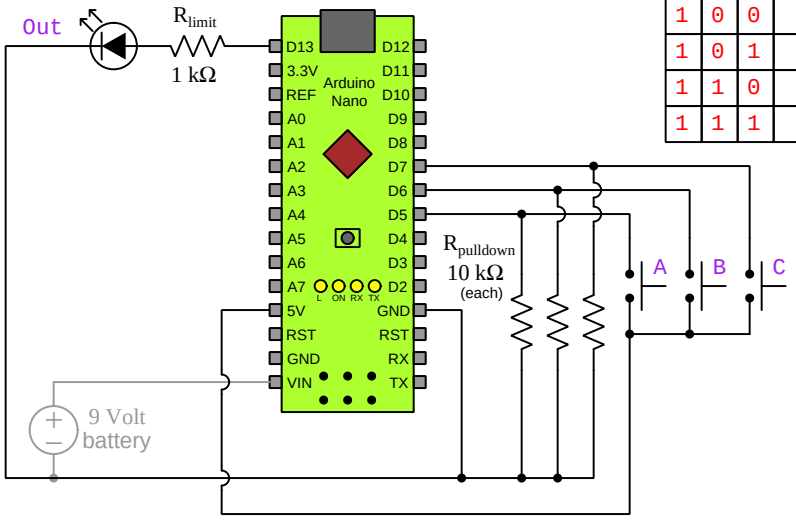
The essence of a Sum-Of-Products (SOP) Boolean expression is that it describes all input conditions causing the output to be true (1). Therefore, in this Sketch we test for each of the three possible input combinations describing a 1 output state as shown in the truth table, and set the output false (0) for all other conditions.

2.6 Example: Arduino NSOP truth table implementation

This example was tested on an Arduino Nano containing the Atmel M328P microcontroller IC:

Schematic diagram

A	B	C	Out
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1



An optional 9 Volt battery provides electrical energy so that the Arduino may run independent of any other computer. When plugged into a programming computer's USB port, the Arduino draws power from that computer and requires no battery of its own.

Functional behavior – Boolean NSOP expression $\overline{\overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}C + AB\overline{C}}$

- $R_{pulldown}$ provides definite "low" logic states to input pins `D5`, `D6`, and `D7` when their respective switches are open. If not for these resistors, the logical state of each input would be uncertain when its switch was open because it would be electrically "floating". The resistance value used here may be quite large, as its only purpose is to quickly drain off any stored electrical charge from the gates of the microcontroller's internal MOSFET transistors when the switch opens. Given the trivial amount of capacitance existing between the gate terminal and channel of a typical MOSFET (typically measured in *pico*-Farads), even a pulldown resistor value in the hundreds of kilo-Ohms results in a very rapid time constant ($\tau = RC$) sufficient for this manual pushbutton application.
- R_{limit} serves to limit the amount of current sunk by the LED when energized. Using a 1 kΩ resistor limits current well below the 20 mA rating for output pins on the Arduino Nano.

Sketch listing

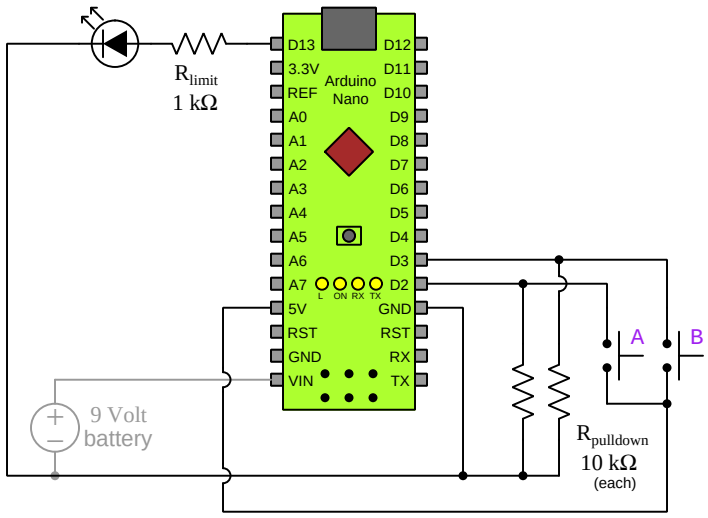
```
int A, B, C;

void setup() {
 pinMode(13, OUTPUT); // sets digital pin 13 as an output
 pinMode(5, INPUT);   // sets digital pin 5 as an input
 pinMode(6, INPUT);   // sets digital pin 6 as an input
 pinMode(7, INPUT);   // sets digital pin 7 as an input
}

void loop() {
  A = digitalRead(5);
  B = digitalRead(6);
  C = digitalRead(7);

  // Evaluating (A' B' C') term
  if (A == 0 && B == 0 && C == 0)
    digitalWrite(13, 0);

  // Evaluating (A' B' C) term
  else if (A == 0 && B == 0 && C == 1)
    digitalWrite(13, 0);

  // Evaluating (A B C') term
  else if (A == 1 && B == 1 && C == 0)
    digitalWrite(13, 0);

  else
    digitalWrite(13, 1);
}
```

The essence of a Negative-Sum-Of-Products (SOP) Boolean expression is that it describes all input conditions causing the output to be false (0). Therefore, in this Sketch we test for each of the three possible input combinations describing a 0 output state as shown in the truth table, and set the output true (1) for all other conditions.

2.7 Example: Arduino up counter with reset

This example was tested on an Arduino Nano containing the Atmel M328P microcontroller IC:

Schematic diagram



Functional behavior – microcontroller counts the number of times pushbutton `A` is pressed and activates the LED if that number is equal to or greater than five.

- $R_{pulldown}$ provides definite "low" logic states to input pins `D2` and `D3` when their respective switches are open. If not for these resistors, the logical state of each input would be uncertain when its switch was open because it would be electrically "floating". The resistance value used here may be quite large, as its only purpose is to quickly drain off any stored electrical charge from the gates of the microcontroller's internal MOSFET transistors when the switch opens. Given the trivial amount of capacitance existing between the gate terminal and channel of a typical MOSFET (typically measured in *pico*-Farads), even a pulldown resistor value in the hundreds of kilo-Ohms results in a very rapid time constant ($\tau = RC$) sufficient for this manual pushbutton application.

- R_{limit} serves to limit the amount of current sunk by each LED when energized. Using a 1 kΩ resistor limits current well below the 20 mA rating for output pins on the Arduino Nano.

Sketch listing

```
int A, last_A, B, countval = 0;

void setup() {
  pinMode(2, INPUT);
  pinMode(3, INPUT);
  pinMode(13, OUTPUT);
}

void loop() {
  A = digitalRead(2);
  B = digitalRead(3);

  if (A == HIGH && last_A == LOW)
    ++countval;

  if (B == HIGH)
    countval = 0;

  if (countval >= 5)
    digitalWrite(13, HIGH);

  else
    digitalWrite(13, LOW);

  last_A = A;
}
```

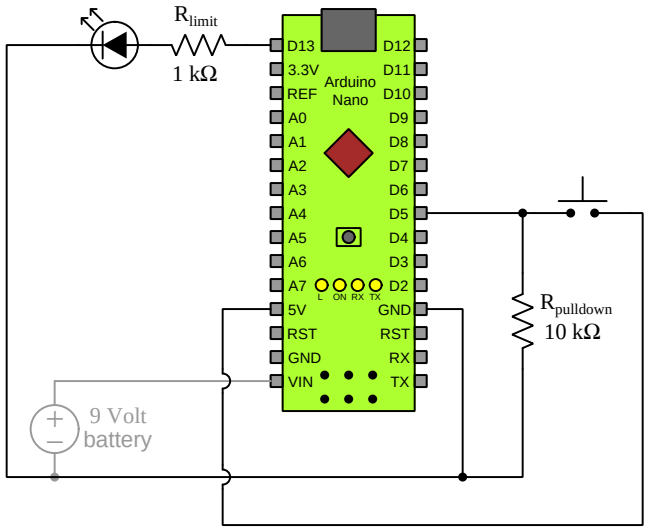
- The line `int A, last_A, B, countval = 0;` line at the very top of the Sketch program declares four *integer variables*, the last of those also being set to an initial value of zero.
- The `pinMode()` instructions set the "direction" of the three I/O pins used for this application, pins `D2` and `D3` configured as inputs and pin `D13` as an output. By locating these instructions within the `setup` portion of the Sketch program, they are only executed once upon power-up.
- The `loop` portion of the Sketch program executes over and over again, just as the name "loop" implies. Looping is important for our program to repeatedly check the inputs for any new states and update the output accordingly. If not for looping, the program would not be responsive to any new logical states applied to the input pins, but would only check them at one moment in time and then never again!
- Each `digitalRead()` instruction reads the digital logic state of the specified pin. By setting `A` equal to the scanned state of pin `D2` and `B` equal to the scanned state of pin `D3` we have two variables in the microcontroller's memory which may be easily referenced for making decisions.

- Out intent in building a counter system is to increment the value of the integer variable `countval` only at the rising edge of a pulse signal detected at pin `D2` and not at any other time. This means we should increment `countval` only at the moment in time when the scanned value of `A` is "high" (1) and the *previous* scanned value of `A` was "low" (0). This is why we have a variable named `A` and another named `last_A`, updating `A` at the top of every loop with the `digitalRead()` instruction and updating `last_A` immediately before repeating the loop to refresh the value of `A` with pin `D2`'s latest status.

2.8 Example: Arduino off-delay timer

This example was tested on an Arduino Nano containing the Atmel M328P microcontroller IC:

Schematic diagram



An optional 9 Volt battery provides electrical energy so that the Arduino may run independent of any other computer. When plugged into a programming computer's USB port, the Arduino draws power from that computer and requires no battery of its own.

Functional behavior – the LED energizes immediately when the pushbutton switch is pressed (closed), and remains on for a set period of time following the release (opening) of the pushbutton switch. In other words, *there is a time delay in turning off.*

- $R_{pulldown}$ provides a definite "low" logic state to input pin `D5` when the switch is open. If not for this resistor, the logical state of pin `D5` would be uncertain because it would be electrically "floating". The resistance value used here may be quite large, as its only purpose is to quickly drain off any stored electrical charge from the gates of the microcontroller's internal MOSFET transistors when the switch opens. Given the trivial amount of capacitance existing between the gate terminal and channel of a typical MOSFET (typically measured in *pico*-Farads), even a pulldown resistor value in the hundreds of kilo-Ohms results in a very rapid time constant ($\tau = RC$) sufficient for this manual pushbutton application.
- R_{limit} serves to limit the amount of current sunk by the LED when energized. Using a 1 kΩ resistor limits current well below the 20 mA rating for output pins on the Arduino Nano.

Sketch listing

```
int LastIN, IN = 0;       // variables storing current and past input states
int delaytime = 3000;     // variable storing desired time delay (ms)
unsigned long timestart;  // variable storing starting time value

void setup() {
  pinMode(13, OUTPUT);    // sets the digital pin 13 as an output
  pinMode(5, INPUT);      // sets the digital pin 5 as an input
}

void loop() {
  LastIN = IN;            // update old switch status
  IN = digitalRead(5);    // read new switch status

  if (IN == 0 && LastIN == 1)
    timestart = millis(); // initializes timestart when switch opens

  if ((IN == 0) && (millis() > (timestart + delaytime)))
    digitalWrite(13, 0);  // turns off pin 13 after time delay

  else if (IN == 0 && timestart == 0)
    digitalWrite(13, 0);  // ensures we don't start in an "on" state

  else
    digitalWrite(13, 1);  // turns on pin 13 before time delay
}
```

An important built-in function used in this Sketch is the `millis()` which returns an unsigned long integer value recording the number of milliseconds elapsed since the microcontroller was first powered up. Also important is our use of the `IN` and `LastIN` variables, used to determine the moment in time when the pushbutton switch transitions from an open state to a closed state. Note how the first instruction within the `loop()` sets `LastIN` equal in value to `IN`, and only after that step does the microcontroller actually read the value of input pin 5 to update the `IN` variable. This means `IN` represents the most recent scan of pin 5's state while `LastIN` represents the state of pin 5 during the loop's *last* iteration.

A slightly simpler version of this Sketch that does not rely on having to detect the pushbutton's moment of transition from open to closed appears here:

Sketch listing

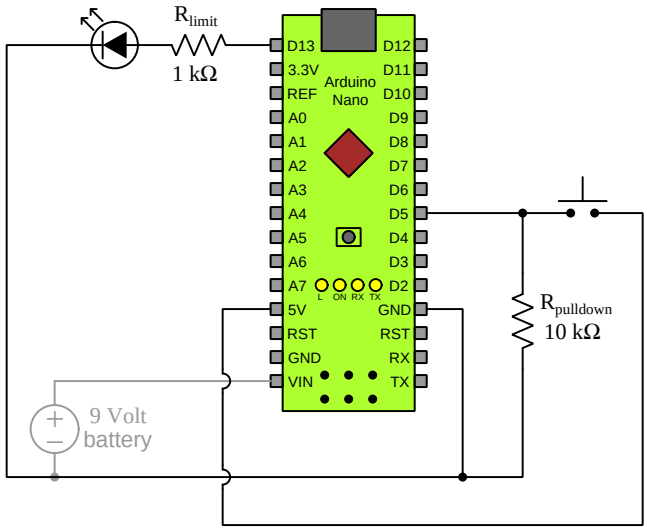
```
int delaytime = 3000;
unsigned long timestart = 0; // variable to store starting time value

void setup() {
 pinMode(13, OUTPUT); // sets digital pin 13 as an output
 pinMode(5, INPUT);   // sets digital pin 5 as an input
}

void loop() {
 if (digitalRead(5) == 1)
   timestart = millis();

 if (millis() > (timestart + delaytime))
   digitalWrite(13, 0); // ...to turn off D13 output pin

 else if (timestart == 0)
   digitalWrite(13, 0); // ensures we don't start in an "on" state

 else
   digitalWrite(13, 1); // turns on D13 output pin
}
```

Instead of updating the `timestart` variable at the moment of the switch's open-to-closed transition as the last Sketch did, this one keeps `timestart` concurrent with the `milli()` system clock so long as the switch is closed. When it opens, `timestart` holds at its last value while `milli()` of course keeps on incrementing, and when the difference between these two values grows great enough, the output goes low.

2.9 Example: Arduino on-delay timer

This example was tested on an Arduino Nano containing the Atmel M328P microcontroller IC:

Schematic diagram



An optional 9 Volt battery provides electrical energy so that the Arduino may run independent of any other computer. When plugged into a programming computer's USB port, the Arduino draws power from that computer and requires no battery of its own.

Functional behavior – the LED energizes only after the pushbutton switch has been continuously pressed (closed) for a set period of time, and will de-energize immediately upon the release (opening) of the pushbutton switch. In other words, *there is a time delay in turning on.*

- $R_{pulldown}$ provides a definite "low" logic state to input pin `D5` when the switch is open. If not for this resistor, the logical state of pin `D5` would be uncertain because it would be electrically "floating". The resistance value used here may be quite large, as its only purpose is to quickly drain off any stored electrical charge from the gates of the microcontroller's internal MOSFET transistors when the switch opens. Given the trivial amount of capacitance existing between the gate terminal and channel of a typical MOSFET (typically measured in *pico*-Farads), even a pulldown resistor value in the hundreds of kilo-Ohms results in a very rapid time constant ($\tau = RC$) sufficient for this manual pushbutton application.
- R_{limit} serves to limit the amount of current sunk by the LED when energized. Using a 1 kΩ resistor limits current well below the 20 mA rating for output pins on the Arduino Nano.

Sketch listing

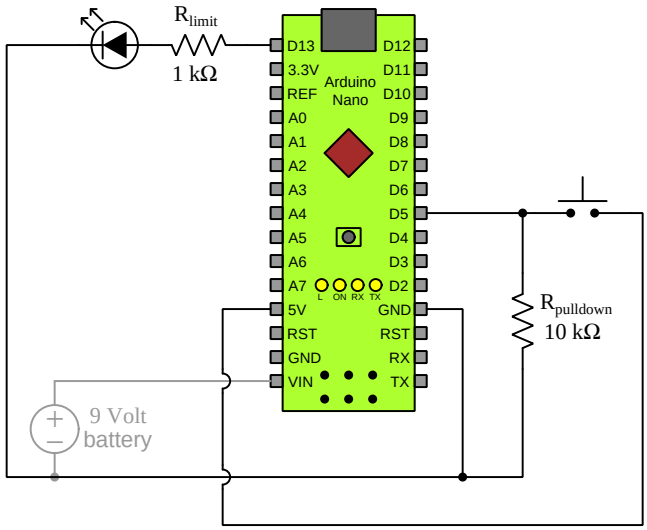
```
int LastIN, IN = 0;       // variables storing current and past input states
int delaytime = 3000;     // variable storing desired time delay (ms)
unsigned long timestart;  // variable storing starting time value

void setup() {
  pinMode(13, OUTPUT);    // sets the digital pin 13 as an output
  pinMode(5, INPUT);      // sets the digital pin 5 as an input
}

void loop() {
  LastIN = IN;            // update old switch status
  IN = digitalRead(5);    // read new switch status

  if (IN == 1 && LastIN == 0)
    timestart = millis(); // initializes timestart when switch closes

  if (IN == 1 && millis() > (timestart + delaytime))
    digitalWrite(13, 1);  // turns on pin 13 after time delay

  else
    digitalWrite(13, 0);  // turns off pin 13
}
```

An important built-in function used in this Sketch is the `millis()` which returns an unsigned long integer value recording the number of milliseconds elapsed since the microcontroller was first powered up.

2.10 Example: Arduino Morse code transmitter

This example was tested on an Arduino Nano containing the Atmel M328P microcontroller IC:

Schematic diagram



An optional 9 Volt battery provides electrical energy so that the Arduino may run independent of any other computer. When plugged into a programming computer's USB port, the Arduino draws power from that computer and requires no battery of its own.

Functional behavior – the LED pulses in proper sequence to represent English characters and/or numerals using Morse Code, one phrase for every press of the pushbutton switch.

- $R_{pulldown}$ provides a definite "low" logic state to input pin `D5` when the switch is open. If not for this resistor, the logical state of pin `D5` would be uncertain because it would be electrically "floating". The resistance value used here may be quite large, as its only purpose is to quickly drain off any stored electrical charge from the gates of the microcontroller's internal MOSFET transistors when the switch opens. Given the trivial amount of capacitance existing between the gate terminal and channel of a typical MOSFET (typically measured in *pico*-Farads), even a pulldown resistor value in the hundreds of kilo-Ohms results in a very rapid time constant ($\tau = RC$) sufficient for this manual pushbutton application.
- R_{limit} serves to limit the amount of current sunk by the LED when energized. Using a 1 kΩ resistor limits current well below the 20 mA rating for output pins on the Arduino Nano.

Sketch listing

```
#define TIME 100

int LastIN, IN = 0; // variables to store switch states
int Morse[36][5] = {
                     {3,3,3,3,3}, // 0      -----
                     {1,3,3,3,3}, // 1      .----
                     {1,1,3,3,3}, // 2      ..---
                     {1,1,1,3,3}, // 3      ...--
                     {1,1,1,1,3}, // 4      ....-
                     {1,1,1,1,1}, // 5      .....
                     {3,1,1,1,1}, // 6      -....
                     {3,3,1,1,1}, // 7      --...
                     {3,3,3,1,1}, // 8      ---..
                     {3,3,3,3,1}, // 9      ----.
                     {0,0,0,1,3}, // A = 10    .-
                     {0,3,1,1,1}, // B = 11  -...
                     {0,3,1,3,1}, // C = 12  -.-.
                     {0,0,3,1,1}, // D = 13   -..
                     {0,0,0,0,1}, // E = 14     .
                     {0,1,1,3,1}, // F = 15  ..-.
                     {0,0,3,3,1}, // G = 16   --.
                     {0,1,1,1,1}, // H = 17  ....
                     {0,0,0,1,1}, // I = 18    ..
                     {0,1,3,3,3}, // J = 19  .---
                     {0,0,3,1,3}, // K = 20   -.-
                     {0,1,3,1,1}, // L = 21  .-..
                     {0,0,0,3,3}, // M = 22    --
                     {0,0,0,3,1}, // N = 23    -.
                     {0,0,3,3,3}, // O = 24   ---
                     {0,1,3,3,1}, // P = 25  .--.
                     {0,3,3,1,3}, // Q = 26  --.-
                     {0,0,1,3,1}, // R = 27   .-.
                     {0,0,1,1,1}, // S = 28   ...
                     {0,0,0,0,3}, // T = 29     -
                     {0,0,1,1,3}, // U = 30   ..-
                     {0,1,1,1,3}, // V = 31  ...-
                     {0,0,1,3,3}, // W = 32   .--
                     {0,3,1,1,3}, // X = 33  -..-
                     {0,3,1,3,3}, // Y = 34  -.--
                     {0,3,3,1,1}, // Z = 35  --..
                    };
```

```
void setup() {
 pinMode(13, OUTPUT); // sets digital pin 13 as an output
 pinMode(5, INPUT); // sets digital pin 5 as an input
}

void loop() {
 LastIN = IN; // update old switch status

 IN = digitalRead(5); // read new switch status

 if (IN == 1 && LastIN == 0) // checks for switch press
 {
   transmit_character(17); // H
   transmit_character(18); // I
   delay (TIME * 7);       // Pause between "HI" and "THERE"
   transmit_character(29); // T
   transmit_character(17); // H
   transmit_character(14); // E
   transmit_character(27); // R
   transmit_character(14); // E
 }
}

void transmit_character (int n) {
 int pulse;

 for (pulse = 0 ; pulse < 5 ; ++pulse)
 {
   digitalWrite(13, 1);
   delay (TIME * Morse[n][pulse]);
   digitalWrite(13, 0);
   delay (TIME);     // Rests for one "dot" time
 }

 delay (TIME * 2);   // Rests two more "dot" times
}
```

This Sketch uses a two-dimensional array of integer numbers to store the proper timing sequences for all 36 characters in the Morse Code standard (26 letters of the English alphabet, plus numerals 0 through 9). No single Morse character is more than five pulses, with many of them being fewer than five. Each "dot" is one `TIME` period (set here at 100 milliseconds) and each "dash" is three `TIME` periods, explaining the 1's and 3's stored in the array, and all 0's in the array serving as "fillers" for Morse characters having less than five pulses each. The amount of "silence" between successive pulses should be one `TIME` period, and the silence between successive Morse-encoded characters three

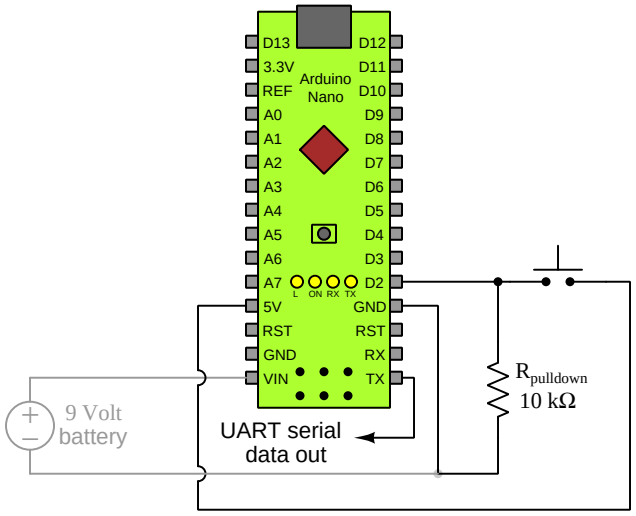
`TIME` periods. Between words there needs to be seven `TIME` periods' worth of silence.

Nothing is output by the microcontroller until the switch gets pressed, at which time a series of `transmit_character()` function calls request characters be transmitted in sequence. Each character is referenced by integer number (the first "subscript" of the array). Meanwhile, the `transmit_character()` function steps through all five possible pulses of each character (i.e. the second "subscript" of the array), adding appropriate "rest" delay times between each pulse and each character.

2.11 Example: Arduino generating UART serial output

This example was tested on an Arduino Nano containing the Atmel M328P microcontroller IC:

Schematic diagram

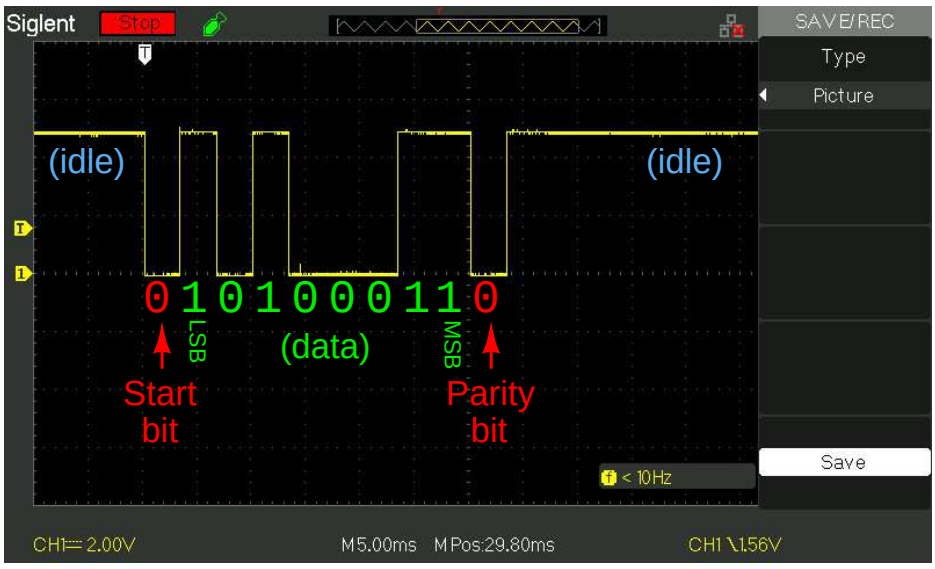


Serial data is where multiple bits of a digital word are output by a device one bit at a time on a single pin. *UART* stands for Universal Asynchronous Receiver/Transmitter which is an industry standard for the hardware and protocol of simple serial data communication. With the UART standard, the output pin's default of "idle" state is high, which turns to a low for one bit period to mark the start of the data transmission, then goes high/low as needed to serially represent the eight individual bits of the digital word, and finally returns to its default "idle" state which is high.

Sketch listing

```
int A, last_A;

void setup() {
  pinMode(2, INPUT);
  Serial.begin(300, SERIAL_8N1);  // 300 bps, 8 data, no parity, 1 stop
}

void loop() {
  A = digitalRead(2);

  if (A == HIGH && last_A == LOW)
    Serial.write(0b11000101);

  last_A = A;
}
```

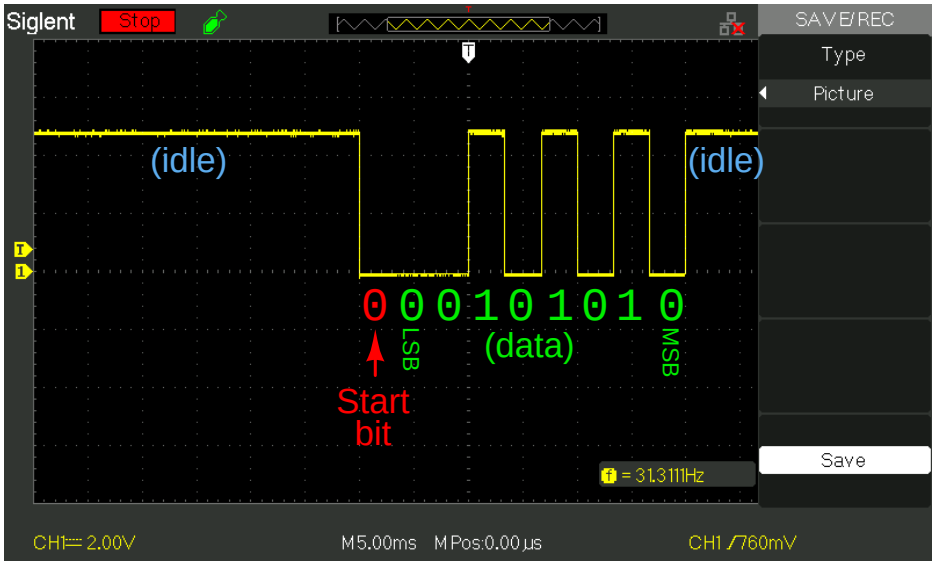
This particular Sketch program is written such that whenever the pushbutton is pressed, the microcontroller outputs the binary number `11000101` in reverse-bit order (i.e. the least-significant bit first and the most-significant bit last) with positive logic (i.e. a positive voltage represents a "1" bit and zero voltage represents a "0" bit). The following oscilloscope screenshot shows the result of pressing the pushbutton while measuring the voltage signal between the UART transmit (`TX`) pin and ground:



Sketch listing

```
int A, last_A;

void setup() {
  pinMode(2, INPUT);
  Serial.begin(300, SERIAL_8N1);  // 300 bps, 8 data, no parity, 1 stop
}

void loop() {
  A = digitalRead(2);

  if (A == HIGH && last_A == LOW)
    Serial.print("T");

  last_A = A;
}
```

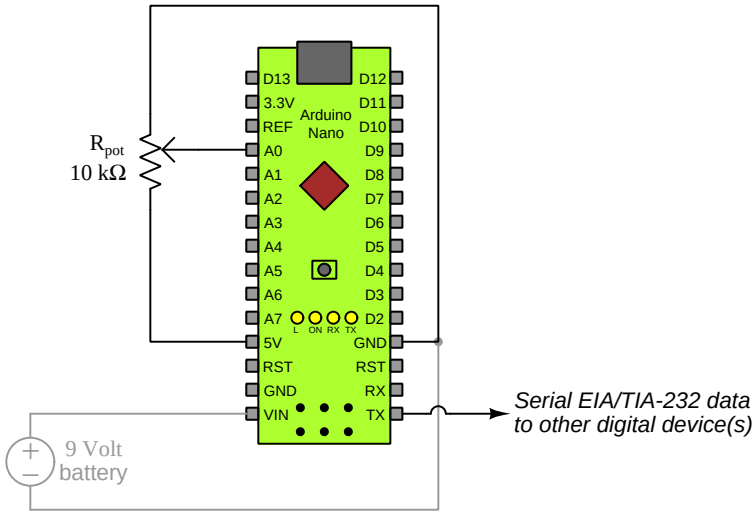
This particular Sketch program is written such that whenever the pushbutton is pressed, the microcontroller outputs the ASCII code for the capital letter `T` in reverse-bit order (i.e. the least-significant bit first and the most-significant bit last), which happens to be `01010100`. The following oscilloscope screenshot shows the result of pressing the pushbutton while measuring the voltage signal between the UART transmit (`TX`) pin and ground:



2.12 Example: Arduino reporting analog via serial

This example was tested on an Arduino Nano containing the Atmel M328P microcontroller IC:

Schematic diagram



An optional 9 Volt battery provides electrical energy so that the Arduino may run independent of any other computer. When plugged into a programming computer's USB port, the Arduino draws power from that computer and requires no battery of its own.

Functional behavior – the potentiometer's position is read as an integer number between 0 and 1023, that numerical value reported as ASCII text out the serial in.

- R_{pot} serves as a voltage divider to send a 0-5 Volt analog signal to analog input pin `A0`. Its value is not particularly important, but should not be so low as to unnecessarily load down the Arduino's on-board 5 Volt regulator, and not be so high that the analog input causes the wiper's potential to "sag" due to loading effects.

Sketch listing

```
int x;

void setup() {
  Serial.begin(9600);  // Configures UART speed for 9600 bps
}

void loop() {
  x = analogRead(A0);
  Serial.println(x);
  delay(500);
}
```

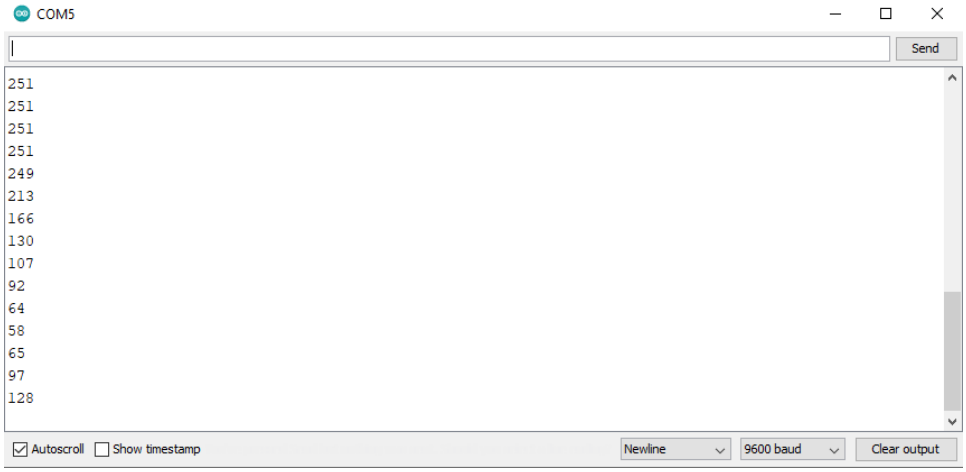
Reading analog voltages at any of the Arduino's analog input pins is pure simplicity – there isn't even a need to set up the pin to be an analog input because it defaults to that mode! All we need to do is use the `analogRead()` function for the specified pin, the 0-5 Volt analog voltage converted into a 0-1023 "count" value by the microcontroller's internal analog-to-digital converter (ADC).

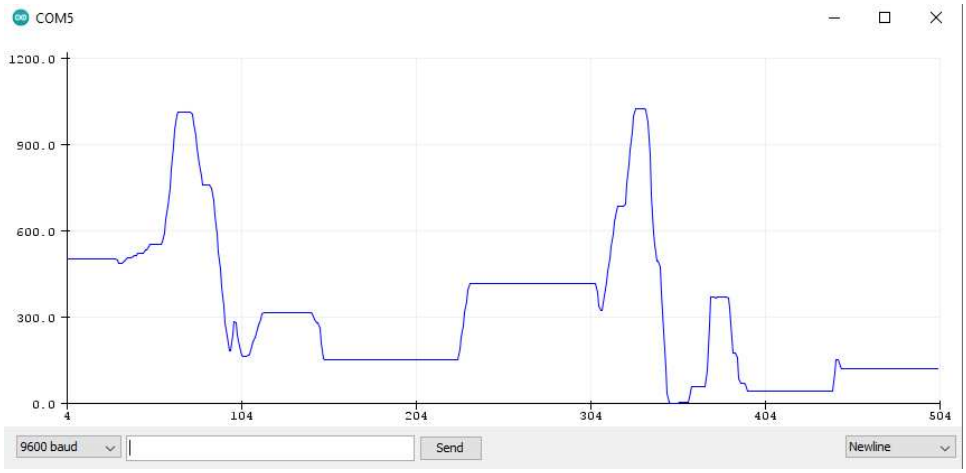
For serial data output, we must set up the serial bit rate to the desired value (9600 bits per second is quite common) and then invoke the `Serial.println()` function to print lines of text on any device capable of receiving and displaying that data out of pin `TX1`. Note that the signals read between pin `TX1` and ground are 5-Volt TTL-level logic and thus are not proper EIA/TIA-232 signal levels ($\pm$ 5 Volts minimum). If we wish to connect the Arduino to a standard EIA/TIA-232 serial port on some other device, we will need to insert a *transceiver* circuit between the two.

A pair of extremely useful utilities that are part of the Arduino integrated development environment (IDE) are the following:

- **Serial Monitor** – a window on the programming computer displaying all ASCII text output by the Arduino when executing a `Serial.print` function
- **Serial Plotter** – a window on the programming computer displaying all numerical data output as ASCII text by the Arduino when executing a `Serial.print` function, shown as a time-domain graph

Examples of these utilities in action displaying data from this Sketch appear here, first the Serial Monitor and then the Serial Plotter:



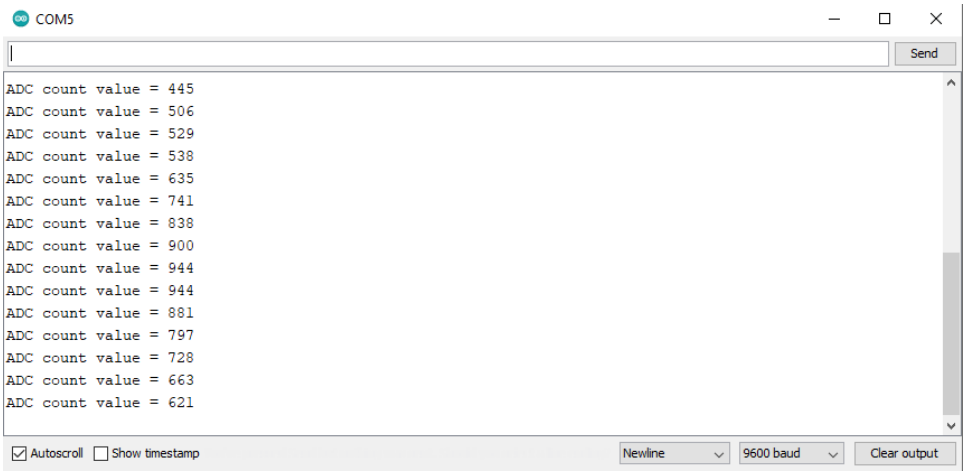


With just a minor addition of code to this Sketch, we may add some literal text to the serial output stream so that the Serial Monitor's display is more easily understood by anyone reading it:

Sketch listing

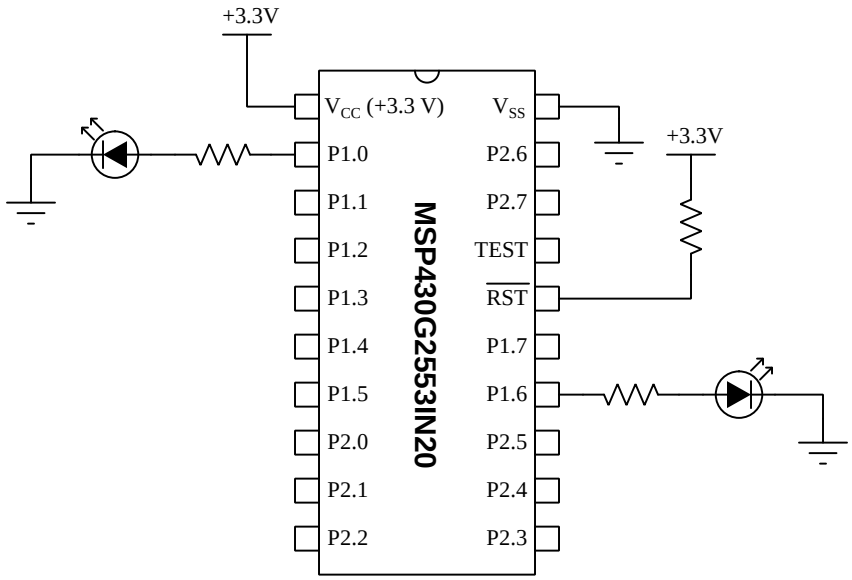
```
int x;

void setup() {
  Serial.begin(9600);  // Configures UART speed for 9600 bps
}

void loop() {
  x = analogRead(A0);
  Serial.print("ADC count value = ");
  Serial.println(x);
  delay(500);
}
```

The `Serial.print()` function does not include a "newline" character at the end of the serial data stream, which means the value of `x` printed by the `Serial.println()` function appears immediately to the right of the `ADC count value =` string when viewed on the Serial Monitor:



2.13 Example: MSP430 alternating LED blink

Schematic diagram



On the next page is a listing of the entire C-language code, tested and run on a Texas Instruments model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS).

The statement beginning with a pound symbol (`#include <msp430.h>`) is a *directive* telling the compiler how to convert the C source code into object code and then executable machine code for the microcontroller to run.

C code listing

```
#include <msp430.h>

void delayloop(void);

void main(void)
{
  WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer
  P1DIR |= 0xFF; // configure all Port 1 pins as outputs

  while(1)
  {
    P1OUT = 0x01;
    delayloop();
    P1OUT = 0x40;
    delayloop();
  }
}

void delayloop()
{
  unsigned int delay;

  for(delay=0xFFFF; delay>0; --delay);
}
```

The line `P1DIR |= 0xFF;` is an example of a bitwise operation and assignment. It is equivalent to the more verbose expression `P1DIR = P1DIR | 0xFF`, setting `P1DIR` equal to the bitwise-OR function of its former value and `0xFF`.

An interesting point to note about this program is that the compiler identified the `delayloop()` function as an inefficient use of computational power and recommended the use of the MCU's built-in timer capabilities. With standard optimization turned on (`-O2`) the compiler actually skipped the `for` loop which left the program with no time delay at all! I had to disable all optimizations in order to have the code compile and run as written. Interestingly, when run in *debug* mode and single-stepped, the program did exactly what it was supposed to do.

We may achieve the same effect by using a special *intrinsic* function provided by Code Composer Studio named `__delay_cycles()`. This instruction, which accepts only an *unsigned long integer constant* value as its argument and cannot work with a variable, generates a definite sequence of assembly-language instructions when compiled to produce the same effect as the `delayloop()` function we made in the previous example. If all you need is a fixed delay time, the `__delay_cycles()` instruction is a very good solution and makes for much more compact C code:

C code listing

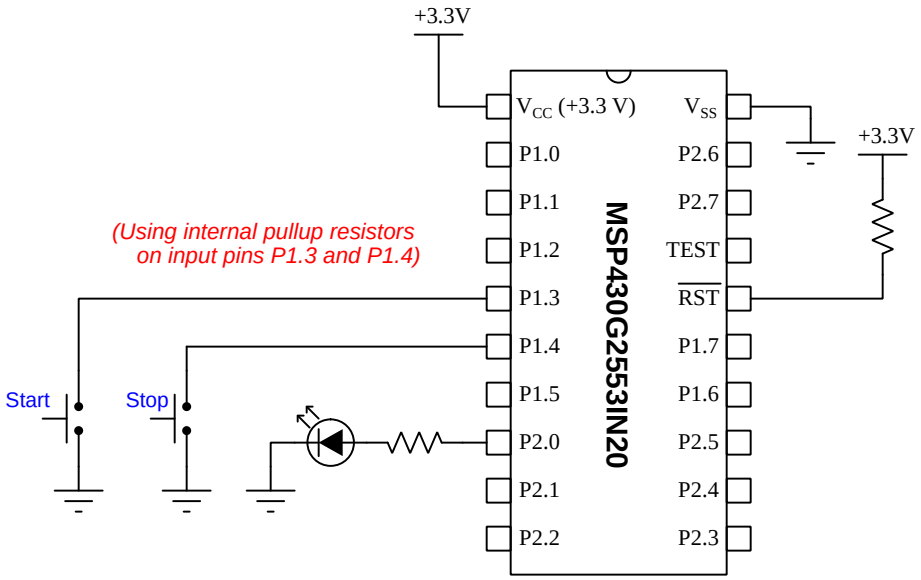
```
#include <msp430.h>

void main(void)
{
  WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer
  P1DIR |= 0xFF; // configure all Port 1 pins as outputs

  while(1)
  {
    P1OUT = 0x01;
    __delay_cycles(100000);
    P1OUT = 0x40;
    __delay_cycles(100000);
  }
}
```

The actual amount of time delay, as with our `delayloop()` function, depends on the clock speed of the microcontroller.

The two `P1OUT` assignment statements control the alternating `P1.0` and `P1.6` states. When setting `P1OUT` equal to `0x01`, what we are really doing is setting Port 1's eight bits to the following binary states: `0b00000001`, making pin `P1.0` "high" and the rest low. When setting `P1OUT` equal to `0x40`, what we are really doing is setting Port 1's eight bits to the following binary states: `0b01000000`, making pin `P1.6` "high" and the rest low. By editing these values assigned to `P1OUT`, we may specify any two-state "blinking" sequence desired with Port 1's eight output pins.

2.14 Example: MSP430 start-stop control

Schematic diagram



On the next page is a listing of the entire C-language code, tested and run on a Texas Instruments model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS).

The statement beginning with a pound symbol (`#include <msp430.h>`) is a *directive* telling the compiler how to convert the C source code into object code and then executable machine code for the microcontroller to run.

C code listing

```
#include <msp430.h>

void main(void)
{
  WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer

  P2DIR = 0x01;  // Pin P2.0 is an output (LED)
  P2OUT = 0x00;  // Ensure LED begins in the "off" state
  P1DIR = 0x00;  // All P1.x pins are inputs
  P1REN = 0x18;  // Enable pullup resistors for P1.4 and P1.3
  P1OUT = 0x18;  // Resistors are pullup, not pulldown

  while(1)
  {
    if ((P1IN & BIT4) == 0)      // Stop LED if switch P1.4 pressed
      P2OUT &= ~BIT0;

    else if ((P1IN & BIT3) == 0) // Start LED if switch P1.3 pressed
      P2OUT |= BIT0;
  }
}
```

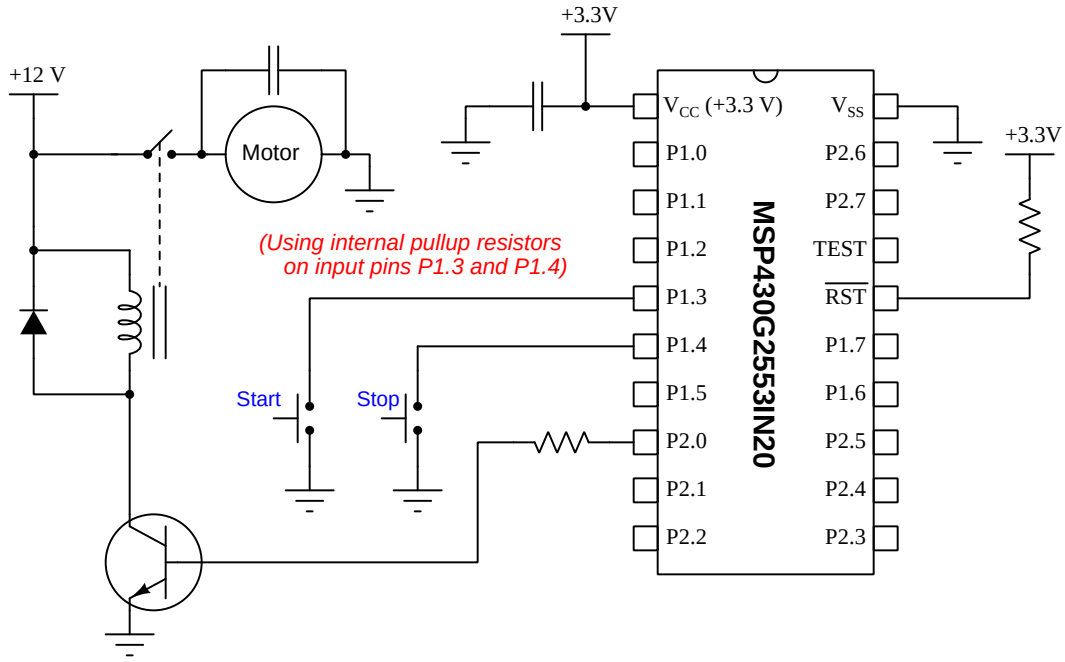
Note the use of the symbolic constants `BIT0`, `BIT3`, and `BIT4` to represent the hexadecimal values `0x01`, `0x08`, and `0x10`, respectively. The "tilde" symbol (`~`) is the bitwise logical operator for inversion, which means while `BIT0` represents `0b00000001`, `~BIT0` must represent `0b11111110`.

These symbolic constants, as well as many more, are all defined in the header file `msp430.h` for the purpose of making the source code of your program easier to read. The microcontroller doesn't care whether you use `BIT0` or `0b00000001` or `0x01`, as the assembler replaces all those symbols with the constant values they represent.

This form of program is useful for controlling the starting and stopping of a load such as an LED or an electric motor. In fact, the only modification we would have to make to the circuit to control a motor is to have the microcontroller drive a transistor and/or relay to switch more voltage/current to the motor than output `P2.0` is capable of delivering on its own (see an example of this on the next page).

A safety feature of this program is how the "stop" command is given precedence over the "start" command. This is why an `else if` conditional is used for the "start" instruction: if both "start" and "stop" buttons are pressed, the program will execute the "stop" condition and skip any evaluation of the "start" condition. This is also why the `P2OUT` register is cleared immediately after enabling pin `P2.0` as an output: to ensure the LED (or motor) will not begin in an energized state by chance.

The following schematic shows how this same start-stop program could be used to control an electric motor operating on a higher-voltage power supply, using both a transistor and an electromechanical relay to "interpose" between the microcontroller and motor:



Both the +12 Volt and the +3.3 Volt DC supplies must share a common ground connection, so that current from pin P2.0 driving the NPN transistor's base terminal can find a path back to the +3.3 Volt source after exiting the transistor's emitter terminal, but otherwise should be separate power sources. The reason for this separation is to eliminate the potential problem of electrical "noise" generated by the motor interfering with the microcontroller (which requires very "clean" DC power). Note also the presence of "decoupling" capacitors across the microcontroller's power terminals and motor terminals, both helpful in mitigating[3] electrical noise produced by the motor.

The *commutating diode* connected "backwards" in parallel with the relay's coil provides a safe discharge path for any inductive "kickback" that may result when the transistor turns off and the stored energy in the relay's coil inductance acts to maintain current in the same direction as before when the transistor was on. This diode will be reverse-biased and non-conducting when the transistor is turned on, but when the relay coil's inductance generates a reverse voltage polarity when the transistor turns off this diode will forward-bias and conduct to provide that inductance a non-destructive pathway for its current. Without this commutating diode in place, we run the risk of destroying the transistor (and also the microcontroller!) when de-energizing an inductive load.

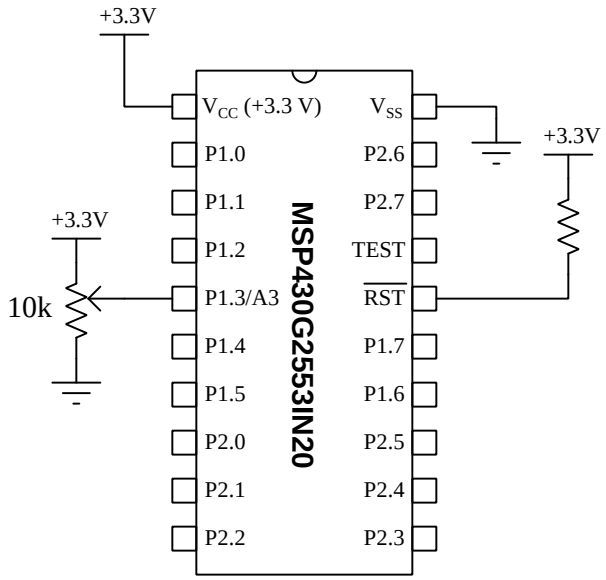
What was formerly a current-limiting resistor for the LED is now a current-limiting resistor for the transistor's base terminal. It must be of sufficient resistance to keep the driving current less

[3] Without this mitigation, it's entirely possible that the electrical noise produced by the running motor will cause the microcontroller to "glitch" (i.e. reset).

than both the transistor's base terminal current limit and the microcontroller's output pin current limit (usually on the order of several milliAmperes – but check the datasheets to be sure!), but not so large in resistance that the transistor fails to fully turn on (saturate) for efficient operation.

2.15 Example: MSP430 crude analog input

Schematic diagram



On the next page is a listing of the entire C-language code, tested and run on a Texas Instruments model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS) *in Debug mode*.

The statement beginning with a pound symbol (`#include <msp430G2553.h>`) is a *directive* telling the compiler how to convert the C source code into object code and then executable machine code for this specific model of microcontroller to run.

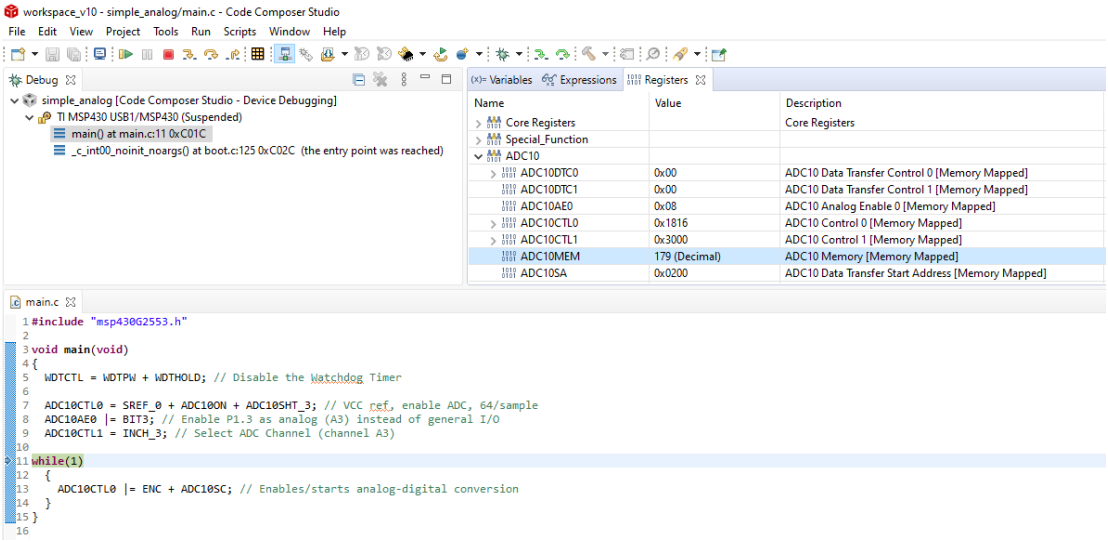
C code listing

```
#include "msp430G2553.h"

void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;  // Disable the Watchdog Timer

  /* Configure ADC */
  ADC10CTL0 = SREF_0 + ADC10ON + ADC10SHT_3; // VCC ref, enable ADC, 64/sample
  ADC10AE0 |= BIT3;       // Enable P1.3 as analog (A3) instead of general I/O
  ADC10CTL1 = INCH_3;     // Select ADC Channel (channel A3)

  while(1)
  {
     ADC10CTL0 |= ENC + ADC10SC;   // Enables/starts analog-digital conversion
  }
}
```

This is an *extremely* simple program for reading a single analog input channel on the MSP430 microcontroller, and in fact it is only usable while using the "Step Into" function of Code Composer Studio's "Debug" mode because there is no time delay in the code or any other provision to give the microcontroller enough time to adequately process the analog signal. By repeatedly clicking on the "Step Into" button within CCS, you may watch the numerical value stored in `ADC10MEM` as you vary the voltage applied between pin `P1.3` and Ground on the microcontroller.

In this example circuit, the analog voltage is provided to pin `P1.3` using a three-terminal component called a *potentiometer*. This resistive device uses a movable "wiper" contact to divide the 3.3 Volt DC power supply voltage by a proportion determined by the wiper's position. With the wiper placed fully up pin `P1.3` sees 3.3 Volts with respect to ground; placed fully down it sees 0 Volts; when placed mid-position the voltage will be 1.65 Volts.

In the following screenshot, you can see the Debug view of CCS showing the program's execution (currently halted at the `while` instruction) and the decimal value 179 shown within `ADC10MEM`:



The "Step Into" control button is the one resembling a yellow arrow pointing right and down, immediately to the right of the red-square "Stop" button on the CCS toolbar. The `ADC10MEM` "count" value of 179 just happens to correspond to the analog voltage signal being read by the microcontroller at the time.

All analog-to-digital converters generate a (digital) numerical value in response to the amount of (analog) voltage sensed. In the case of the MSP430G2553 microcontroller, that digital number value has a range of 0 to 1023 and should be directly proportional to the analog voltage over a range of 0 Volts to 3.3 Volts (the microcontroller's regulated DC power supply voltage). We may express this mathematically as a simple proportion:

$$\frac{V_{analog}}{3.3} = \frac{n_{digital}}{1023}$$

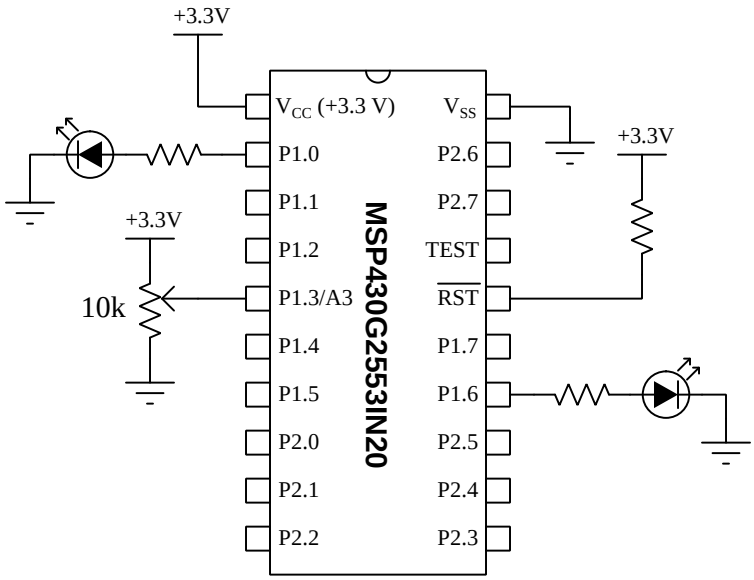
Where,

V_{analog} = Analog voltage sensed at the input terminal with reference to ground

$n_{digital}$ = Digital "count" value produced by the analog-to-digital conversion process

2.16 Example: MSP430 analog-controlled LEDs

Schematic diagram



On the next page is a listing of the entire C-language code, tested and run on a Texas Instruments model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS).

The statement beginning with a pound symbol (`#include <msp430G2553.h>`) is a *directive* telling the compiler how to convert the C source code into object code and then executable machine code for this specific model of microcontroller to run.

C code listing

```
#include "msp430G2553.h"

void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;  // Disable the Watchdog Timer

  /* Configure Output pins for LEDs */
  P1DIR |= BIT0 + BIT6; // Set P1.0 and P1.6 as outputs

  /* Configure Timer A */
  TA0CTL = TASSEL_2 + MC_2 + TAIE; // SMCLK clock, continuous, enable interrupt

  /* Configure ADC */
  ADC10CTL0 = SREF_0 + ADC10ON + ADC10SHT_3; // VCC ref, enable ADC, 64/sample
  ADC10AE0 |= BIT3;       // Enable P1.3 as analog (A3) instead of general I/O
  ADC10CTL1 = INCH_3;     // Select ADC Channel (channel A3)

  __bis_SR_register(GIE); // General Interrupts Enabled

  while(1)
  {
     if (ADC10MEM > 341)  // ADC count value stored in ADC10MEM register
         P1OUT |= BIT6;   // Turn on P1.6 if voltage > 1/3 of VCC

     if (ADC10MEM <= 341)
         P1OUT &= ~BIT6;  // Turn off P1.6 if voltage <= 1/3 of VCC

     if (ADC10MEM > 682)
         P1OUT |= BIT0;   // Turn on P1.0 if voltage > 2/3 of VCC

     if (ADC10MEM <= 682)
         P1OUT &= ~BIT0;  // Turn off P1.0 if voltage < 2/3 of VCC
  }
}

#pragma vector=TIMER0_A1_VECTOR
__interrupt void Timer_ISR (void)  // Timer A interrupt service routine
{
  ADC10CTL0 |= ENC + ADC10SC;      // Enables/starts analog-digital conversion
  TA0CTL &= ~TAIFG;                // Clears the Timer A interrupt flag
}
```

This program uses Timer A to generate a periodic interrupt event to re-enable and re-start the analog-to-digital conversion process.

Noteworthy features of this program include:

- Extensive use of symbolic constants to represent what would otherwise be arcane bit patterns; e.g. `BIT6` is a symbol for the binary constant `0b01000000` (hex `0x40`)
- Use of the timer circuit's functionality to replace what would otherwise be code running within the `while()` loop to repeatedly activate the microcontroller's on-board 10-bit analog-to-digital converter
- Inclusion of an interrupt service routine (ISR) called by Timer A's interrupt event

Let's examine this code section by section.

After disabling the watchdog timer, the first instruction sets the "direction" of pins `P1.0` and `P1.6` to both be outputs. We will use these two outputs to drive LEDs, turning each of them on at different sensed voltage levels.

The on-board 10-bit analog-to-digital converter requires a short period of time to complete each conversion of a sampled analog input voltage, and must be re-started after each conversion is complete. A convenient way to do this periodic re-start is to configure the on-board timer (which is nothing more than a 16-bit counter driven by the clock signal) and have that timer generate an interrupt request every time it "rolls over" from full-count back to zero. This is called the timer's "continuous" mode, where it counts from `0x0000` to `0xFFFF` repeatedly, setting its interrupt request flag (IFG) every time it returns to a value of zero. The `TA0CTL` initialization line sets up Timer A for this purpose.

Next, we need to initialize registers controlling the analog-to-digital converter itself. The example shown here is perhaps the simplest configuration possible for the MSP430's 10-bit ADC: we declare the analog reference voltages to be the chip's power supply (V_{CC} = 3.3 Volts and V_{SS} = 0 Volts), enable the ADC, and specify 64 counts per sample-and-hold cycle to give the analog signal voltage the maximum time to settle. On the model MSP430G2553IN20, every pin associated with Port 1 is able to serve as an analog input rather than a general I/O pin, and with the `ADC10AE0` initialization we declare the pin normally associated with I/O `P1.3` to be `A3` instead. Lastly, we select this analog-enabled pin to be the analog input signal for the ADC.

The last instruction executed before entering the `while()` loop sets the General Interrupt Enable (`GIE`) bit in the microcontroller's Status Register, a necessary action if we are to use any general interrupts in our program.

In its default mode, the ADC10 converter generates an unsigned 10-bit count value ranging from 0 to 1023 (decimal) for a corresponding analog voltage signal ranging from the low reference voltage value to the high reference voltage value. Since we are using the chip's power supply as those reference limits, we should expect to find a count value of 0 in the `ADC10MEM` register when pin `A3` senses 0 Volts, 1023 in that same register when it senses full (+3.3 Volts) signal voltage, and a count value proportionate to these limits for any voltage in-between +3.3 and 0 Volts. It is the

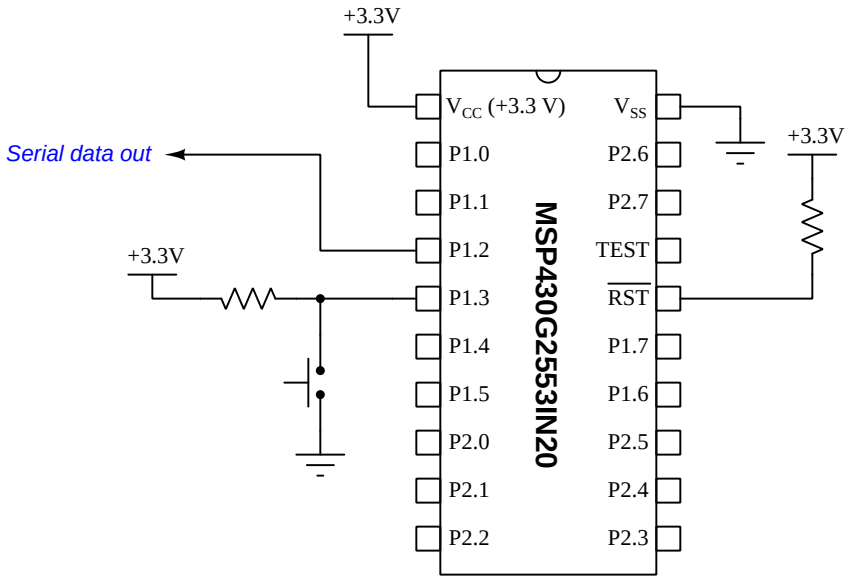
intent of this program to energize LED `P1.6` for any voltage greater than $\frac{1}{3}$ of the supply voltage, and energize LED `P1.0` for any voltage exceeding $\frac{2}{3}$ the supply voltage. Therefore, the values 341 (one-third of 1023) and 682 (two-thirds of 1023) serve as the conditional thresholds for activating these LEDs.

The interrupt service routine function contains two instructions: one of them re-enables and re-starts the analog-digital conversion process, necessary after each conversion has completed. The second instruction clears the interrupt flag that was set by the timer when it returned to a value of zero. This clearing of the flag bit is typical for interrupt service routines, and is necessary[4] to prevent that routine from being repeatedly called in an endless loop.

[4]Interestingly, certain interrupt flags within the MSP430 microcontroller do not require the ISR to explicitly clear the flag bit. An example of this is the Capture/Compare interrupt flag generated by certain counting modes of the timer (`CCIFG`), which gets automatically cleared every time the interrupt request is granted. For most interrupts, expect that you will have to clear that interrupt flag bit within the code you write for the interrupt service request function.

2.17 Example: MSP430 UART serial text transmission

Schematic diagram



On the next two pages is a listing of the entire C-language code, tested and run on a Texas Instruments model MSP430G2553IN20 microcontroller using a LaunchPad model MSP-EXP430G2ET development board and version 10 of Code Composer Studio (CCS).

The statement beginning with a pound symbol (`#include <msp430.h>`) is a *directive* telling the compiler how to convert the C source code into object code and then executable machine code for this specific model of microcontroller to run.

C code listing

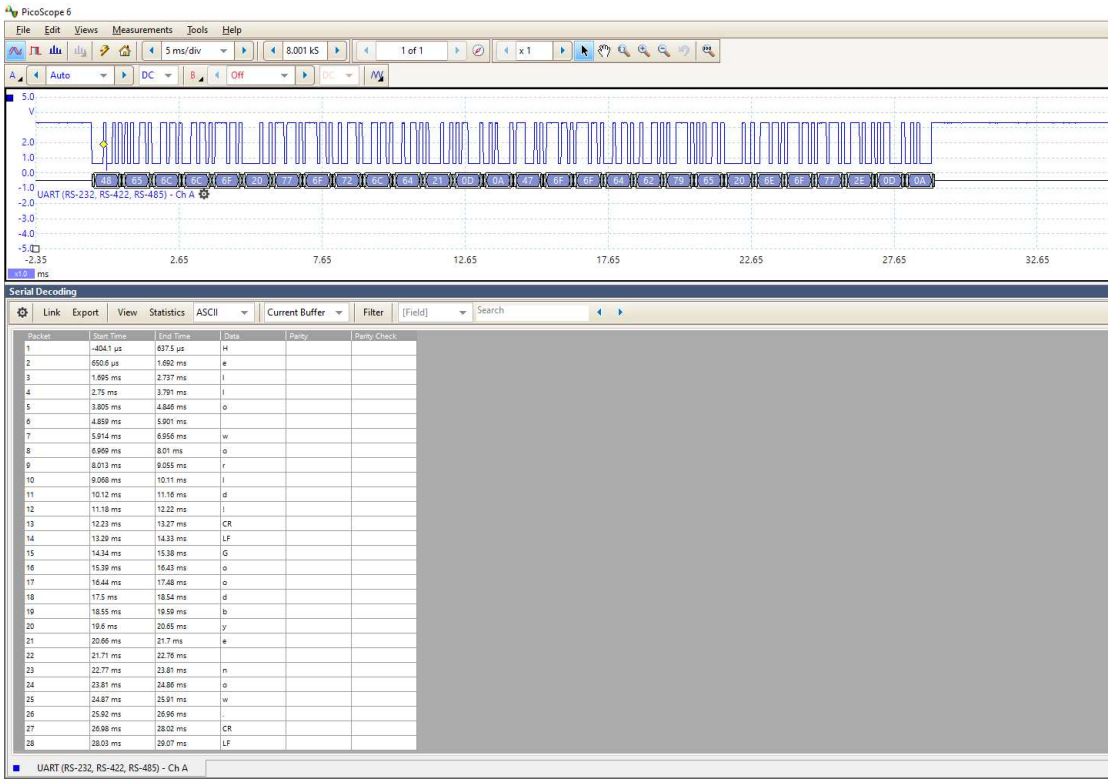
```
#include <msp430.h>

void transmit_ASCII(char * tx_data);   // Prototype for transmit function

void main(void)
{
  unsigned int button, lastbutton;

  WDTCTL = WDTPW + WDTHOLD;         // Stop watchdog timer

  BCSCTL1 = CALBC1_1MHZ;            // Set DCO to 1 MHz
  DCOCTL = CALDCO_1MHZ;

  P1DIR = 0xF7;                     // P1.3 input, all other P1 pins output
  P1REN = 0x08;                     // Enables pullup/pulldown resistor on P1.3
  P1OUT = 0x08;                     // Selects pullup resistor for P1.3
  P1SEL = BIT1 + BIT2;              // P1.1 is UCA0RXD and P1.2 is UCA0TXD
  P1SEL2 = BIT1 + BIT2;             //  when P1SEL and P1SEL2 bits set

  UCA0CTL1 |= UCSSEL_2;             // Have USCI use System Master Clock
  UCA0BR0 = 104;                    // These two USCI_A0 8-bit clock prescalers
  UCA0BR1 = 0;                      //  comprise a 16-bit value to set bit rate.
                                    //  Value of 104 = 9600 bps with 1MHz clock
                                    //  (as per table 15.4 in MSP430 user guide)
  UCA0MCTL = UCBRS0;                // Bit rate may be further tweaked using the
                                    //  "modulation" register
  UCA0CTL1 &= ~UCSWRST;             // Start the USCI's state machine

  while(1)                          // Endless loop
  {
      if ((P1IN & BIT3) == BIT3)    // "button" represents state of P1.3
          button = 1;               // == 1 when P1.3 is high
      else
          button = 0;               // == 0 when P1.3 is low

      if((button == 0) && (lastbutton == 1))     // Detects P1.3 high-to-low
      {                                         //  without requiring interrupt
          transmit_ASCII("Hello world!\r\n");   // Transmit text
          transmit_ASCII("Goodbye now.\r\n");   // Transmit more text
      }
      lastbutton = button;
  }
```

```
}

void transmit_ASCII(char * char_array) // Function accepts char. array pointer
{
    unsigned int n = 0;
    while(char_array[n])             // Increment until null pointer (0) reached
    {
        while ((UCA0STAT & UCBUSY)); // Wait if line TX/RX module busy with data
        UCA0TXBUF = char_array[n];   // Load each character into transmit buffer
        ++n;                         // Increment variable for array address
    }
}
```

Pressing the pushbutton brings `P1.3` input to a "low" state. When the present state is low and the last state was high it means a high-to-low transition has occurred and this initiates the transmission of our text message. Without some provision to detect the falling edge of this input signal what would happen is the message would be transmitted over and over again for as long as `P1.3` remains low. Other methods exist for detecting a high-to-low transition (e.g. using an interrupt generated by `P1.3`), but this current/last state method is worth considering which is why I presented it in this example.

In the C language it is impossible to pass an array or other complex data structure as an argument to function, and so instead the `transmit_ASCII()` function receives a *pointer* to an array comprised of ASCII characters, that array being named `char_array` within the function. Within the `transmit_ASCII` function's `while()` loop we index this array to read its contents one character at a time until the end is reached as indicated by a "null" value. Note also the inner `while()` loop which acts to halt the microcontroller's execution (by uselessly looping) until the status is no longer busy.

When the output signal (`P1.2`) is measured on a digital oscilloscope capable of RS-232 serial data decoding, we see the following result:



Chapter 3

Tutorial

3.1 Microcontrollers and PLCs

Semiconductor logic gates and electromechanical relays are both useful devices for constructing discrete (on/off) control circuits for a wide range of automation applications. However, both logic gates and relays suffer from the limitation of being "hard-wired" devices which must literally be re-wired in order to alter the control system's function. The benefits of a *programmable* system are obvious, where it is possible to alter the logical relationships between inputs and outputs simply by "loading" digital codes into the device. This Tutorial explores two popular types of programmable control devices, microcontrollers and PLCs.

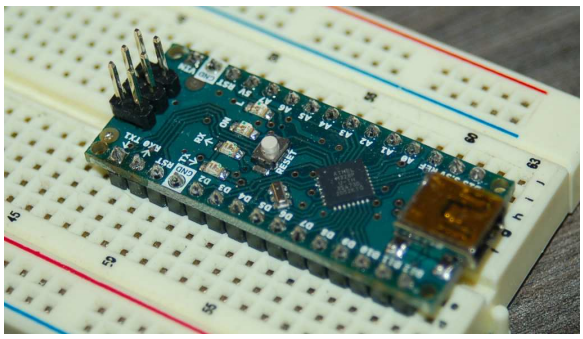
A *microcontroller* is a complete digital computer packaged as a single integrated circuit. These single-chip devices are programmed much the same way one would program a general-purpose digital computer: by writing text-based code that is then translated into the "machine code" instructions native to the computer's processor unit. Writing code for anything but trivial applications requires a fairly high level of proficiency, and so microcontrollers are only useful as one's coding skill.

A similar technology invented specifically for industrial control systems, to be used by personnel familiar with electrical wiring but not necessarily coding, is the *programmable logic controller* or *PLC*. Like a microcontroller, a PLC offers connection terminals for discrete (on/off) devices, both for receiving information (input) and providing information (output). Unlike a microcontroller, a PLC is a much larger device more closely resembling a general-purpose computer system than a single IC "chip", and its programming language options usually include *ladder diagram* which is a graphical language intended to resemble hard-wired relay logic circuitry. PLCs were invented as a tool to enable the same benefits of computer-based customizable logic control in industrial applications for people relatively unskilled in traditional coding.

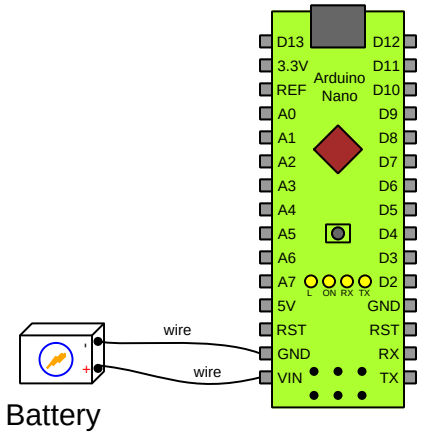
Both microcontrollers and PLCs are ubiquitous in our modern society. Most adults living in the United States of America own dozens of microcontrollers, embedded in the larger systems and devices we take for granted such as automobiles, kitchen appliances, and personal electronic devices. Those same people live and work in environments often controlled by PLCs, with PLC-controlled lights directing automobile traffic in their cities, PLC-controlled water and gas pipeline networks delivering those commodities, PLC-controlled elevators used in tall buildings, and PLC-controlled factories producing nearly everything they consume.

Each of these technologies is complex enough that a person could build an entire career becoming an expert in either one. This Tutorial is a gentle introduction to both, and the reader is advised to consult other resources to gain a fuller understanding of either technology.

A highly popular model of microcontroller designed for hobbyist and beginning student use is the *Arduino*. A photograph of an Arduino "Nano" model appears below, designed to plug directly into a solderless breadboard for convenient connection to other electronic components:



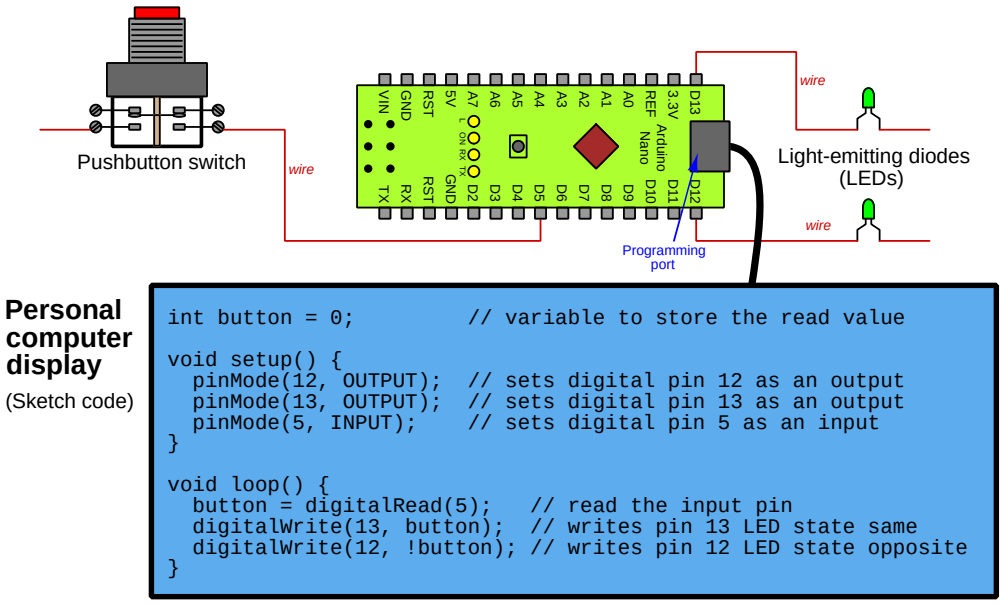
Each metal terminal (or "pin") on the Arduino has a designated function or set of possible functions, most of them for input or output (I/O) where the microcontroller either receives or sends digital logic (high/low) voltage signals with respect to ground. Here we see a pictorial illustration of an Arduino Nano with each pin labeled, powered by a 9-Volt battery:



A USB port on the Arduino provides connection to a personal computer, where development software runs to allow the user to write code and download it into the microcontroller's memory. Once programmed and connected to a suitable source of electrical power, the microcontroller will then execute its program independent of any other computer.

In a very real sense the aim of the Arduino (and other hobbyist-grade devices such as the BASIC Stamp, PICAXE, OOPic) is to make microcontrollers more accessible to people lacking strong coding skills. These educational-level devices use coding that is easier to understand than traditional programming languages and geared toward direct logical control of real-world devices, albeit with some limitations. Thus, the Arduino is to professional-grade microcontrollers as the PLC is to general-purpose computers.

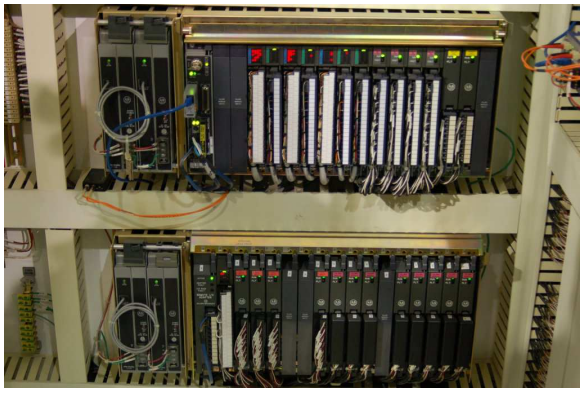
Here we see a partial wiring diagram and Sketch-language code causing an Arduino Nano microcontroller to control the states of two light-emitting diodes (LEDs) when a pushbutton switch is pressed and released:



All code contained within the `setup` section (those lines of text between the { and } brace characters) instructs the microcontroller to use pins 12 and 13 as outputs (to control the LEDs), and pin 5 as an input (from the switch). All code contained within the `loop` section is what the microcontroller repeatedly executes, in this program reading the digital status of input pin 5 and using that status value to drive the two outputs (pins 12 and 13) oppositely of each other. All code to the right of the double-slash characters (`//`) functions as *comments* to make the code easier for human readers to understand, but is ignored by the microcontroller itself.

The microcontroller's processor executes the text-based code in the same order as an English reader would read a printed document: from left to right, top to bottom, and it does so at a blindingly fast pace. This speed of execution is a function of both the processor's clock frequency and the number of clock cycles necessary to execute each instruction, but suffice it to say that the microcontroller is able to execute any program *much* faster than any human can read the code.

PLCs tend to be much larger in physical size than microcontrollers, and of sufficiently rugged construction to permit direct industrial use. Like microcontrollers, PLCs excel at applications receiving data from sensors and outputting commands to indicator and actuator hardware. Below we see one model of PLC called the Allen-Bradley PLC-5 equipped with *dozens* of I/O channels. Each of the two metal racks is approximately 60 centimeters in width, holding as many as 16 I/O "card" modules each:

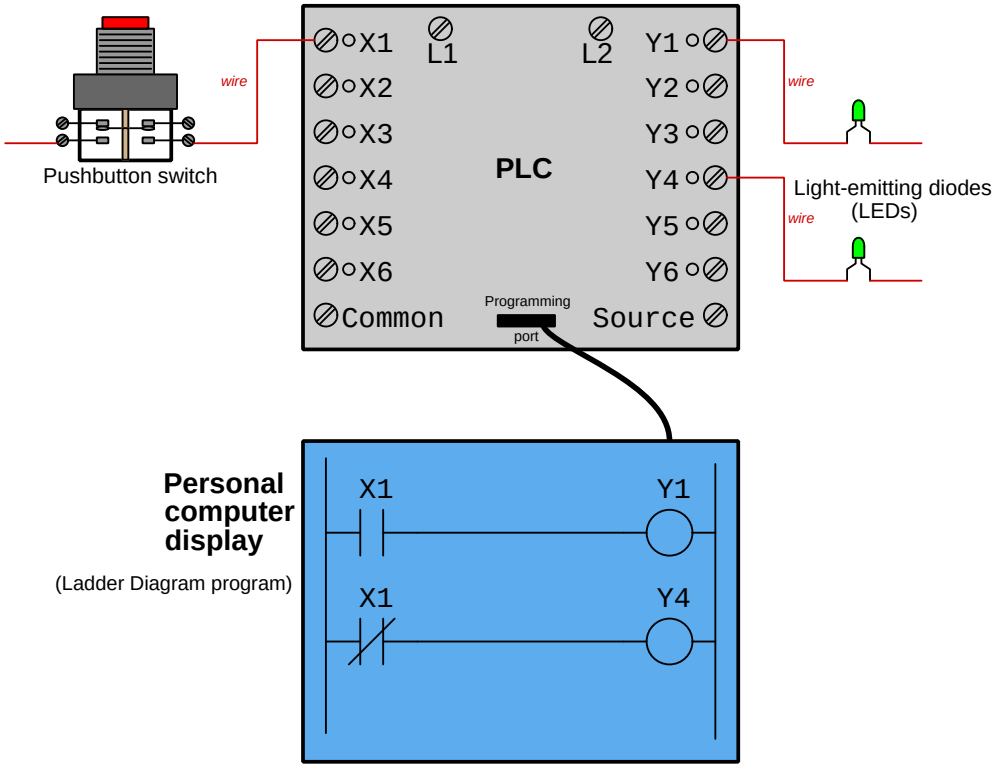


One interesting design feature of most PLCs is the modularity of their I/O. Rather than dedicated pins on an IC serving as input and output channels, PLCs such as this Allen-Bradley PLC-5 support add-on hardware in the form of "cards" that plug into a backplane. These cards may be replaced, upgraded, or swapped with cards having different functionality, all while using the same processor unit to store and run the logic program.

PLCs are also self-contained computers in their own right, able to function without connection to any other computer. Like a microcontroller, though, programming a PLC requires the use of a personal computer to initially download the program.

The most popular programming language for PLCs is *Ladder Diagram*, a graphical language designed to resemble an electrical wiring diagram. This interesting choice of programming language stems from the history of the PLC as a device designed to replace electro-mechanical relay control circuits, where the first people tasked with programming the PLCs were industrial electricians accustomed to interpreting complex wiring diagrams.

Here we see a partial wiring diagram and ladder-diagram code causing a PLC to control the states of two light-emitting diodes (LEDs) when a pushbutton switch is pressed and released, much like the Arduino Nano diagram and code listing shown earlier:



The "ladder-diagram" code is supposed to mimic the electrical wiring diagram of a switch-and-relay circuit, with vertical line-pairs symbolizing virtual switches and circles symbolizing output (write) instructions. Each virtual-switch symbol is a read instruction, either passing or blocking virtual electric power to the write instructions. PLC programming software shows the states of each symbol using color highlighting, kind of like debugging software is able to show the states of variables within text-based code.

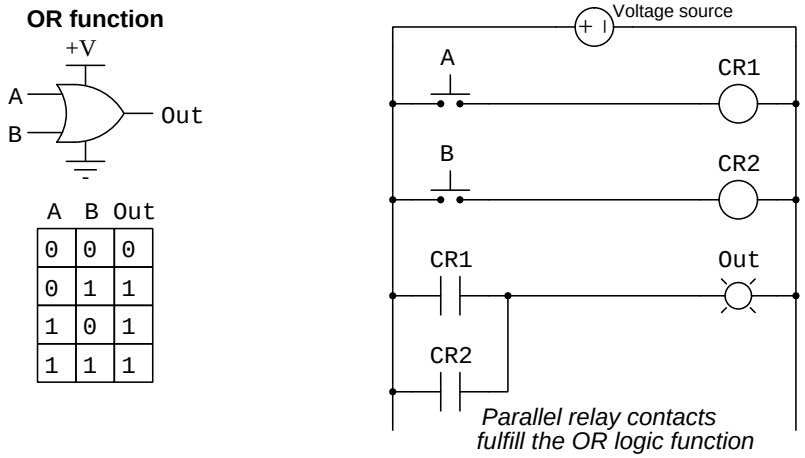
All "rungs" of code in a ladder diagram PLC program get executed repeatedly, just like the text-based code within an Arduino's `loop` section. The "scan order" for most PLC ladder diagram languages is left to right, top to bottom, in a manner similar to the execution of text-based code, and just like microcontrollers the speed of execution boggles the human mind.

The following sections explore basic microcontroller and PLC functionality through practical examples. All examples shown use generic microcontroller/PLC hardware and programming languages, specific to none but generally applicable to all. For the microcontroller examples in particular, every program will be shown in a non-specific programming language called *pseudocode* consisting of written English statements declaring the intent of each line, potentially translatable into actual code while not actually being consistent with any programming language. If you are interested in specific wiring/programming examples, the Case Tutorial chapter contains verbatim code written in standardized programming languages and with literal wiring diagrams.

You will find that learning to program any particular model of microcontroller or PLC is a process born of practice, and of frequent reference to the device's technical documentation. This Tutorial will get you started in understanding basic principles of microcontroller and PLC usage, but the details of any particular device's wiring and programming is best left to manufacturer documentation.

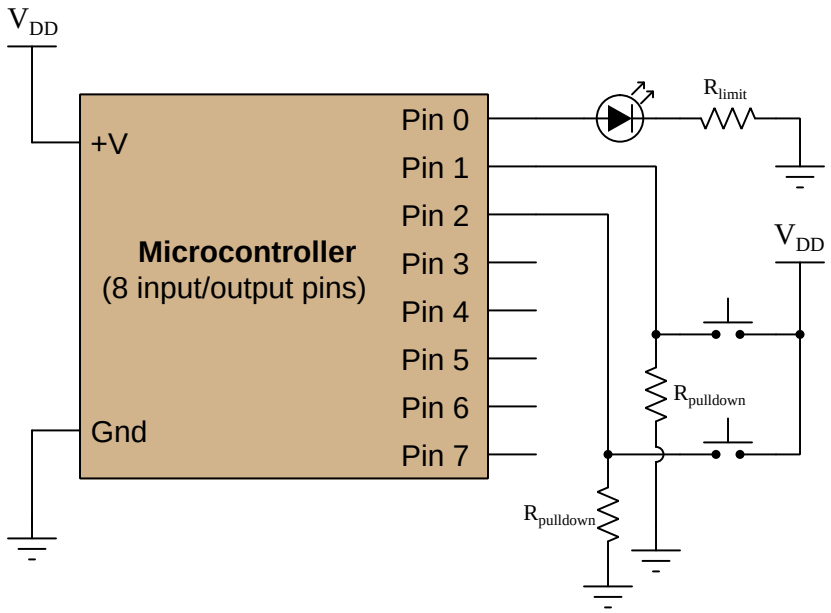
3.2 Logical OR function

The following schematic diagrams show how it is possible to implement an elementary two-input OR function directly with hardware, using both a semiconductor logic gate (left) and electromechanical relay circuit (right) drawn using standard industrial "ladder diagram" convention:



If either input A or input B is high (active), the output becomes high (active) as well.

Next, we see how to implement the same logical OR function using a microcontroller:

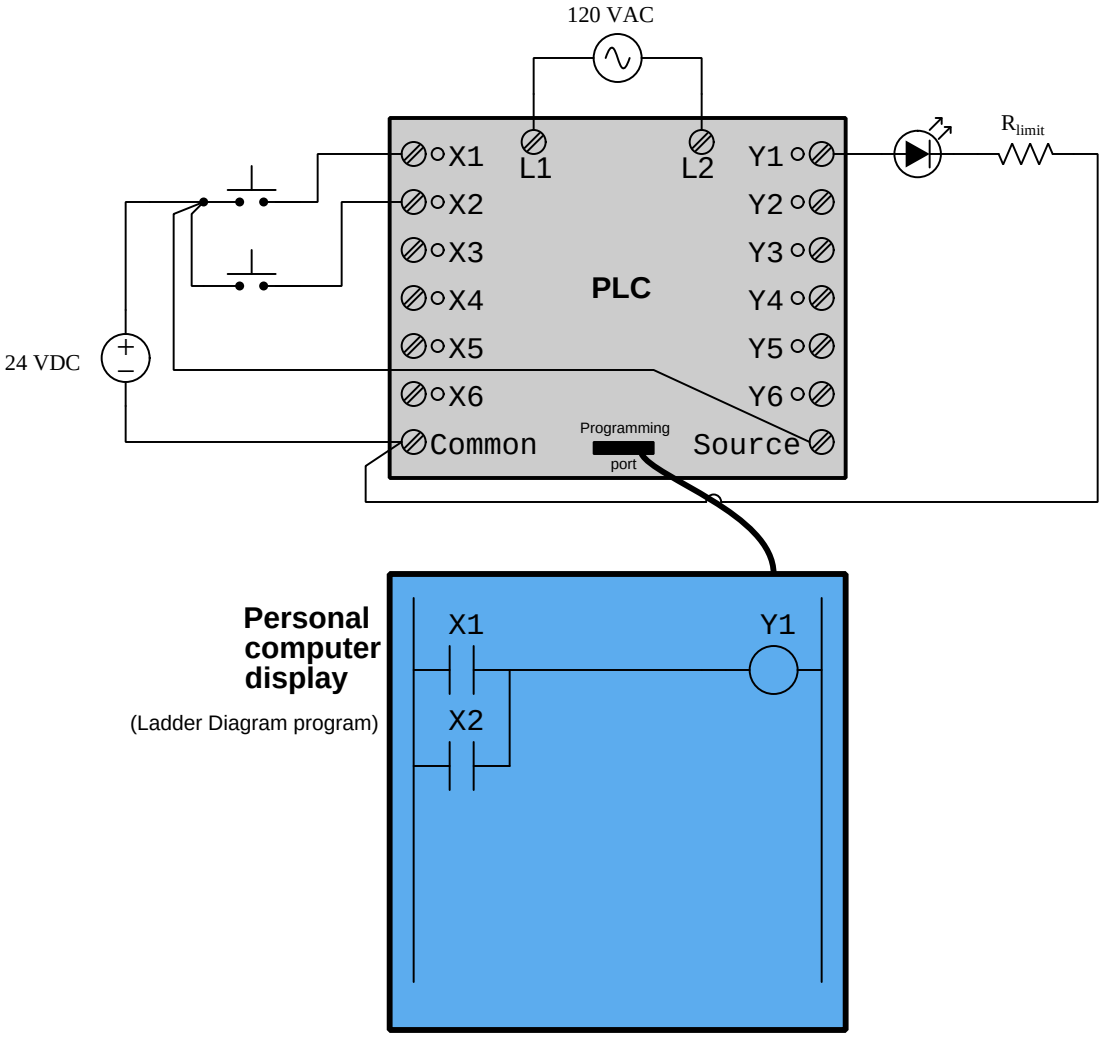


Pseudocode listing

```
Declare Pin0 as an output
Declare Pin1 and Pin2 as inputs

LOOP
  IF Pin1 is HIGH, set Pin0 HIGH
  ELSEIF Pin2 is HIGH, set Pin0 HIGH
  ELSE set Pin0 LOW
  ENDIF
ENDLOOP
```

Here, the IF/ENDIF structure identifies and acts on three possible conditions: (1) `Pin1` is high, (2) `Pin2` is high, or (3) neither is high. Only if both input pins are low will the output (`Pin0`) go to a low state. Thus, this microcontroller program fulfills the basic OR function by forcing the output high if any input happens to be high.

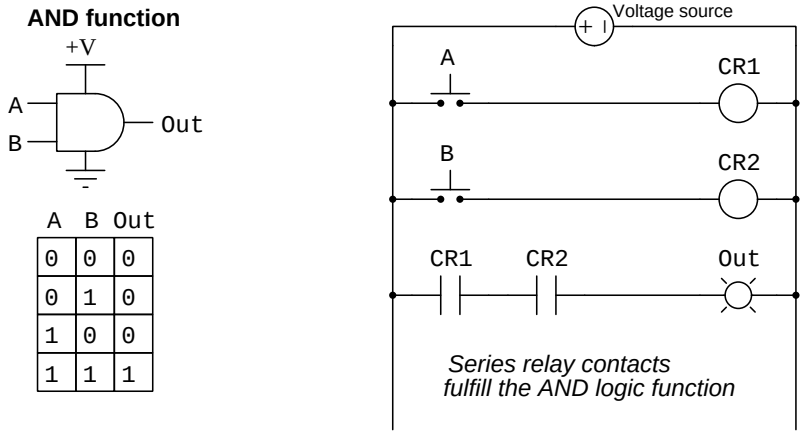
Next, we see how to implement the same logical OR function using a PLC:



The virtual ladder-diagram "circuit" has two parallel-connected "contacts", each one actuated by its respective input terminal's logic state. Being normally-open, these virtual contacts are open in their resting states (i.e. when its respective `X` input is logically "low" or de-energized) and closed when their respective input terminals go to a "high" logical state. If either or both of these paralleled contacts close, they send virtual "electricity" to the `Y1` "coil" which in turn will make a real electrical connection form between the `Source` and `Y1` terminals to energize the LED.

3.3 Logical AND function

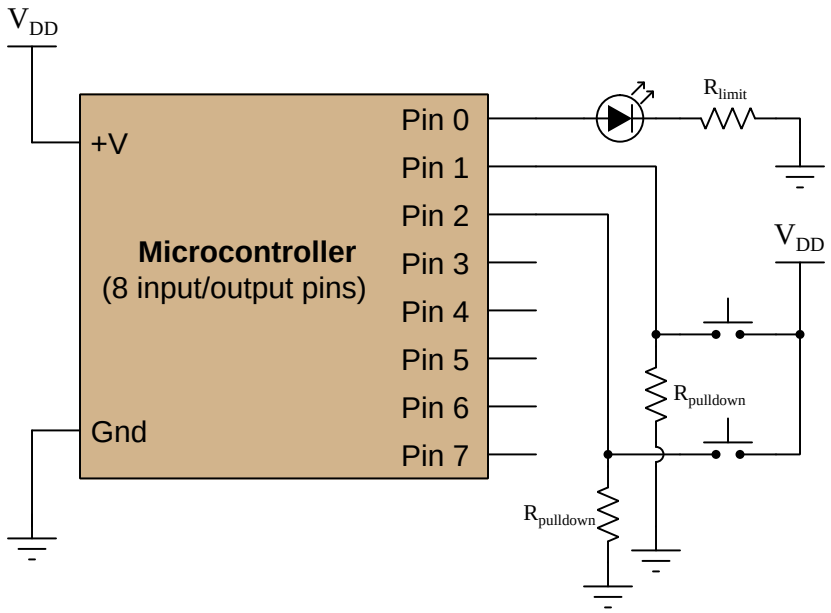
The following schematic diagrams show how it is possible to implement an elementary two-input AND function directly with hardware, using both a semiconductor logic gate (left) and electromechanical relay circuit (right) drawn using standard industrial "ladder diagram" convention:



A	B	Out
0	0	0
0	1	0
1	0	0
1	1	1

Both input A and input B must go to a "high" state (active) to make output "high" (active).

Next, we see how to implement the same logical AND function using a microcontroller:

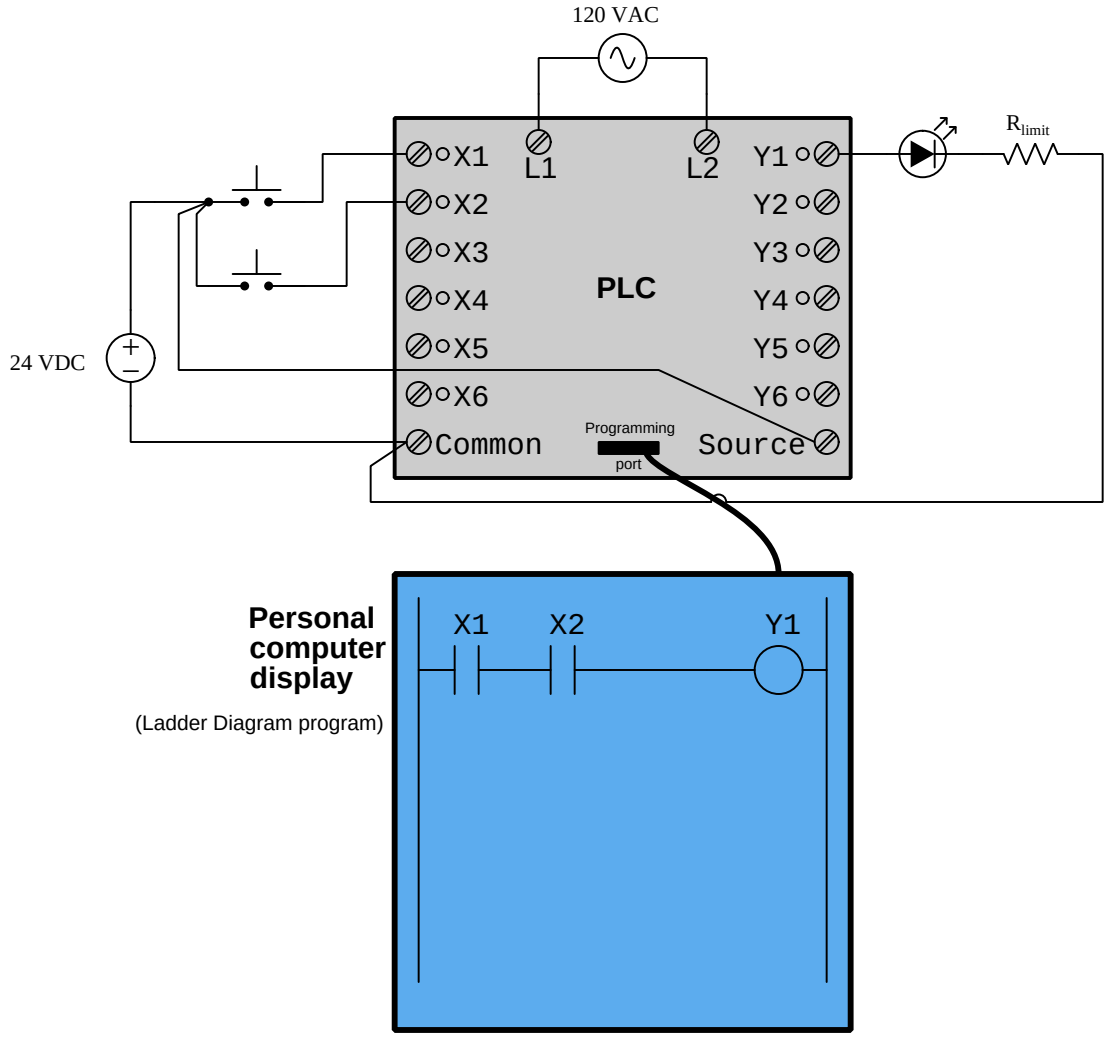


Pseudocode listing

```
Declare Pin0 as an output
Declare Pin1 and Pin2 as inputs

LOOP
  IF Pin1 is LOW, set Pin0 LOW
  ELSEIF Pin2 is LOW, set Pin0 LOW
  ELSE set Pin0 HIGH
  ENDIF
ENDLOOP
```

Here, the IF/ENDIF structure identifies and acts on three possible conditions: (1) `Pin1` is low, (2) `Pin2` is low, or (3) neither is low. Only if both input pins are high will the output (`Pin0`) go to a high state. Thus, this microcontroller program fulfills the basic AND function by forcing the output low if any input happens to be low.

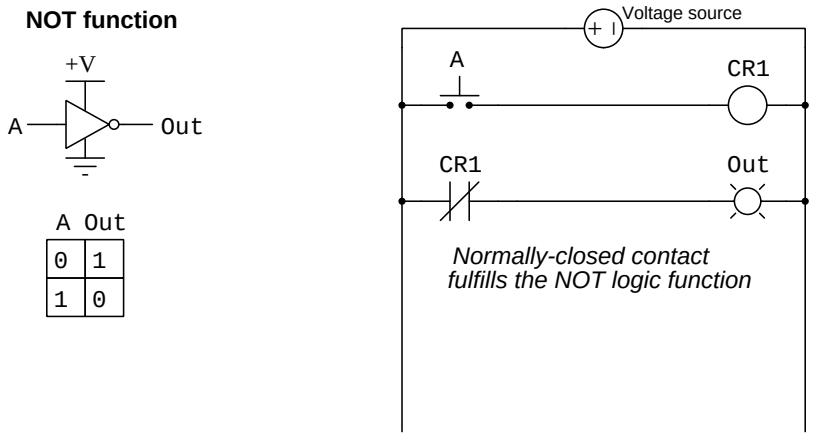
Next, we see how to implement the same logical AND function using a PLC:



The virtual ladder-diagram "circuit" has two series-connected "contacts", each one actuated by its respective input terminal's logic state. Being normally-open, these virtual contacts are open in their resting states (i.e. when its respective `X` input is logically "low" or de-energized) and closed when their respective input terminals go to a "high" logical state. If both of these series-connected contacts close, they send virtual "electricity" to the `Y1` "coil" which in turn will make a real electrical connection form between the `Source` and `Y1` terminals to energize the LED.

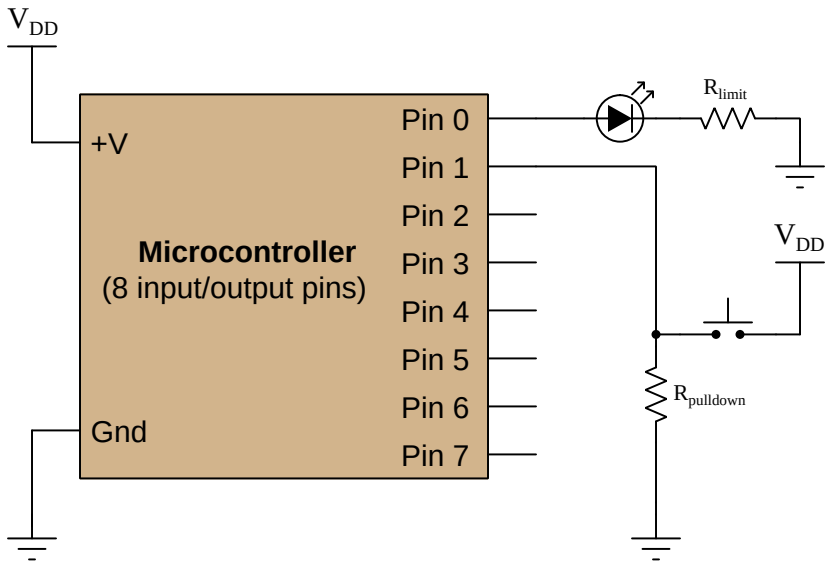
3.4 Logical NOT function

The following schematic diagrams show how it is possible to implement an elementary inverter (NOT) function directly with hardware, using both a semiconductor logic gate (left) and electromechanical relay circuit (right) drawn using standard industrial "ladder diagram" convention:



In these circuits the output state is exactly opposite of the input, the NOT function *complementing* the input's state.

Next, we see how to implement the same logical NOT function using a microcontroller:

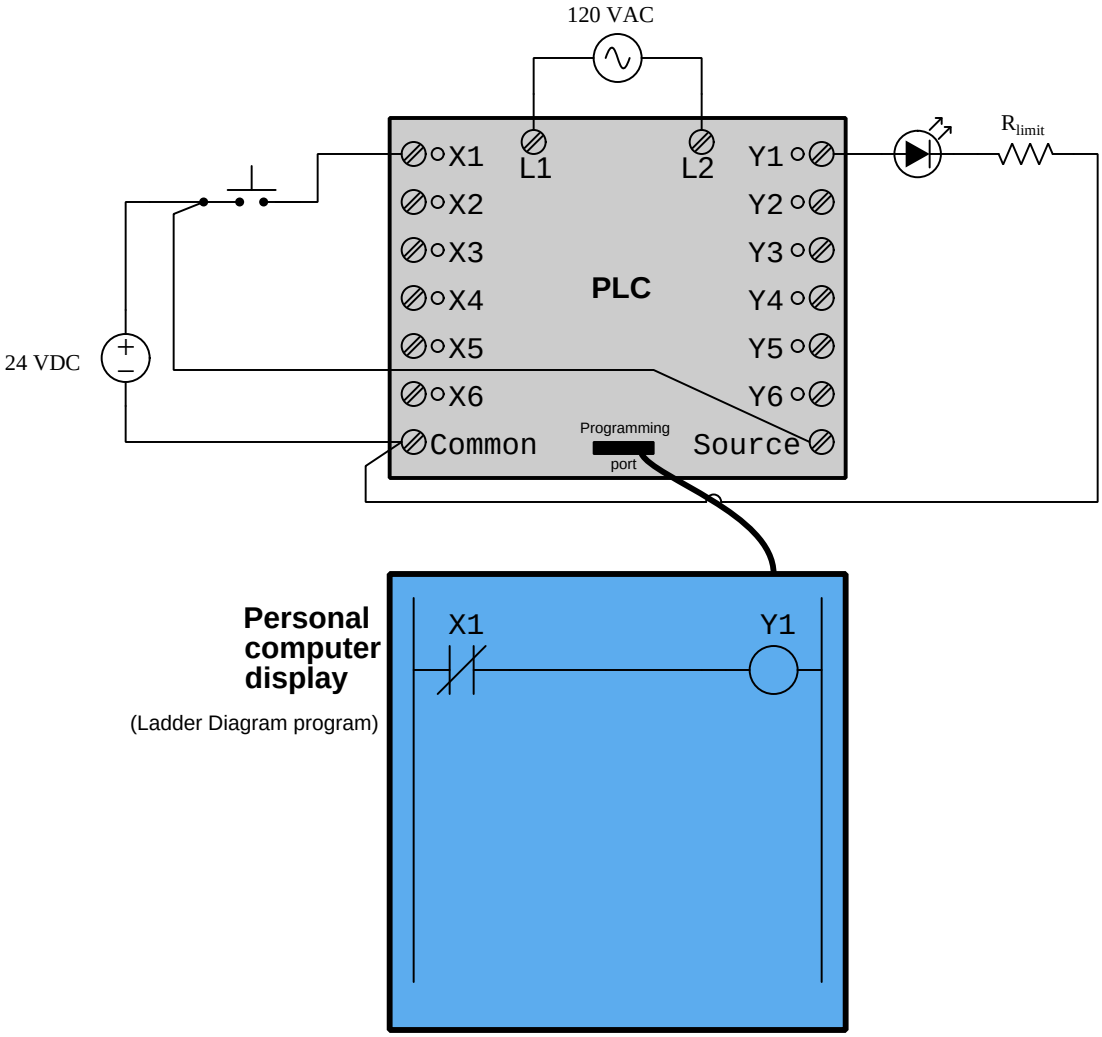


Pseudocode listing

```
Declare Pin0 as an output
Declare Pin1 as an input

LOOP
  IF Pin1 is LOW, set Pin0 HIGH
  ELSE set Pin0 LOW
  ENDIF
ENDLOOP
```

Here, the IF/ENDIF structure identifies and acts on only two possible conditions: (1) `Pin1` is low, or (2) `Pin1` is high. In either case the output (`Pin0`) gets assigned to the opposite state. Thus, this microcontroller program fulfills the basic NOT function by continually complementing the input's state.

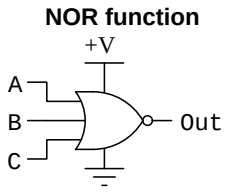
Next, we see how to implement the same logical NOT function using a PLC:



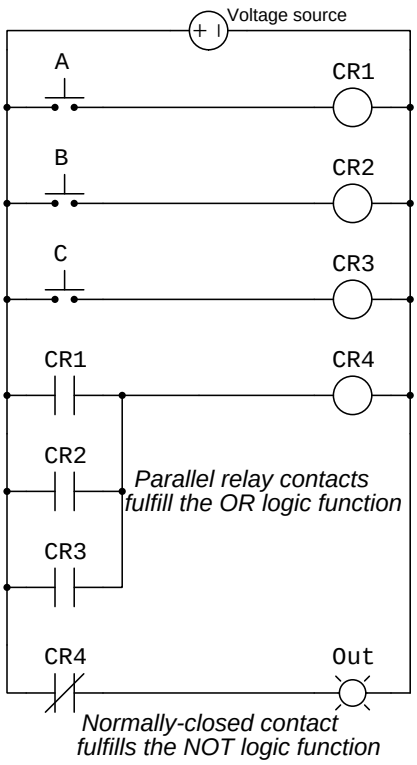
The virtual ladder-diagram "circuit" has just one "contact", and it is normally-closed which means is passes virtual "electricity" to output coil `Y1` when in its resting state, which will be when input `X1` is de-energized. If someone presses the pushbutton to energize input `X1`, the virtual contact `X1` will be forced "open" which prevents virtual electricity from reaching coil `Y1`, thereby de-energizing the LED.

3.5 Logical NOR function

The following schematic diagrams show how it is possible to implement an elementary three-input NOR function directly with hardware, using both a semiconductor logic gate (left) and electromechanical relay circuit (right) drawn using standard industrial "ladder diagram" convention:

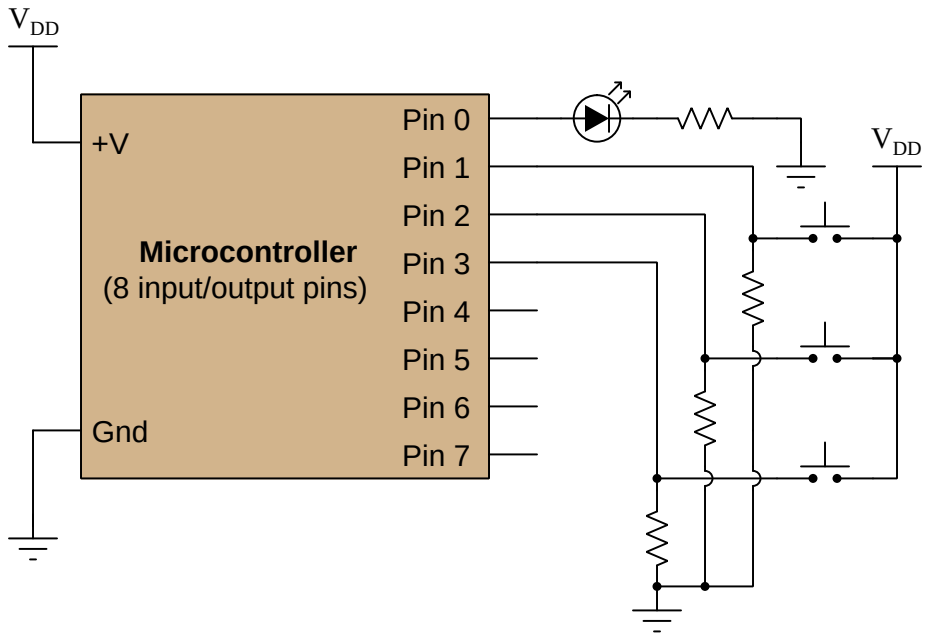


A	B	C	Out
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0



If any of the three inputs are high (active), the output goes low.

Next, we see how to implement the same logical NOR function using a microcontroller:

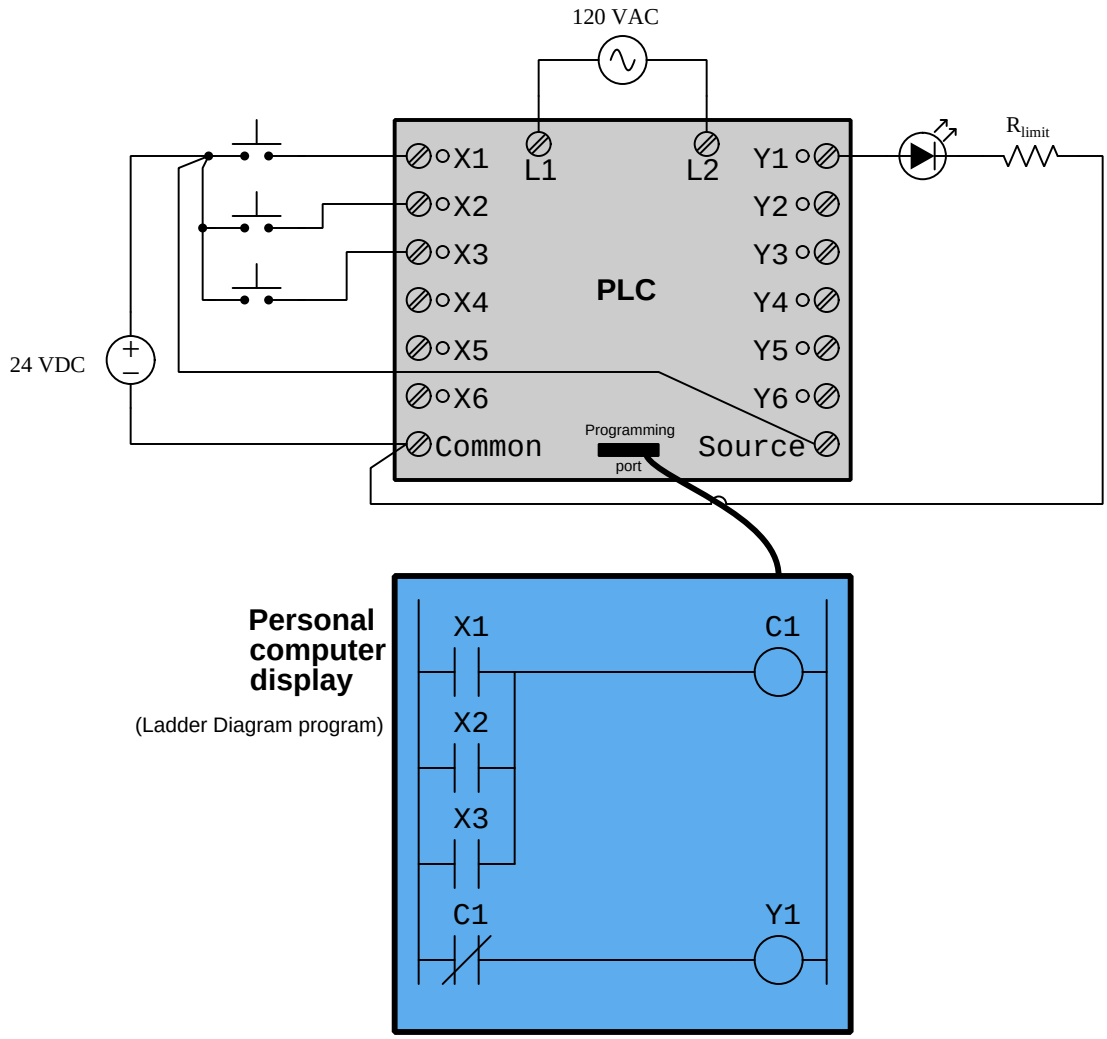


Pseudocode listing

```
Declare Pin0 as an output
Declare Pin1 and Pin2 and Pin3 as inputs

LOOP
  IF Pin1 is HIGH, set Pin0 LOW
  ELSEIF Pin2 is HIGH, set Pin0 LOW
  ELSEIF Pin3 is HIGH, set Pin0 LOW
  ELSE set Pin0 HIGH
  ENDIF
ENDLOOP
```

Here, the IF/ENDIF structure identifies and acts on four possible conditions: (1) `Pin1` is high, (2) `Pin2` is high, (3) `Pin3` is high, or (4) none of them are high. Only if all input pins are low will the output (`Pin0`) go to a high state. Thus, this microcontroller program fulfills the basic NOR function by forcing the output low if any input happens to be high.

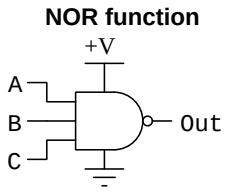
Next, we see how to implement the same logical NOR function using a PLC:



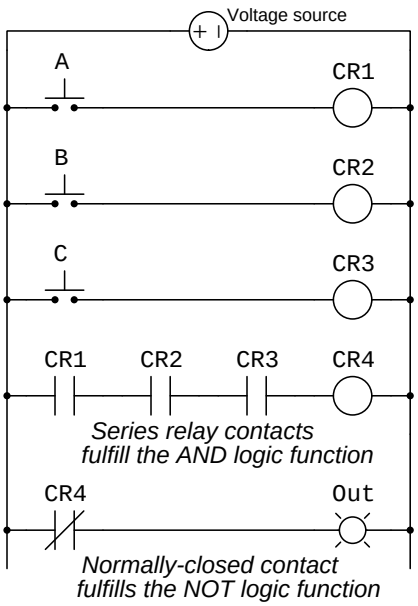
The virtual ladder-diagram "circuit" has three parallel-connected "contacts", each one actuated by its respective input terminal's logic state. Being normally-open, these virtual contacts are open in their resting states (i.e. when its respective `X` input is logically "low" or de-energized) and closed when their respective input terminals go to a "high" logical state. If any of these paralleled contacts close, they send virtual "electricity" to the PLC's internal `C1` "coil" which has no effect on the external world but only sets a bit in the PLC's memory. This bit, however, controls the status of normally-closed "contact" `C1` which in turn controls "coil" `Y1` having real control over the LED's status.

3.6 Logical NAND function

The following schematic diagrams show how it is possible to implement an elementary three-input NAND function directly with hardware, using both a semiconductor logic gate (left) and electromechanical relay circuit (right) drawn using standard industrial "ladder diagram" convention:

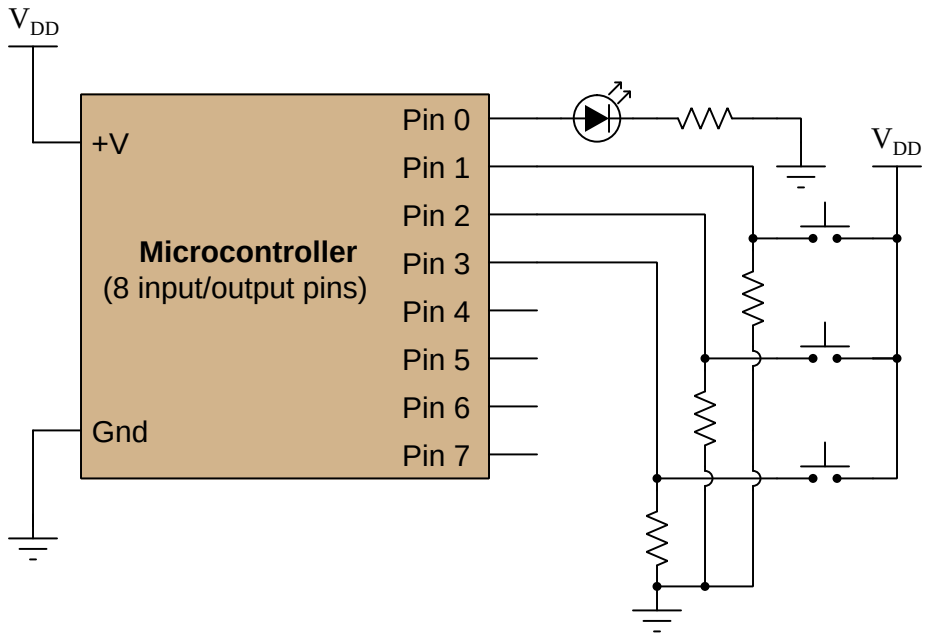


A	B	C	Out
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0



If any of the three inputs are low, the output goes high.

Next, we see how to implement the same logical NAND function using a microcontroller:

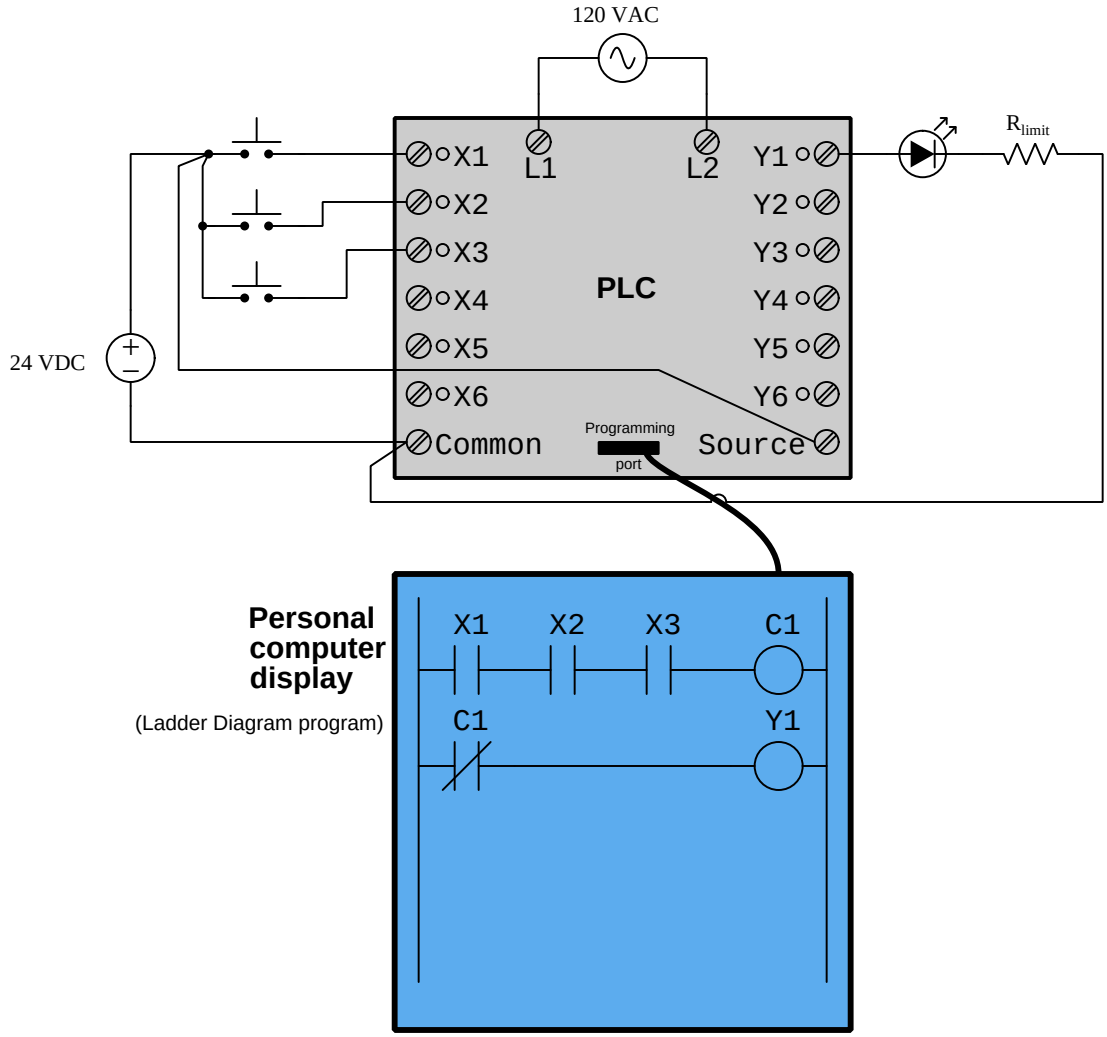


Pseudocode listing

```
Declare Pin0 as an output
Declare Pin1 and Pin2 and Pin3 as inputs

LOOP
  IF Pin1 is LOW, set Pin0 HIGH
  ELSEIF Pin2 is LOW, set Pin0 HIGH
  ELSEIF Pin3 is LOW, set Pin0 HIGH
  ELSE set Pin0 LOW
  ENDIF
ENDLOOP
```

Here, the IF/ENDIF structure identifies and acts on four possible conditions: (1) `Pin1` is low, (2) `Pin2` is low, (3) `Pin3` is low, or (4) none of them are low. Only if all input pins are high will the output (`Pin0`) go to a low state. Thus, this microcontroller program fulfills the basic NAND function by forcing the output high if any input happens to be low.

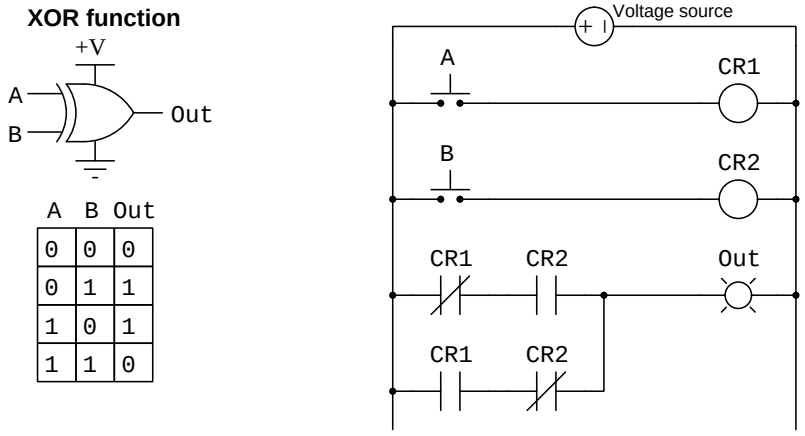
Next, we see how to implement the same logical NAND function using a PLC:



The virtual ladder-diagram "circuit" has three series-connected "contacts", each one actuated by its respective input terminal's logic state. Being normally-open, these virtual contacts are open in their resting states (i.e. when its respective `X` input is logically "low" or de-energized) and closed when their respective input terminals go to a "high" logical state. If all of these paralleled contacts close, they send virtual "electricity" to the PLC's internal `C1` "coil" which has no effect on the external world but only sets a bit in the PLC's memory. This bit, however, controls the status of normally-closed "contact" `C1` which in turn controls "coil" `Y1` having real control over the LED's status.

3.7 Logical XOR function

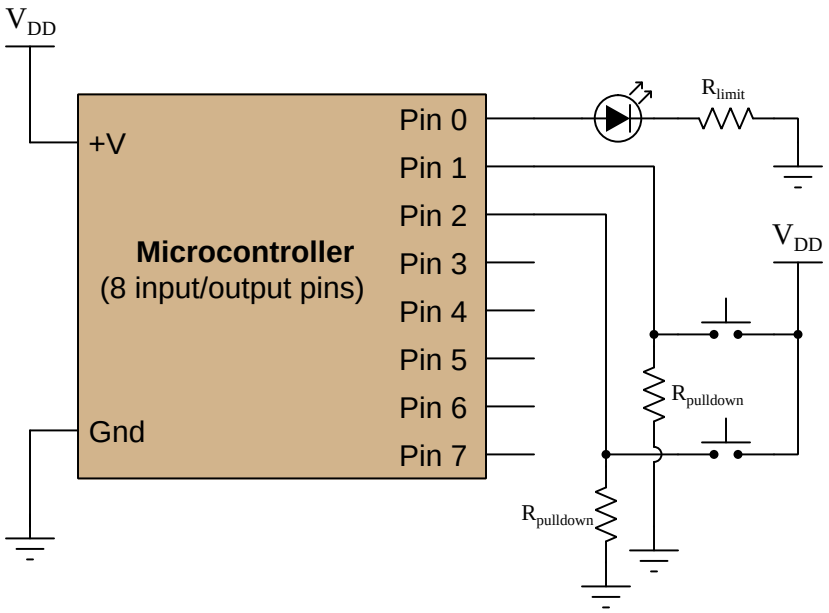
The following schematic diagrams show how it is possible to implement a two-input Exclusive-OR (XOR) function directly with hardware, using both a semiconductor logic gate (left) and electromechanical relay circuit (right) drawn using standard industrial "ladder diagram" convention:



A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

If the logical states of input A and input B differ, the output goes high; otherwise the output will be low.

Next, we see how to implement the same logical XOR function using a microcontroller:

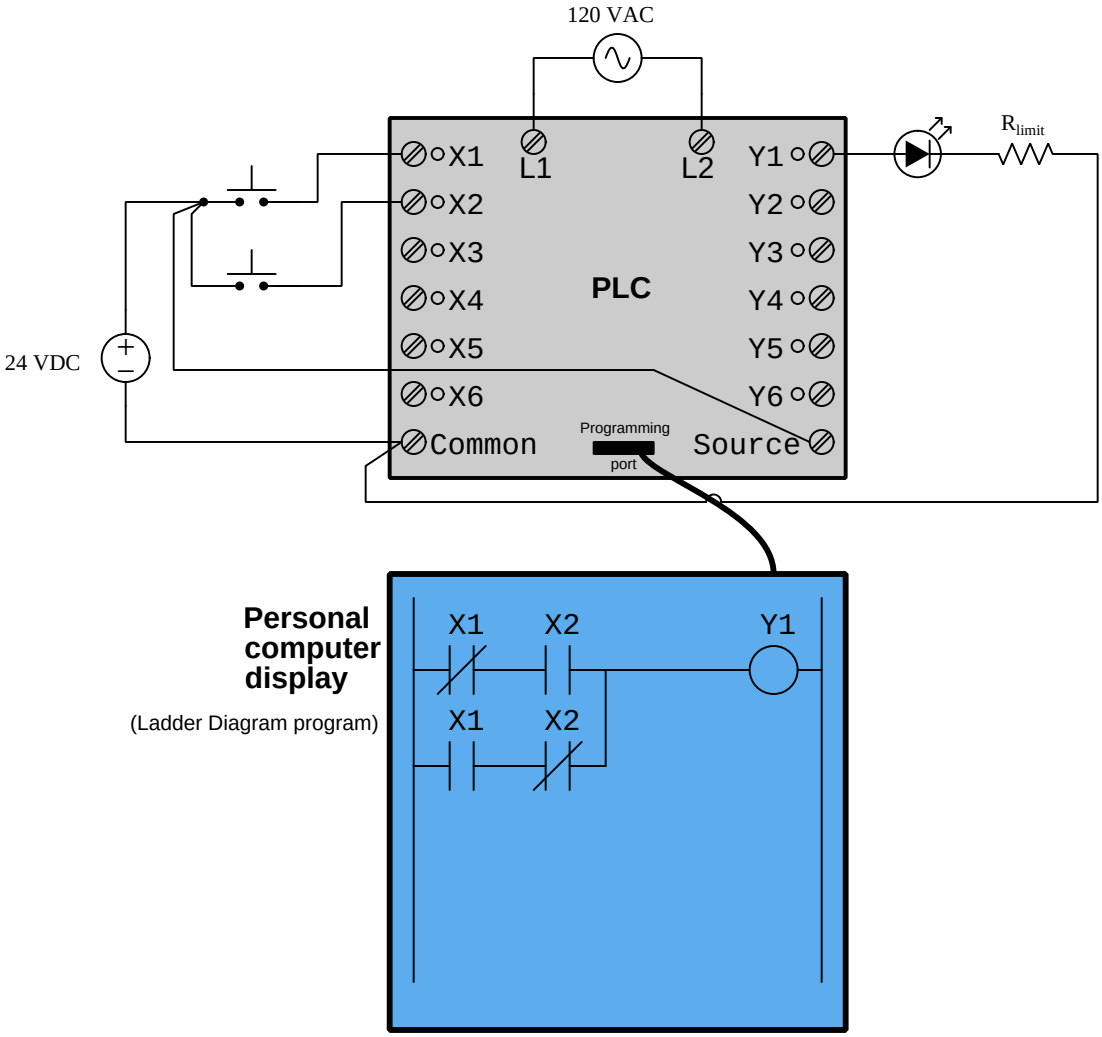


Pseudocode listing

```
Declare Pin0 as an output
Declare Pin1 and Pin2 as inputs

LOOP
  IF Pin1 is LOW and Pin2 is HIGH, set Pin0 HIGH
  ELSEIF Pin1 is HIGH and Pin2 is LOW, set Pin0 HIGH
  ELSE set Pin0 LOW
  ENDIF
ENDLOOP
```

Here, the IF/ENDIF structure identifies and acts on three possible conditions: (1) `Pin1` is high and `Pin2` is low, (2) `Pin1` is low and `Pin2` is high, or (3) any other condition. Only if one of the first two conditions is met (i.e. the inputs differ in state) will the output (`Pin0`) go to a high state.

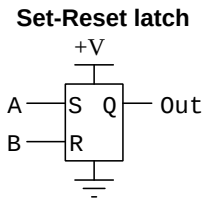
Next, we see how to implement the same logical XOR function using a PLC:



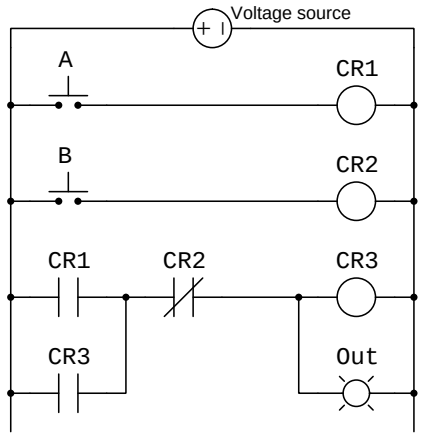
Like some of the previous ladder-diagram PLC programs, this one uses both normally-open and normally-closed virtual "contacts", each one actuated by its respective input terminal's logic state. Note how we may use multiple "contacts" per input, all of them actuated by the same input logic state. Each normally-open "contact" is open and each normally-closed "contact" is closed when its input is low; those states changing when the input becomes high. Thus, the upper rung of this PLC program passes virtual "electricity" when `X1` is low and `X2` is high, while the lower run passes virtual "electricity" when `X1` is high and `X2` is low. If either of these paralleled rungs become "conductive", they send virtual "electricity" to the `Y1` "coil" which in turn will make a real electrical connection form between the `Source` and `Y1` terminals to energize the LED.

3.8 Latching function

The following schematic diagrams show how it is possible to implement a latching function directly with hardware, using both a semiconductor Set-Reset latch (left) and electromechanical relay circuit (right) drawn using standard industrial "ladder diagram" convention:

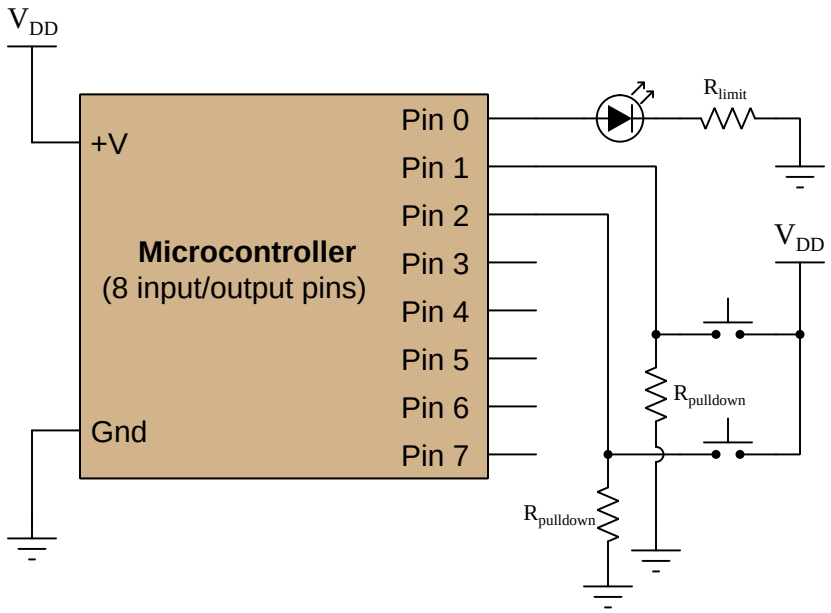


A	B	Out
0	0	*Latch*
0	1	0
1	0	1
1	1	0



Input A acts to "set" the latch circuit and activate its output. Input B acts to do the opposite, "resetting" the latch and de-activating its output. When neither input A nor input B is activated, the latch circuit holds in its prior state – in other words, it *latches* to that prior state. In both of these circuit examples, input B has priority over input A, which means if both inputs are activated simultaneously, the circuit resets.

Next, we see how to implement the same logical latch function using a microcontroller:

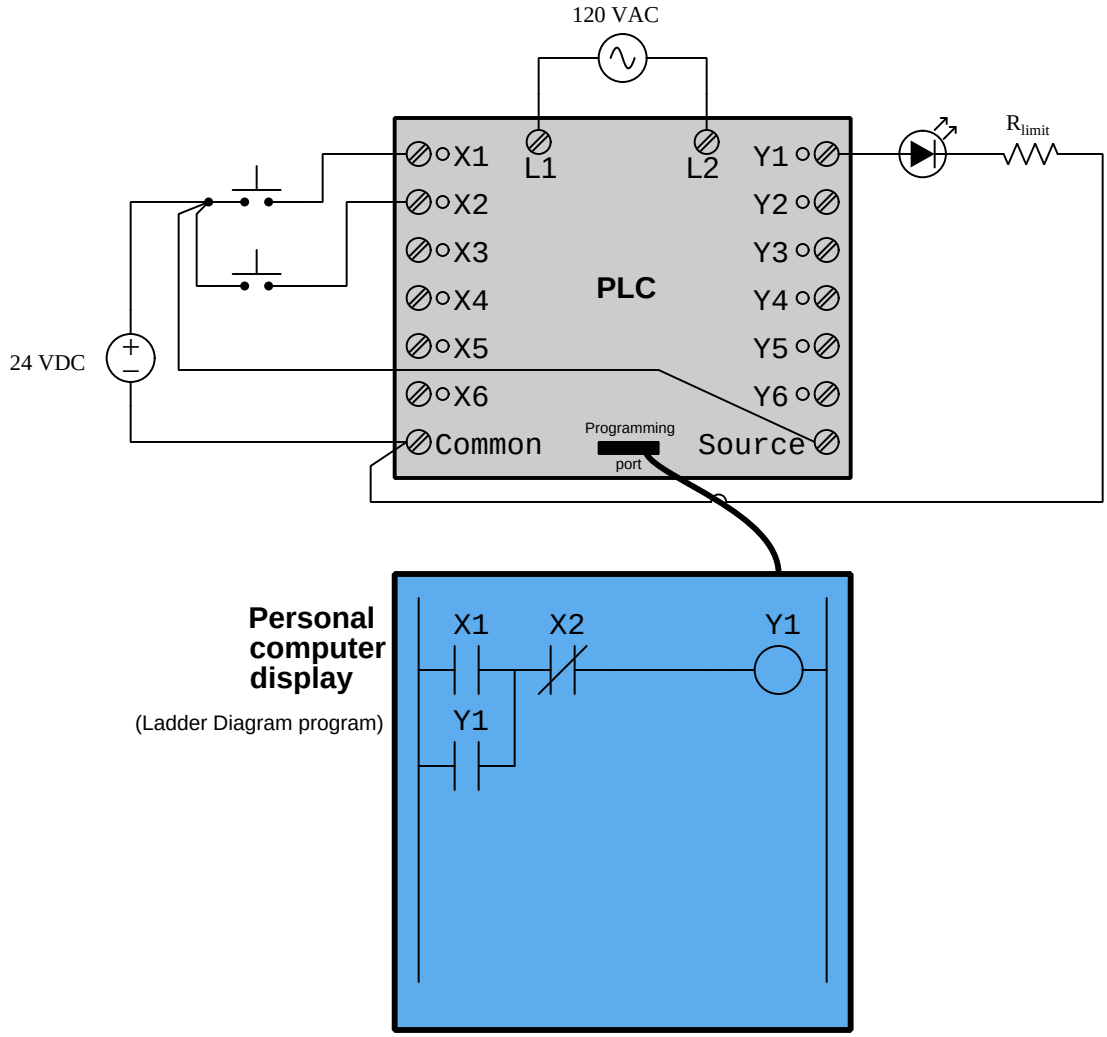


Pseudocode listing

```
Declare Pin0 as an output
Declare Pin1 and Pin2 as inputs

LOOP
  IF Pin1 is HIGH, set Pin0 HIGH
  IF Pin2 is HIGH, set Pin0 LOW
ENDLOOP
```

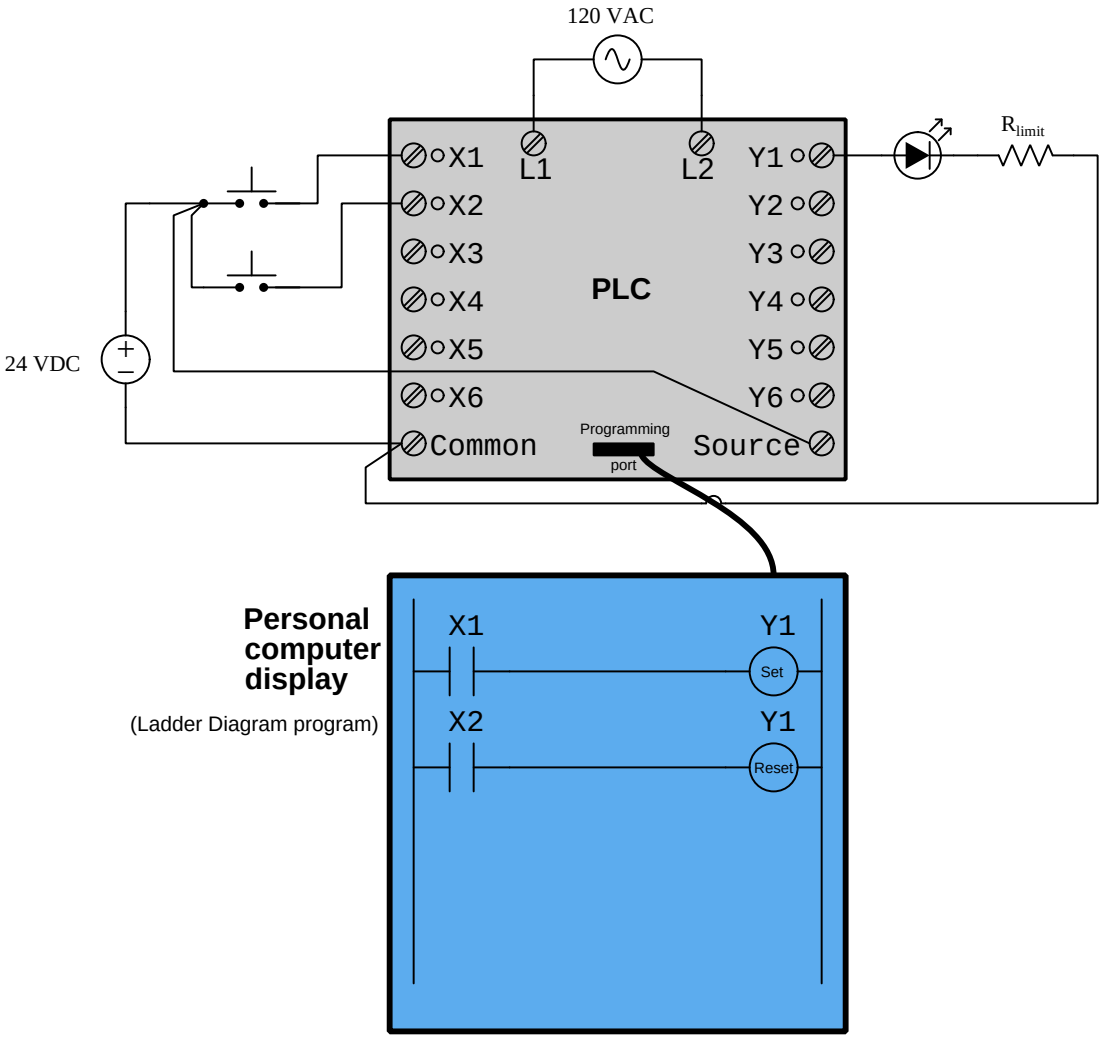
As we can see, this program is simplicity itself, because the retentive nature of microcontroller memory naturally implements the latch function. All we need to do is set the `Pin0` output to a "high" state if the Set input (`Pin1`) activates, and reset that same bit in memory (`Pin0`) if the Reset input (`Pin2`) activates. If neither `Pin1` nor `Pin2` input is active, neither of the `IF` conditional statements will evaluate as true, and the microcontroller will take no action on the state of `Pin0`. In the absence of any action writing new data to that `Pin0` bit, the microcontroller's memory naturally retains that bit's last state.

Next, we see how to implement the same logical latch function using a PLC:



We often associate PLC inputs (`X`) with contact instructions and PLC outputs (`Y`) with coil instructions, yet this particular program contains a contact instruction labeled with a `Y` (output) bit. A more accurate way to think of PLC contact and coil instructions is to see them as *read* and *write* instructions, respectively. The `X1` and `X2` contact instructions read the statuses of those inputs, while the `Y1` contact instruction reads the state last written to the `Y1` bit by the `Y1` coil instruction. Any time a "1" state gets written to `Y1`, the `Y1` contact instruction "closes" and *seals in* the flow of virtual electricity to the `Y1` coil instruction, at least until someone activates `X2` to interrupt that virtual flow. This "seal-in" action grants the program its latching behavior.

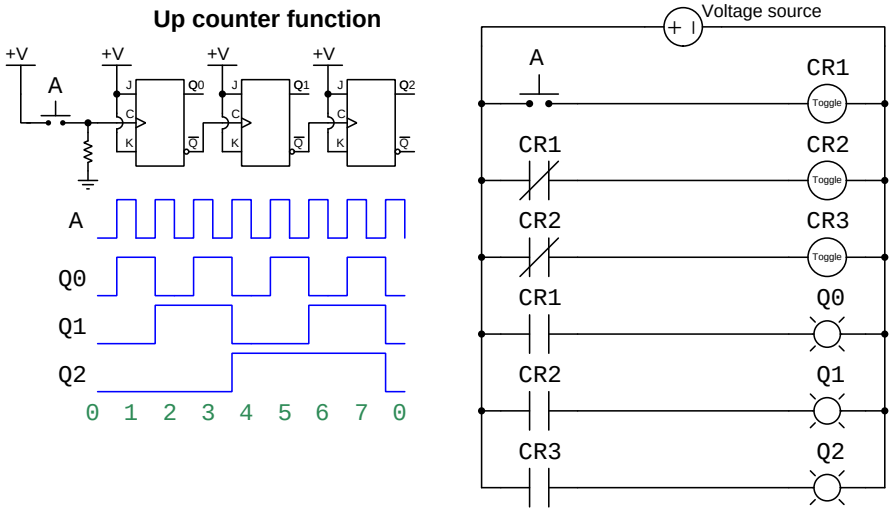
Standard coil instructions in PLCs constantly write data to their respective bits: when "powered" with virtual electricity they write "1" states; when "unpowered" they write "0" states. However, most PLCs additionally offer special types of coil instructions that only write when "powered" and do nothing when "unpowered". These are, quite appropriately, referred to as *set* coils and *reset* coils, respectively. A latching PLC program written using these special coil instructions is every bit as simple as the latching microcontroller program shown previously, as we show here:



3.9 Counter functions

Aside from logical functions such as AND, OR, XOR, and others, another class of functions useful for real-world control applications are *counters*. The first type of counter function we will explore is the *up* counter, which means activation of its input results in an incrementing of the output.

The following schematic diagrams show how it is possible to implement an up counter function directly with hardware, using both a semiconductor logic gate (left) and electromechanical relay circuit (right) drawn using standard industrial "ladder diagram" convention, the count sequence triggered by a signal marked "A" coming from a hand-actuated pushbutton switch:



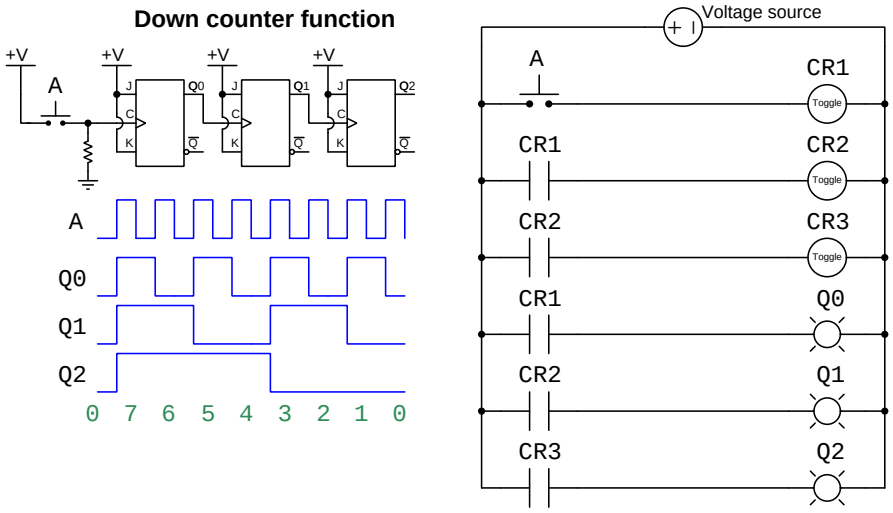
Semiconductor counter circuits utilize a type of device known as a *JK flip-flop* to produce a binary count sequence. Each JK flip-flop has two output lines, Q and $\overline{Q}$ which always exhibit opposite logical states. When both the J and K inputs are tied "high" (to the +V power supply rail), each voltage pulse applied at the "Clock" input causes the Q and $\overline{Q}$ output states to *toggle* or swap. A good "active learning" exercise is to examine the pulse diagram showing the input (A) signal and the resulting output (Q) signals, identifying where exactly in time each of the Q outputs changes state at the command of that flip-flop's incoming clock signal.

A special type of electromechanical relay called a *toggle relay* performs the same "toggle" function as the JK flip-flop, its contact(s) changing state with each new energization pulse of the controlling coil. Cascaded together like the three JK flip-flops shown, three toggle-style relays would also produce a three-bit binary count sequence. However, counting functions are more commonly implemented in electromechanical form using special *electromechanical counters* rather than by cascaded toggle relays. Electromechanical counters use solenoid-actuated ratchet-and-pawl mechanisms to increment the angle of a rotary shaft with each new electrical pulses energizing the counter's coil, that rotary shaft driving a set of display numerals showing an accumulated count value. The purpose of electromechanical counters is to provide a human-readable accumulated count value, such as the number of items passed by on a conveyor belt, etc.

When an "up" counter reaches its maximum count value, the next input pulse signal will typically cause the count value to *roll around* back to zero. For example, with the three-bit binary counters

shown above the count sequence would be 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, 2, 3, etc. Similarly, a five-digit electromechanical counter would begin at 00000 and count up to 99999 then back down to 00000 with the next pulse.

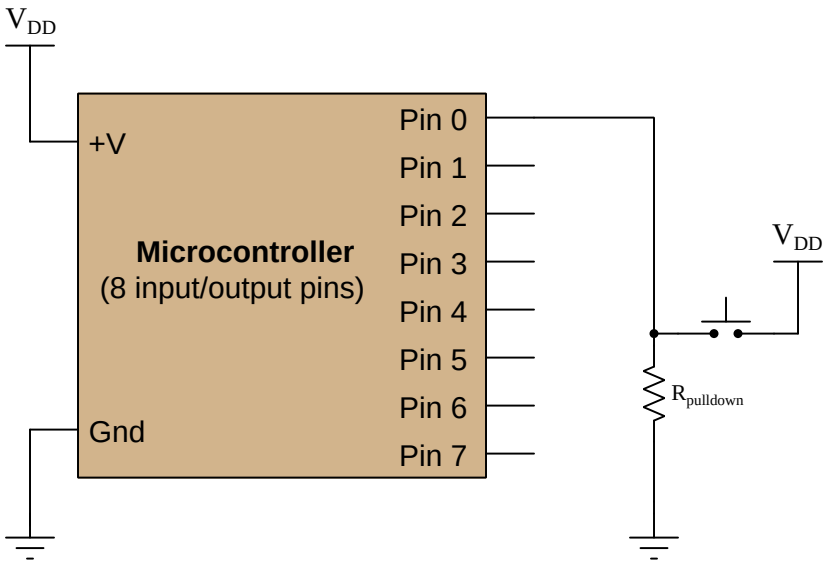
A variation on this concept is the *down counter*, where the numerical value decrements rather than increments with each new input signal pulse.



Also, there are *up-down counters* with two different inputs, either one to increment and another to decrement the accumulated count value, or one to trigger the count and another to command the direction of that count.

Each of the counter circuits explored in the previous pages trigger the count sequence to step whenever the input (pulse) signal transitions from a "low" state to a "high" state. In order for us to implement a counter function within a programmable device such as a microcontroller, we must write the program in such a way that it looks for a condition where the present value of the pulse input signal is "high" but the *last* value in the previous scan of the program was "low". A common method for doing this is to declare a variable within the program to store the last state of the input pin and then update that "last" variable at the end of each loop.

The following microcontroller program (written in pseudocode) shows how this works:

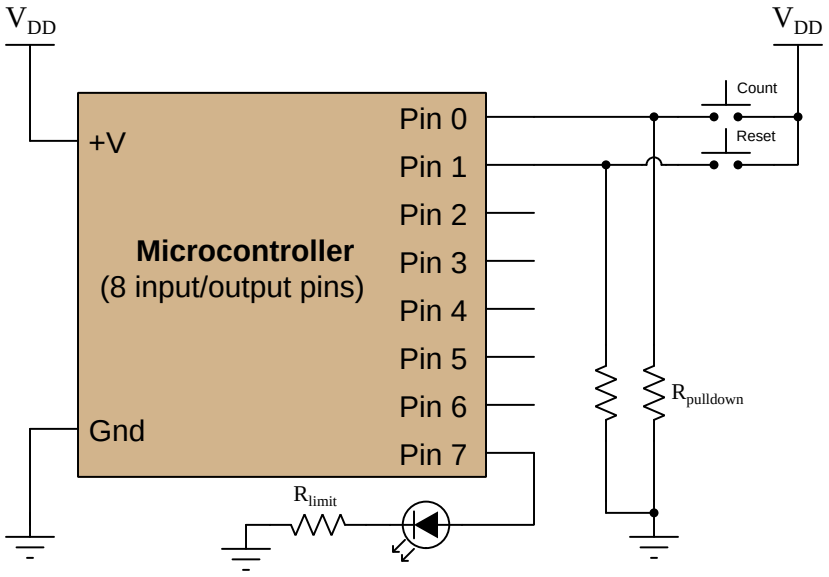


Pseudocode listing

```
Declare Pin0 as an input
Declare Last_Pin0 as a variable
Declare Count as a variable

LOOP
  IF Pin0 is HIGH and Last_Pin0 is LOW, add one to Count
  Set Last_Pin0 equal to the value of Pin0
ENDLOOP
```

This program's counting limit is bound only by the number of bits the microcontroller uses for the `Count` variable, with eight bits to thirty-two bits commonly offered in modern microcontrollers. An eight-bit binary integer's count range extends from 0 to 255, whereas a 32-bit binary integer is able to count from 0 to 4,294,967,295.

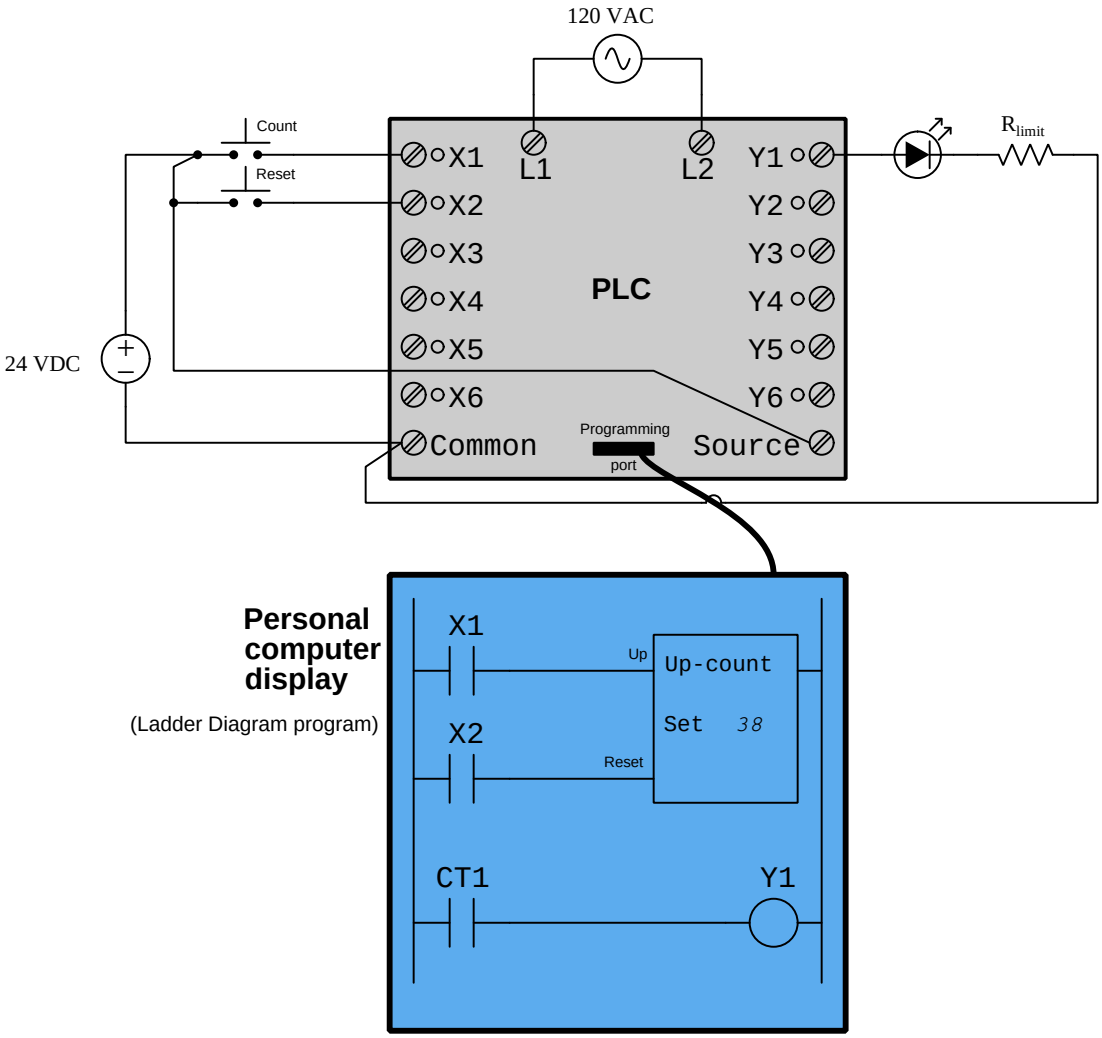
A more sophisticated up-counting program with the capability of resetting the accumulated count value back to zero as well as driving a load (LED) based on the count value meeting or exceeding a pre-set value (arbitrarily set to 38) is shown below:



Pseudocode listing

```
Declare Pin0 as an input
Declare Pin1 as an input
Declare Pin7 as an output
Declare Last_Pin0 as a variable
Declare Count as a variable

LOOP
  IF Pin0 is HIGH and Last_Pin0 is LOW, add one to Count

  IF Pin1 is HIGH, force Count to equal zero

  IF Count is greater than or equal to 38, set Pin7 HIGH
  ELSE, set Pin7 LOW
  ENDIF

  Set Last_Pin0 equal to the value of Pin0
ENDLOOP
```

Counter functions are so commonly useful for the control of industrial processes that all PLCs come with special counter functions built-in to their ladder diagram programming language. Here we see an up-counter function implemented in a PLC:

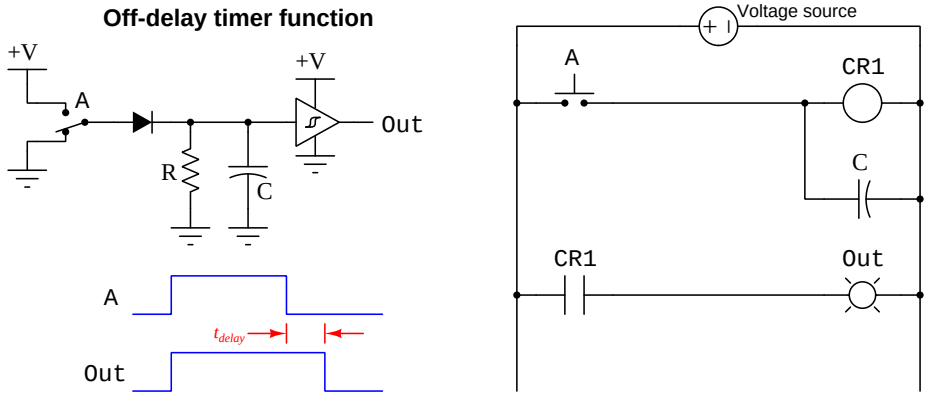


This PLC program energizes the LED if the count value meets or exceeds 38, just like the microcontroller program shown previously.

3.10 Off-delay timer function

Aside from logical functions such as AND, OR, XOR, and others, another class of functions useful for real-world control applications are *timers*. The first type of timer function we will explore is the *off-delay* timer, which means activation of its input results in immediate activation of the output, but de-activation of the input causes the output to wait a specified amount of time before de-activating. In other words, *an off-delay timer delays in turning its output off.*

The following schematic diagrams show how it is possible to implement an off-delay timer function directly with hardware, using both a semiconductor logic gate (left) and electromechanical relay circuit (right) drawn using standard industrial "ladder diagram" convention:



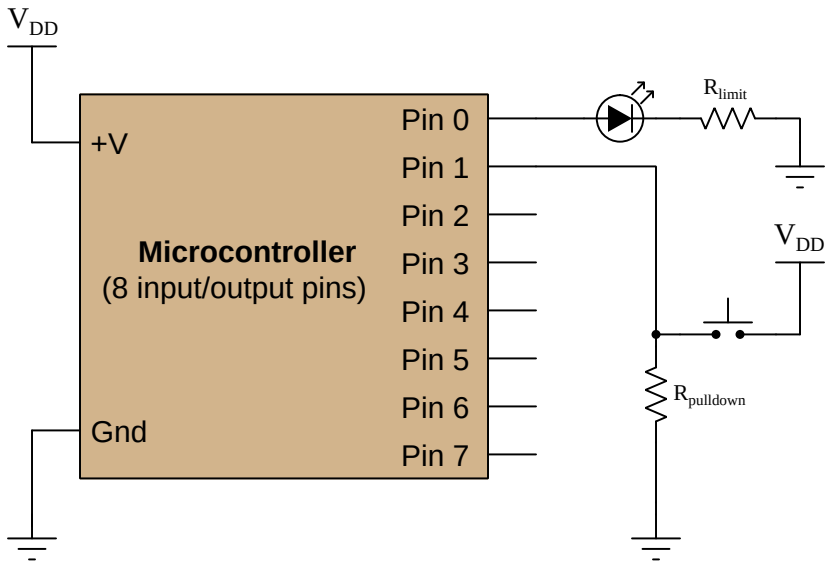
In both of these circuits, activation of the input (A) results in the capacitor immediately receiving a full "charge" of energy. In the semiconductor circuit this charge maintains a high state at the buffer gate's input terminal, forcing its output high as well. In the relay circuit this charge maintains voltage across relay coil CR1, actuating its normally-open contact and forcing it closed to illuminate the lamp.

When input A in the semiconductor circuit goes low, the diode prevents that low state from sinking current and thereby discharging the capacitor. So, the capacitor slowly discharges through the resistor, until the threshold is reached where the Schmitt-trigger[1] buffer gate interprets the decaying voltage as a logical low state and its output finally returns to its original low state. The relay circuit acts similarly, the coil CR1's resistance acting as the dissipating element to the capacitor's internal store of energy once switch A returns to its normal (open) state. Once the coil's voltage falls below the drop-out level its contact will return to its normal (open) state as well, finally de-energizing the lamp.

The amount of off-delay time in either circuit is a function of capacitance (C) and resistance (R), forming an RC "time constant" (τ) describing how rapidly or slowly the capacitor's voltage decays once the initiating stimulus (A) ceases. Additionally, the buffer gate's low-input threshold voltage and the relay's drop-out voltage are important factors, dictating how low the capacitor's voltage must fall in either case before the output returns to its original state.

[1] A *Schmitt trigger* logic gate is specially designed to tolerate input signal voltages that slowly drift between "low" and "high" threshold values, thereby converting an input signal having a lazy rise/fall time into an output signal that crisply transitions between one state and another as a digital pulse signal should.

Next, we see one way to implement an off-delay timing function using a microcontroller:



Pseudocode listing

```
Declare Pin0 as an output
Declare Pin1 as an input

LOOP
  IF Pin1 is HIGH, jump to DELAY routine
  ENDIF
ENDLOOP

DELAY
  Set Pin0 HIGH
  Stop execution for a specified period
  Set Pin0 LOW
ENDDELAY
```

This program uses a *routine* (also known as a *subroutine* or a *function*) outside of the main "loop" where the flow of execution gets redirected and then returns where it left off when that routine completes.

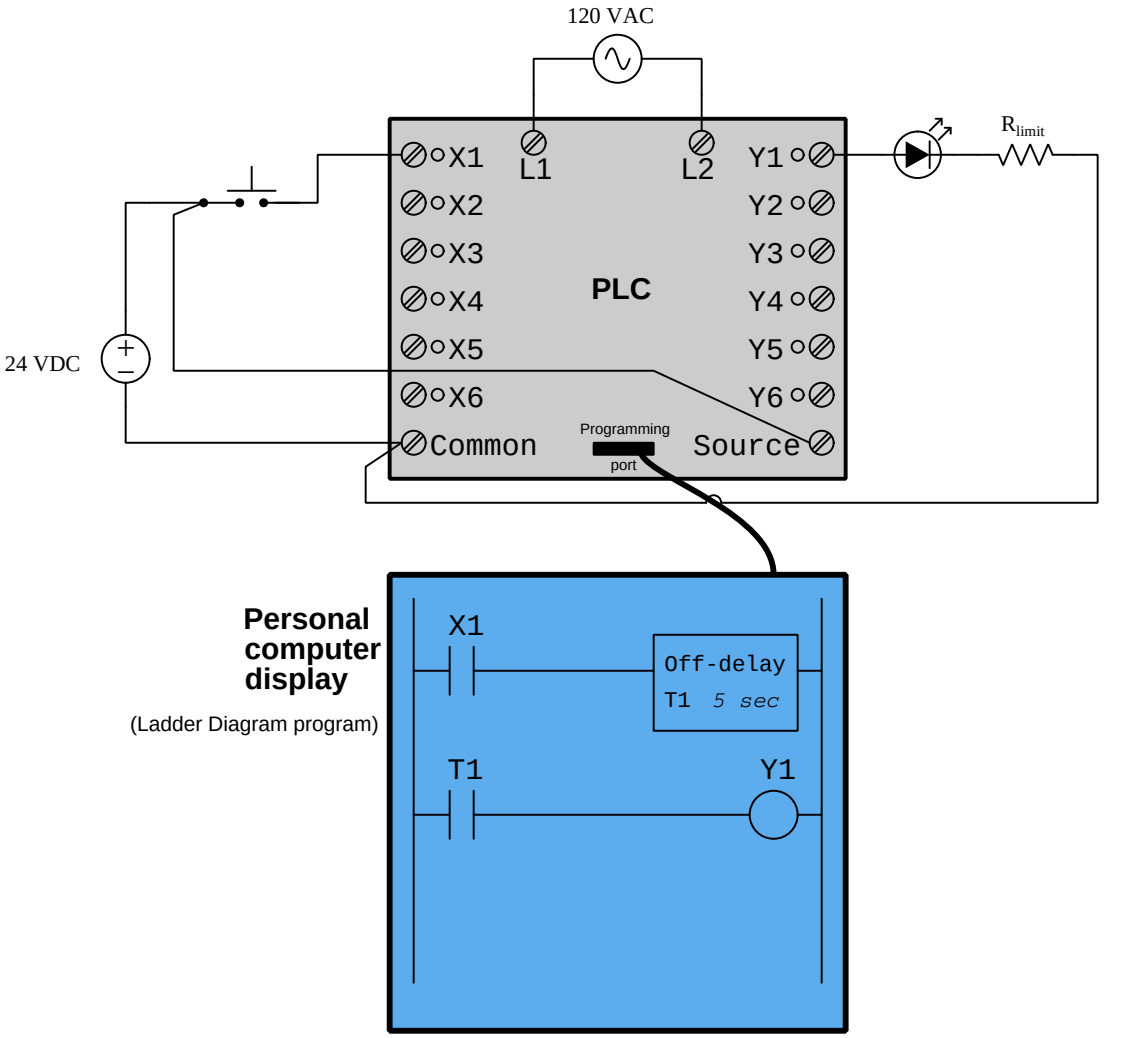
One major disadvantage of the previous timing program is that it entirely halts the microcontroller's execution in implementing the time delay. Whether this is done by having the microcontroller execute a "useless" loop within the `DELAY` routine, or by having it go to a "sleep" state for a period of time, the price we pay for this strategy is having the microcontroller be unavailable to do other useful tasks while it's delaying.

A more sophisticated and capable approach relies on an internal "clock" within the microcontroller, which may be read and compared against stored numerical values. Instead of instructing the microcontroller to "freeze" during the delay period, we instead have it repeatedly check its own clock while (potentially) doing other practical tasks so that the execution of those other tasks is not hindered. Here is one way to code this strategy:

Pseudocode listing

```
Declare Pin0 as an output
Declare Pin1 as an input
Declare TIME as a variable, initially zero
Declare DELAY as a constant equal to our desired time delay

LOOP
  IF Pin1 is HIGH, set TIME equal to current clock value
  ELSE maintain the value stored in TIME
  ENDIF

  IF current clock value is greater than (TIME + DELAY), set Pin0 LOW
  ELSE set Pin0 HIGH
  ENDIF

  (other useful code goes here)

ENDLOOP
```

Timing functions are so commonly useful for the control of industrial processes that all PLCs come with special timer functions built-in to their ladder diagram programming language. Here we see an off-delay timing function implemented in a PLC:

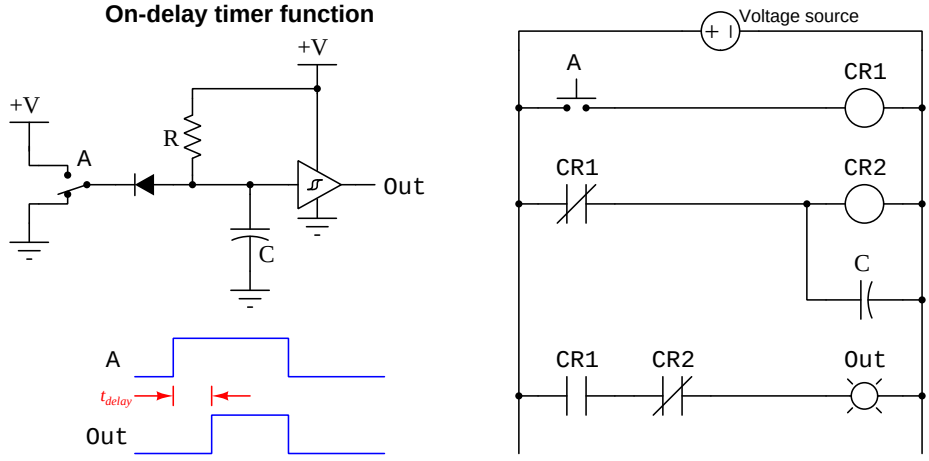


This program introduces a new label, `T`, associated with timer functions. Here, the virtual "contact" `T1` is actuated by the timer function: immediately when `X1` closes, and for an additional five second period of time following `X1` returning to its resting (open) state.

3.11 On-delay timer function

The next type of timer function we will explore is the *on-delay* timer, which means activation of its input results in time-delayed activation of the output, but de-activation of the input causes immediate de-activation of the output. In other words, *an on-delay timer delays in turning its output on.*

The following schematic diagrams show how it is possible to implement an on-delay timer function directly with hardware, using both a semiconductor logic gate (left) and electromechanical relay circuit (right) drawn using standard industrial "ladder diagram" convention:

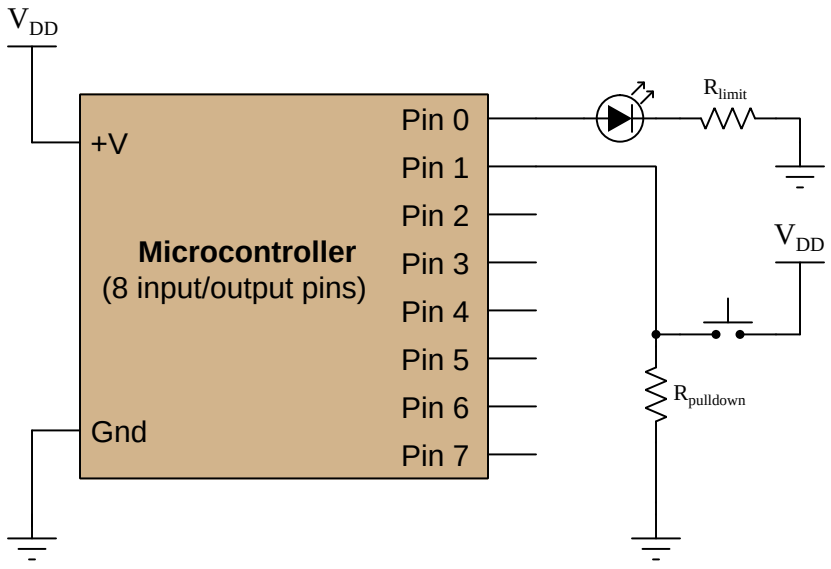


In both of these circuits, activation of the input (A) results in the capacitor's voltage either growing (gate circuit) or decaying (relay circuit) over time. When the capacitor voltage achieves a certain level, the output finally goes high in response to the input.

The amount of on-delay time in either circuit is a function of capacitance (C) and resistance (R), forming an RC "time constant" (τ) describing how rapidly or slowly the capacitor's voltage varies once the initiating stimulus (A) ceases. Additionally, the buffer gate's high-input threshold voltage and the relay's drop-out voltage are important factors, dictating how far the capacitor's voltage must go before the output switches states.

A note on active reading and analysis here: timer circuits such as this (and the previous off-delay type) are challenging to understand because conditions of voltage and current aren't consistent over time. A good problem-solving strategy for analyzing either the logic gate or the relay logic circuits shown here is to copy the diagram of interest and re-draw it several different times, each of those re-drawn diagrams representing the condition of the circuit at a different point in time identifiable on the timing diagram (the blue pulse-waveform illustration). Then, you may annotate each of those diagrams with voltage and current values representing what occurs at those different points in time.

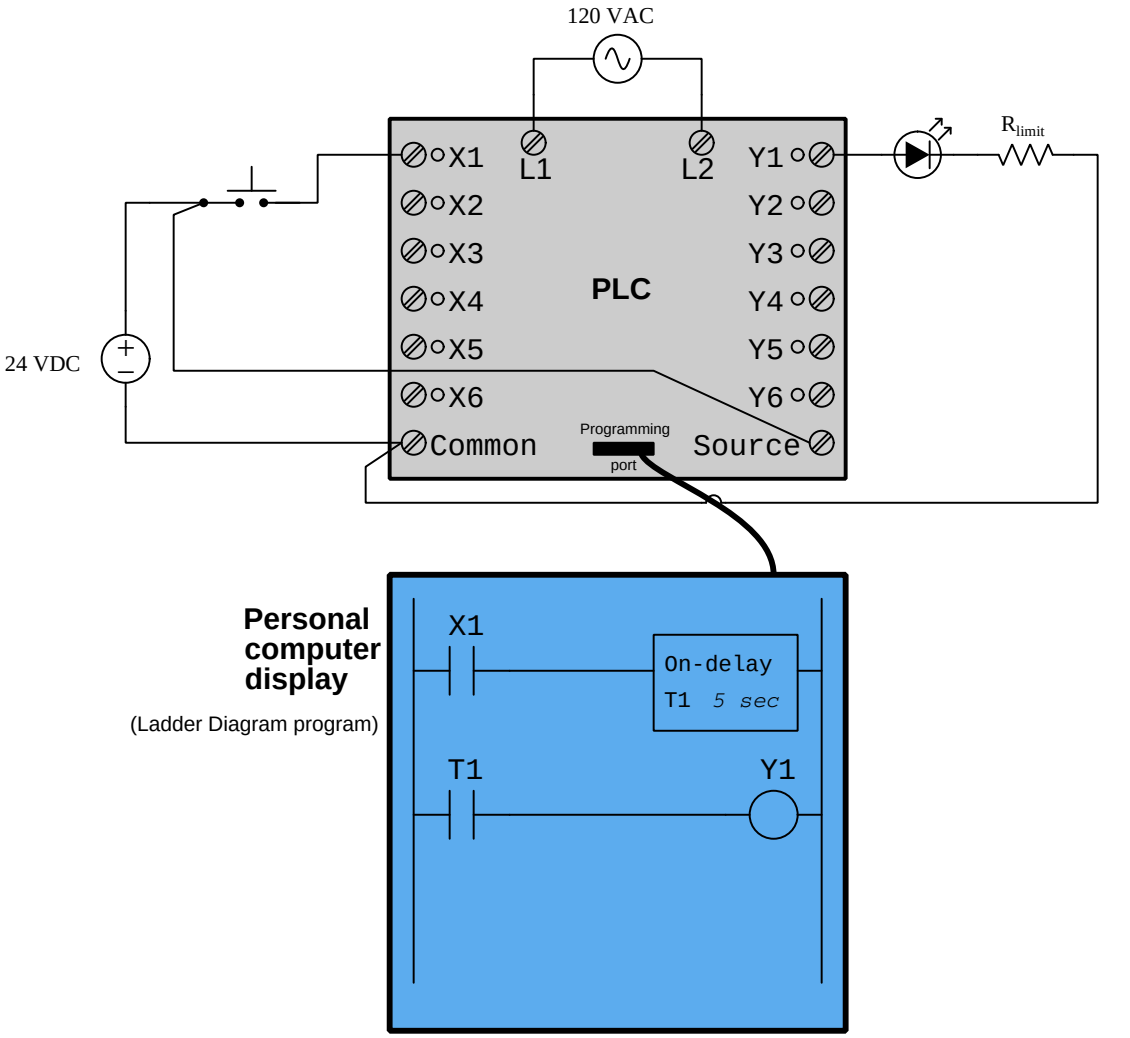
Next, we see one way to implement an on-delay timing function using a microcontroller:



Pseudocode listing

```
Declare Pin0 as an output
Declare Pin1 as an input
Declare TIME as a variable, initially zero
Declare DELAY as a constant equal to our desired time delay

LOOP
  IF Pin1 is LOW, set TIME equal to current clock value
  ELSE maintain the value stored in TIME
  ENDIF

  IF current clock value is greater than (TIME + DELAY), set Pin0 HIGH
  ELSE set Pin0 LOW
  ENDIF

  (other useful code goes here)

ENDLOOP
```

Timing functions are so commonly useful for the control of industrial processes that all PLCs come with special timer functions built-in to their ladder diagram programming language. Here we see an on-delay timing function implemented in a PLC:

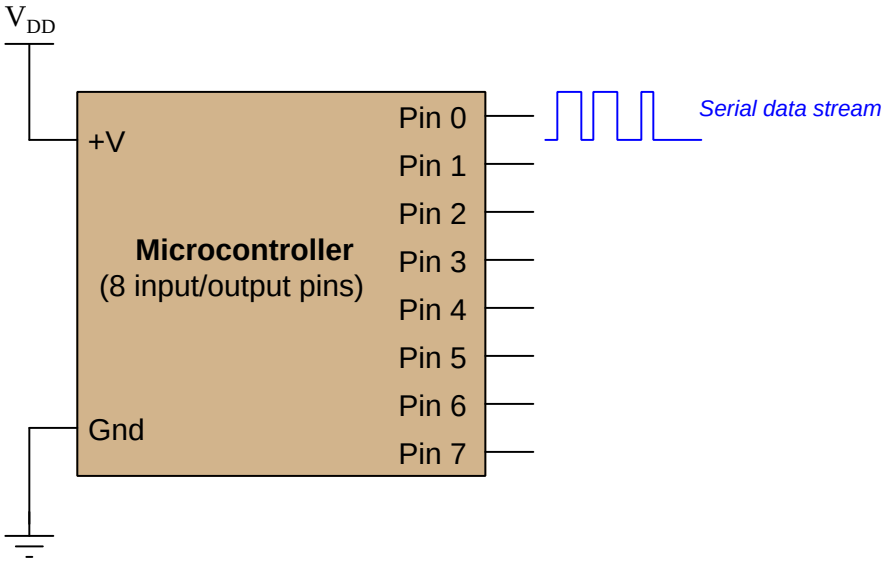


The `T1` virtual "contact" is actuated by the timer function: five seconds after continuous closure of "contact" `X1`, returning to its resting (open) state immediately as the timing function is de-energized by the opening of "contact" `X1`.

3.12 Serial data communication

The ability of programmable digital devices to exchange data with each other is one of the more compelling advantages of digital technology. When digital devices transmit and/or receive binary data as a sequence of individual bits sent one at a time in rapid succession, we call it *serial* data communication, and it is a feature supported in one form or another by most programmable digital devices.

Here we see an illustration showing a microcontroller transmitting (i.e. outputting) a stream of serial data:



The blue-colored pulse waveform depicted here represents a sequence of "high" and "low" logic states represented as a voltage changing from full V_{DD} voltage ("high") to ground potential ("low") and back again. Viewed on an oscilloscope display, a serial data stream is a pulse waveform.

While the basic concept of serial data communication is easy enough to grasp – simply use digital "high" and "low" states in sequence over time to represent a set of binary bits – there is much detail that needs to be agreed upon before we may actually build functional communication systems that do this. Here are just some of those details:

- What voltage levels will represent "1" and "0" binary bit states?
- How rapidly will each bit be communicated (i.e. how many bits per second)?
- How will transmitting and receiving devices be synchronized together so that the bits are transmitted and received in their proper times?
- In what order will the bits of a binary word be communicated, from LSB to MSB or vice-versa?
- If multiple binary words must be communicated in one uninterrupted data stream, how do we differentiate one word from another?

- How may we check a received binary word for errors that may occur during transmission?

Though some of these questions may seem to have obvious answers, such as using positive DC supply voltage to represent 1 and zero voltage to represent 0 as we typically do with semiconductor logic gates, there are in fact multiple answers for each. Any set of definitive answers to these and similar questions constitutes a *standard* for serial data communication, and as one might guess there are many, many such standards in existence[2]. In this section we will focus on just one serial communication standard called *UART*.

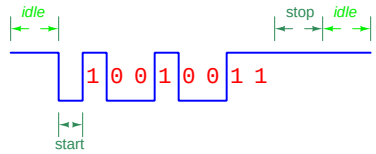
UART is an acronym standing for *Universal Asynchronous Receiver-Transmitter*, used both to describe a particular standard of serial data communication as well as to label any integrated circuit used to transmit and receive pulse signals complying with that standard. Features of the UART serial standard include:

- The "idle" voltage state while not communicating will have the same voltage level as a "1" logical state
- Each new communication sequence is prefaced by a "start" signal having the same voltage level as a "0" logical state
- Binary bits communicated in order of LSB (first) to MSB (last) at a constant rate, the total number of bits being pre-configured in the system
- An optional "parity" bit follows the MSB of the binary word, its value being whatever is necessary to bring the total number of "1" bits to either an odd or an even number as pre-configured for the system
- Each communication sequence is concluded by a "stop" signal having the same voltage level as a "1" logical state, or "idle" state for a minimum of one bit's time

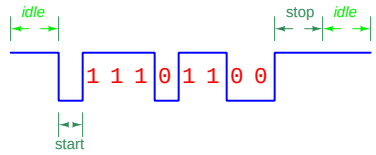
[2] As the joke says, the wonderful thing about standards is that there are so many to choose from!

UART communication is perhaps easiest to understand by way of practical example. In the following illustrations we will show three examples of UART serial data signals representing eight-bit binary numbers, assuming the use of positive DC source voltage for "idle" and logical "1" states and zero voltage for logical "0" states, no parity bit, and two stop bits:

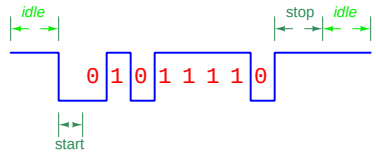
Binary number = 11001001



Binary number = 00110111

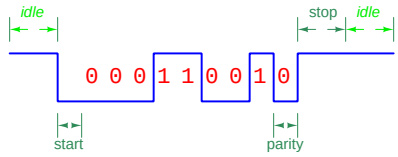


Binary number = 01111010

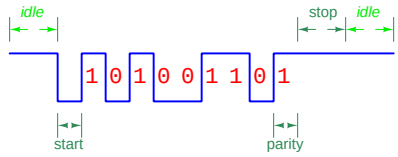


Next we will examine three more examples of UART signals using the same parameters as before (positive logic levels, 8 data bits, and 2 stop bits), but this time including a *parity bit* set for *odd parity*:

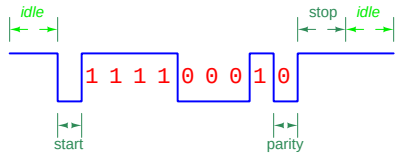
Binary number = 10011000



Binary number = 01100101



Binary number = 10001111



Note how each one of the bits' time duration is identical, the data bits each occupying the

exact same amount of time on the horizontal axis as each start bit, each stop bit, and each parity bit. The duration of the "idle" time is arbitrary, depending only on how often binary numbers get communicated.

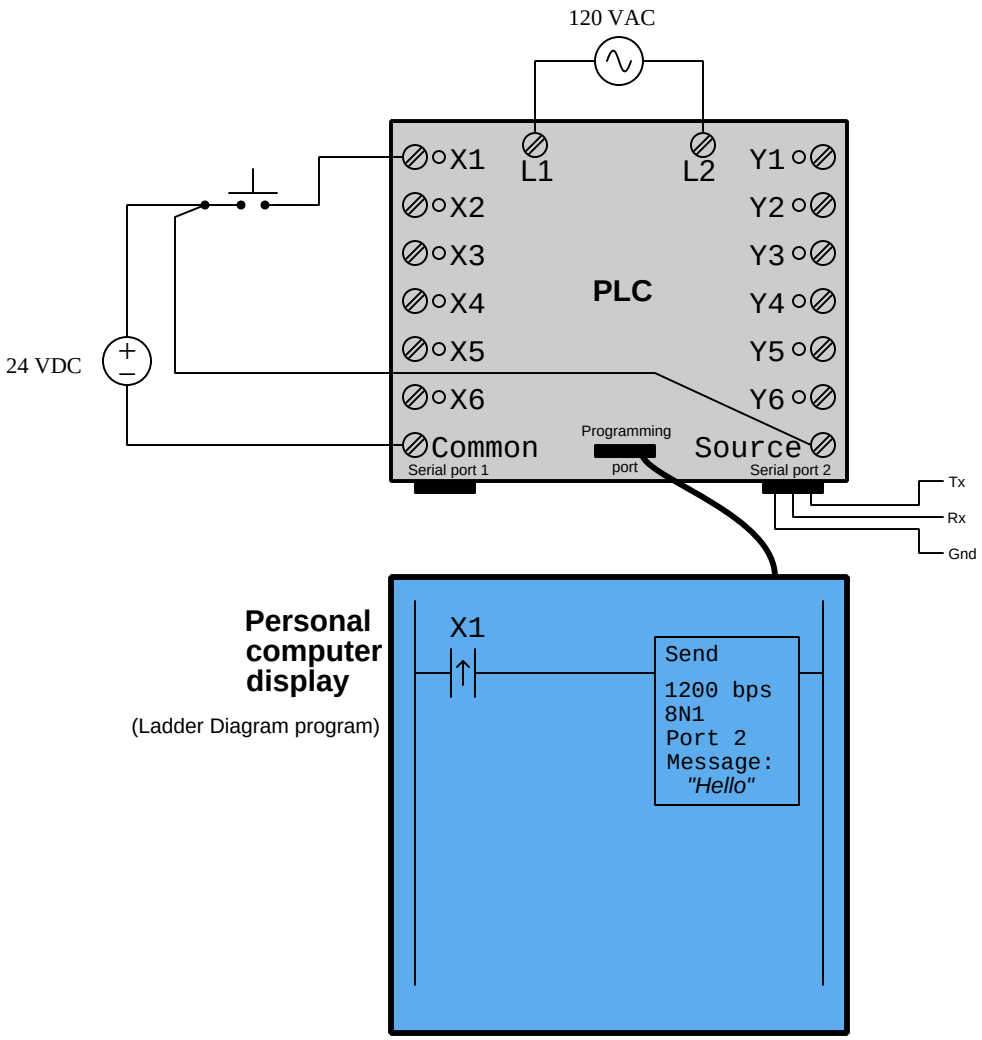
Parity bits exist for the purpose of *error detection* in serial data communication. Errors usually originate in communication systems due to excessive electrical *noise*, and so it is possible for noise to grow strong enough to corrupt one of the bits so as to be interpreted at the receiving end as the wrong logical state; i.e. a bit transmitted as a 0 being received as a 1, or vice-versa. If both transmitting and receiving devices are pre-configured to generate and interpret (respectively) a parity bit whose value is based on the number of "1" bits in the data payload, any single-bit corruption will be detectable as such when the receiver does its own parity calculation and determines a mis-match.

In the last set of UART signal examples shown, parity was configured for "odd". This meant that the transmitting device counted up the number of "1" bits in the eight-bit binary number and then made the parity bit either a "1" or a "0" as necessary to bring the total "1" bit count to an odd value. For example, the binary number `10011000` in the first parity example already had an odd number of "1" bits and so the transmitting device set the parity bit to zero because it didn't need any more "1" bits to make an odd count. However, the binary number `01100101` used in the second parity example had an even number of "1" bits and so the transmitting device did need to set the parity but to one in order to being the total "1" bit count to an odd number. The receiving device simply has to count all the "1" bits it receives in total to see if it finds an odd number. If it happens to count an even number of "1" bits in total, it would knows something has gone wrong.

Microcontroller programming for serial data transmission and reception varies substantially between different microcontroller models, so much so that no attempt will be made here to represent a generic "pseudocode" example. High-level microcontroller programming languages such as Sketch provide extremely easy-to-use commands, where a single instruction is used to configure the UART parameters (e.g. bit rate, number of data bits, etc.) and another single instruction triggers the data transfer. Microcontrollers programmed in lower-level programming languages such as C which lack dedicated serial-communication instructions often require a complex[3] set of instructions necessary to move the necessary configuration data into specific microcontroller registers and then move the binary data into or out of other registers as communication occurs.

[3] For example, the Texas Instruments MSP430 microcontroller requires your serial-transmit program load data into a designated place in memory called the *transmit buffer*, then monitor status variables to determine when the UART hardware inside the MSP430 has completed that task before sending any additional data to the buffer.

Serial data communication is an important feature of PLCs as well, enough so that special instructions exist to send and receive serial data, and dedicated connection points are provided on the PLC hardware for the UART signals. Here we see a simple serial transmission function implemented in a PLC:



A "transition" or "edge" style of contact instruction is used for `X1`, so that the `Send` function is activated only *once* every time the pushbutton switch connected in input terminal `X1` is pressed. If the `Send` function were continually activated, it is likely that the instruction would be re-executed before it had time to send the first message, which would result in a garbled message.

Unlike microcontrollers which typically input and output serial data signals at standard logic-level voltages (e.g. +5 Volts and 0 Volts, or +3.3 Volts and 0 Volts), PLCs serial ports are typically

equipped with *driver* amplifiers designed to input and output serial data pulses at much higher voltage levels. For example, if a PLC's serial port conforms to the EIA/TIA-232 communication standard[4], it must output and input peak-to-peak signal strengths of *at least* $\pm$ 5 Volts.

[4]This standard, also referred to as *RS-232*, happens to use negative logic for its UART signals, with a negative voltage representing "1" logical as well as "idle" states, and a positive voltage representing "0" logical states.

Chapter 4

Historical References

This chapter is where you will find references to historical texts and technologies related to the module's topic.

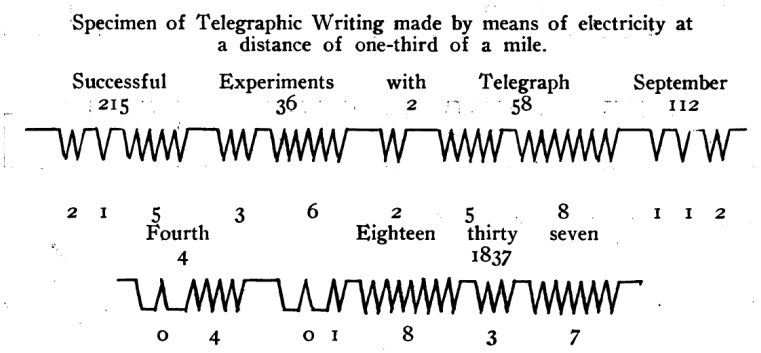
Readers may wonder why historical references might be included in any modern lesson on a subject. Why dwell on old ideas and obsolete technologies? One answer to this question is that the initial discoveries and early applications of scientific principles typically present those principles in forms that are unusually easy to grasp. Anyone who first discovers a new principle must necessarily do so from a perspective of ignorance (i.e. if you truly *discover* something yourself, it means you must have come to that discovery with no prior knowledge of it and no hints from others knowledgeable in it), and in so doing the discoverer lacks any hindsight or advantage that might have otherwise come from a more advanced perspective. Thus, discoverers are forced to think and express themselves in less-advanced terms, and this often makes their explanations more readily accessible to others who, like the discoverer, comes to this idea with no prior knowledge. Furthermore, early discoverers often faced the daunting challenge of explaining their new and complex ideas to a naturally skeptical scientific community, and this pressure incentivized clear and compelling communication. As James Clerk Maxwell eloquently stated in the Preface to his book *A Treatise on Electricity and Magnetism* written in 1873,

> It is of great advantage to the student of any subject to read the original memoirs on that subject, for science is always most completely assimilated when it is in its nascent state . . . `[page xi]`

Furthermore, grasping the historical context of technological discoveries is important for understanding how science intersects with culture and civilization, which is ever important because new discoveries and new applications of existing discoveries will always continue to impact our lives. One will often find themselves impressed by the ingenuity of previous generations, and by the high degree of refinement to which now-obsolete technologies were once raised. There is much to learn and much inspiration to be drawn from the technological past, and to the inquisitive mind these historical references are treasures waiting to be (re)-discovered.

4.1 The original telegraph code

Samuel Morse, inventor of the telegraph, actually did not use what is known today as *Morse Code* to transmit his first messages. Instead, his original method of encoding was based on decimal numbers, with specific words represented by unique numbers. Page 29 of the book *Early History of the Electro-Magnetic Telegraph* by Alfred Vail documents a typical message communicated using Morse's system:



The *words* in the diagram were the intelligence transmitted.

The *numbers* (in this instance arbitrary) are the numbers of the words in a telegraphic dictionary.

The *points* are the markings of the register, each point being marked every time the Electric fluid passes.

The register marks but one kind of mark to wit, (V) this can be varied two ways. By intervals, thus, (V VV VVV), signifying one, two, three, etc., and by reversing, thus, (Λ). Examples of both these varieties are seen in the diagram.

The single numbers are separated by *short,* and the whole numbers by *long intervals.*

The jagged lines form an oscillograph of the telegraph circuit's digital state over time, and as you can see it used groups of successive pulses to count digits of decimal numbers. For example, the number 215 was encoded as two pulses followed by a brief rest, then one pulse followed by another rest, then a sequence of five pulses. A longer rest separated individual numbers from each other, as opposed to the shorter rests which merely separated digits of the same number from each other.

A "telegraphic dictionary" served to cross-reference common words with number values. As you can see from the example shown, the number 215 represented the word *Successful*, while 36 represented *Experiments*, 2 represented *With*, and so on. One can only imagine how many numbers would have been required to represent even a rudimentary cross-section of the English vocabulary!

Chapter 5

Derivations and Technical References

This chapter is where you will find mathematical derivations too detailed to include in the tutorial, and/or tables and other technical reference material.

5.1 ASCII character codes

ASCII characters consist of seven-bit digital words. The following table shows all 128 possible combinations of these seven bits, from 0000000 (ASCII "NUL" character) to 1111111 (ASCII "DEL" character). For ease of organization, this table's columns represent the most-significant three bits of the seven-bit word, while the table's rows represent the least-significant four bits. For example, the capital letter "C" would be encoded as 1000011 in the ASCII standard.

↓ LSB / MSB →	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	‘	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	”	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	−	=	M	]	m	}
1110	SO	RS	.	>	N	ˆ	n	˜
1111	SI	US	/	?	O	_	o	DEL

It is worth noting that the ASCII codes for the Arabic numerals 0 through 9 are simply the four-bit binary representation of those numbers preceded by 011. For example, the number six (0110) is represented in ASCII as 0110110; the number three (0011) in ASCII as 0110011; etc. This is useful to know, for example, if you need to program a computer to convert single decimal digits to their corresponding ASCII codes: just take each four-bit numerical value and add forty-eight (0x30 in hexadecimal) to it.

Chapter 6

Programming References

A powerful tool for mathematical modeling is text-based *computer programming*. This is where you type coded commands in text form which the computer is able to interpret. Many different text-based languages exist for this purpose, but we will focus here on just two of them, *C++* and *Python*.

6.1 Programming in C++

One of the more popular text-based computer programming languages is called *C++*. This is a *compiled* language, which means you must create a plain-text file containing C++ code using a program called a *text editor*, then execute a software application called a *compiler* to translate your "source code" into instructions directly understandable to the computer. Here is an example of "source code" for a very simple C++ program intended to perform some basic arithmetic operations and print the results to the computer's console:

```
#include <iostream>
using namespace std;

int main (void)
{
  float x, y;

  x = 200;
  y = -560.5;

  cout << "This simple program performs basic arithmetic on" << endl;
  cout << "the two numbers " << x << " and " << y << " and then" << endl;
  cout << "displays the results on the computer's console." << endl;

  cout << endl;

  cout << "Sum = " << x + y << endl;
  cout << "Difference = " << x - y << endl;
  cout << "Product = " << x * y << endl;
  cout << "Quotient of " << x / y << endl;

  return 0;
}
```

Computer languages such as C++ are designed to make sense when read by human programmers. The general order of execution is left-to-right, top-to-bottom just the same as reading any text document written in English. Blank lines, indentation, and other "whitespace" is largely irrelevant in C++ code, and is included only to make the code more pleasing[1] to view.

[1] Although not included in this example, *comments* preceded by double-forward slash characters (`//`) may be added to source code as well to provide explanations of what the code is supposed to do, for the benefit of anyone reading it. The compiler application will ignore all comments.

Let's examine the C++ source code to explain what it means:

- `#include <iostream>` and `using namespace std;` are set-up instructions to the compiler giving it some context in which to interpret your code. The code specific to your task is located between the brace symbols ({ and }, often referred to as "curly-braces").

- `int main (void)` labels the "Main" function for the computer: the instructions within this function (lying between the { and } symbols) it will be commanded to execute. Every complete C++ program contains a `main` function at minimum, and often additional functions as well, but the `main` function is where execution always begins. The `int` declares this function will return an *integer* number value when complete, which helps to explain the purpose of the `return 0;` statement at the end of the `main` function: providing a numerical value of zero at the program's completion as promised by `int`. This returned value is rather incidental to our purpose here, but it is fairly standard practice in C++ programming.

- Grouping symbols such as (parentheses) and {braces} abound in C, C++, and other languages (e.g. Java). Parentheses typically group data to be processed by a function, called *arguments* to that function. Braces surround lines of executable code belonging to a particular function.

- The `float` declaration reserves places in the computer's memory for two *floating-point* variables, in this case the variables' names being `x` and `y`. In most text-based programming languages, variables may be named by single letters or by combinations of letters (e.g. `xyz` would be a single variable).

- The next two lines assign numerical values to the two variables. Note how each line terminates with a semicolon character (`;`) and how this pattern holds true for most of the lines in this program. In C++ semicolons are analogous to periods at the ends of English sentences. This demarcation of each line's end is necessary because C++ ignores whitespace on the page and doesn't "know" otherwise where one line ends and another begins.

- All the other instructions take the form of a `cout` command which prints characters to the "standard output" stream of the computer, which in this case will be text displayed on the console. The double-less-than symbols (`<<`) show data being sent *toward* the `cout` command. Note how verbatim text is enclosed in quotation marks, while variables such as `x` or mathematical expressions such as `x - y` are not enclosed in quotations because we want the computer to display the numerical values represented, not the literal text.

- Standard arithmetic operations (add, subtract, multiply, divide) are represented as `+`, `-`, `*`, and `/`, respectively.

- The `endl` found at the end of every `cout` statement marks the end of a line of text printed to the computer's console display. If not for these `endl` inclusions, the displayed text would resemble a run-on sentence rather than a paragraph. Note the `cout << endl;` line, which does nothing but create a blank line on the screen, for no reason other than esthetics.

After saving this *source code* text to a file with its own name (e.g. `myprogram.cpp`), you would then *compile* the source code into an *executable* file which the computer may then run. If you are using a console-based compiler such as *GCC* (very popular within variants of the Unix operating system[2], such as Linux and Apple's OS X), you would type the following command and press the Enter key:

```
g++ -o myprogram.exe myprogram.cpp
```

This command instructs the `GCC` compiler to take your source code (`myprogram.cpp`) and create with it an executable file named `myprogram.exe`. Simply typing `./myprogram.exe` at the command-line will then execute your program:

```
./myprogram.exe
```

If you are using a graphic-based C++ development system such as Microsoft Visual Studio[3], you may simply create a new console application "project" using this software, then paste or type your code into the example template appearing in the editor window, and finally run your application to test its output.

As this program runs, it displays the following text to the console:

```
This simple program performs basic arithmetic on
the two numbers 200 and -560.5 and then
displays the results on the computer's console.

Sum = -360.5
Difference = 760.5
Product = -112100
Quotient of -0.356824
```

As crude as this example program is, it serves the purpose of showing how easy it is to write and execute simple programs in a computer using the C++ language. As you encounter C++ example programs (shown as source code) in any of these modules, feel free to directly copy-and-paste the source code text into a text editor's screen, then follow the rest of the instructions given here (i.e. save to a file, compile, and finally run your program). You will find that it is generally easier to

[2] A very functional option for users of Microsoft Windows is called *Cygwin*, which provides a Unix-like console environment complete with all the customary utility applications such as GCC!

[3] Using Microsoft Visual Studio community version 2017 at the time of this writing to test this example, here are the steps I needed to follow in order to successfully compile and run a simple program such as this: (1) Start up Visual Studio and select the option to create a New Project; (2) Select the Windows Console Application template, as this will perform necessary set-up steps to generate a console-based program which will save you time and effort as well as avoid simple errors of omission; (3) When the editing screen appears, type or paste the C++ code within the `main()` function provided in the template, deleting the "Hello World" `cout` line that came with the template; (4) Type or paste any preprocessor directives (e.g. `#include` statements, `namespace` statements) necessary for your code that did not come with the template; (5) Lastly, under the Debug drop-down menu choose either Start Debugging (F5 hot-key) or Start Without Debugging (Ctrl-F5 hotkeys) to compile ("Build") and run your new program. Upon execution a console window will appear showing the output of your program.

learn computer programming by closely examining others' example programs and modifying them than it is to write your own programs starting from a blank screen.

6.2 Programming in Python

Another text-based computer programming language called *Python* allows you to type instructions at a terminal prompt and receive immediate results without having to compile that code. This is because Python is an *interpreted* language: a software application called an *interpreter* reads your source code, translates it into computer-understandable instructions, and then executes those instructions in one step.

The following shows what happens on my personal computer when I start up the Python interpreter on my personal computer, by typing `python3`[4] and pressing the Enter key:

```
Python 3.7.2 (default, Feb 19 2019, 18:15:18)
[GCC 4.1.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The `>>>` symbols represent the prompt within the Python interpreter "shell", signifying readiness to accept Python commands entered by the user.

Shown here is an example of the same arithmetic operations performed on the same quantities, using a Python interpreter. All lines shown preceded by the `>>>` prompt are entries typed by the human programmer, and all lines shown without the `>>>` prompt are responses from the Python interpreter software:

```
>>> x = 200
>>> y = -560.5
>>> x + y
-360.5
>>> x - y
760.5
>>> x * y
-112100.0
>>> x / y
-0.35682426404995538
>>> quit()
```

[4] Using version 3 of Python, which is the latest at the time of this writing.

More advanced mathematical functions are accessible in Python by first entering the line `from math import *` which "imports" these functions from Python's math *library* (with functions identical to those available for the C programming language, and included on any computer with Python installed). Some examples show some of these functions in use, demonstrating how the Python interpreter may be used as a scientific calculator:

```
>>> from math import *
>>> sin(30.0)
-0.98803162409286183
>>> sin(radians(30.0))
0.49999999999999994
>>> pow(2.0, 5.0)
32.0
>>> log10(10000.0)
4.0
>>> e
2.7182818284590451
>>> pi
3.1415926535897931
>>> log(pow(e,6.0))
6.0
>>> asin(0.7071068)
0.78539819000368838
>>> degrees(asin(0.7071068))
45.000001524425265
>>> quit()
```

Note how trigonometric functions assume angles expressed in *radians* rather than *degrees*, and how Python provides convenient functions for translating between the two. Logarithms assume a base of e unless otherwise stated (e.g. the `log10` function for common logarithms).

The interpreted (versus compiled) nature of Python, as well as its relatively simple syntax, makes it a good choice as a person's first programming language. For complex applications, interpreted languages such as Python execute slower than compiled languages such as C++, but for the very simple examples used in these learning modules speed is not a concern.

Another Python math library is `cmath`, giving Python the ability to perform arithmetic on complex numbers. This is very useful for AC circuit analysis using *phasors*[5] as shown in the following example. Here we see Python's interpreter used as a scientific calculator to show series and parallel impedances of a resistor, capacitor, and inductor in a 60 Hz AC circuit:

```
>>> from math import *
>>> from cmath import *
>>> r = complex(400,0)
>>> f = 60.0
>>> xc = 1/(2 * pi * f * 4.7e-6)
>>> zc = complex(0,-xc)
>>> xl = 2 * pi * f * 1.0
>>> zl = complex(0,xl)
>>> r + zc + zl
(400-187.38811239154882j)
>>> 1/(1/r + 1/zc + 1/zl)
(355.837695813625+125.35793777619385j)
>>> polar(r + zc + zl)
(441.717448903332, -0.4381072059213295)
>>> abs(r + zc + zl)
441.717448903332
>>> phase(r + zc + zl)
-0.4381072059213295
>>> degrees(phase(r + zc + zl))
-25.10169387356105
```

When entering a value in rectangular form, we use the `complex()` function where the arguments are the real and imaginary quantities, respectively. If we had opted to enter the impedance values in polar form, we would have used the `rect()` function where the first argument is the magnitude and the second argument is the angle in radians. For example, we could have set the capacitor's impedance (`zc`) as $X_C \angle -90^o$ with the command `zc = rect(xc,radians(-90))` rather than with the command `zc = complex(0,-xc)` and it would have worked the same.

Note how Python defaults to rectangular form for complex quantities. Here we defined a 400 Ohm resistance as a complex value in rectangular form (400 +j0 Ω), then computed capacitive and inductive reactances at 60 Hz and defined each of those as complex (phasor) values ($0 - jX_c$ Ω and $0 + jX_l$ Ω, respectively). After that we computed total impedance in series, then total impedance in parallel. Polar-form representation was then shown for the series impedance (441.717 Ω $\angle -25.102^o$). Note the use of different functions to show the polar-form series impedance value: `polar()` takes the complex quantity and returns its polar magnitude and phase angle in *radians*; `abs()` returns just the polar magnitude; `phase()` returns just the polar angle, once again in radians. To find the polar phase angle in degrees, we nest the `degrees()` and `phase()` functions together.

The utility of Python's interpreter environment as a scientific calculator should be clear from these examples. Not only does it offer a powerful array of mathematical functions, but also unlimited

[5] A "phasor" is a voltage, current, or impedance represented as a complex number, either in rectangular or polar form.

assignment of variables as well as a convenient text record[6] of all calculations performed which may be easily copied and pasted into a text document for archival.

It is also possible to save a set of Python commands to a text file using a text editor application, and then instruct the Python interpreter to execute it at once rather than having to type it line-by-line in the interpreter's shell. For example, consider the following Python program, saved under the filename `myprogram.py`:

```
x = 200
y = -560.5

print("Sum")
print(x + y)

print("Difference")
print(x - y)

print("Product")
print(x * y)

print("Quotient")
print(x / y)
```

As with C++, the interpreter will read this source code from left-to-right, top-to-bottom, just the same as you or I would read a document written in English. Interestingly, whitespace *is* significant in the Python language (unlike C++), but this simple example program makes no use of that.

To execute this Python program, I would need to type `python myprogram.py` and then press the Enter key at my computer console's prompt, at which point it would display the following result:

```
Sum
-360.5
Difference
760.5
Product
-112100.0
Quotient
-0.35682426405
```

As you can see, syntax within the Python programming language is simpler than C++, which is one reason why it is often a preferred language for beginning programmers.

[6] Like many command-line computing environments, Python's interpreter supports "up-arrow" recall of previous entries. This allows quick recall of previously typed commands for editing and re-evaluation.

If you are interested in learning more about computer programming in *any* language, you will find a wide variety of books and free tutorials available on those subjects. Otherwise, feel free to learn by the examples presented in these modules.

Chapter 7

Questions

This learning module, along with all others in the ModEL collection, is designed to be used in an inverted instructional environment where students independently read[1] the tutorials and attempt to answer questions on their own *prior* to the instructor's interaction with them. In place of lecture[2], the instructor engages with students in Socratic-style dialogue, probing and challenging their understanding of the subject matter through inquiry.

Answers are not provided for questions within this chapter, and this is by design. Solved problems may be found in the Tutorial and Derivation chapters, instead. The goal here is *independence*, and this requires students to be challenged in ways where others cannot think for them. Remember that you always have the tools of *experimentation* and *computer simulation* (e.g. SPICE) to explore concepts!

The following lists contain ideas for Socratic-style questions and challenges. Upon inspection, one will notice a strong theme of *metacognition* within these statements: they are designed to foster a regular habit of examining one's own thoughts as a means toward clearer thinking. As such these sample questions are useful both for instructor-led discussions as well as for self-study.

[1] Technical reading is an essential academic skill for any technical practitioner to possess for the simple reason that the most comprehensive, accurate, and useful information to be found for developing technical competence is in textual form. Technical careers in general are characterized by the need for continuous learning to remain current with standards and technology, and therefore any technical practitioner who cannot read well is handicapped in their professional development. An excellent resource for educators on improving students' reading prowess through intentional effort and strategy is the book textitReading For Understanding – How Reading Apprenticeship Improves Disciplinary Learning in Secondary and College Classrooms by Ruth Schoenbach, Cynthia Greenleaf, and Lynn Murphy.

[2] Lecture is popular as a teaching method because it is easy to implement: any reasonably articulate subject matter expert can talk to students, even with little preparation. However, it is also quite problematic. A good lecture always makes complicated concepts seem easier than they are, which is bad for students because it instills a false sense of confidence in their own understanding; reading and re-articulation requires more cognitive effort and serves to verify comprehension. A culture of teaching-by-lecture fosters a debilitating dependence upon direct personal instruction, whereas the challenges of modern life demand independent and critical thought made possible only by gathering information and perspectives from afar. Information presented in a lecture is ephemeral, easily lost to failures of memory and dictation; text is forever, and may be referenced at any time.

General challenges following tutorial reading

- Summarize as much of the text as you can in one paragraph of your own words. A helpful strategy is to explain ideas as you would for an intelligent child: as simple as you can without compromising too much accuracy.
- Simplify a particular section of the text, for example a paragraph or even a single sentence, so as to capture the same fundamental idea in fewer words.
- Where did the text make the most sense to you? What was it about the text's presentation that made it clear?
- Identify where it might be easy for someone to misunderstand the text, and explain why you think it could be confusing.
- Identify any new concept(s) presented in the text, and explain in your own words.
- Identify any familiar concept(s) such as physical laws or principles applied or referenced in the text.
- Devise a proof of concept experiment demonstrating an important principle, physical law, or technical innovation represented in the text.
- Devise an experiment to disprove a plausible misconception.
- Did the text reveal any misconceptions you might have harbored? If so, describe the misconception(s) and the reason(s) why you now know them to be incorrect.
- Describe any useful problem-solving strategies applied in the text.
- Devise a question of your own to challenge a reader's comprehension of the text.

General follow-up challenges for assigned problems

- Identify where any fundamental laws or principles apply to the solution of this problem, especially before applying any mathematical techniques.
- Devise a thought experiment to explore the characteristics of the problem scenario, applying known laws and principles to mentally model its behavior.
- Describe in detail your own strategy for solving this problem. How did you identify and organized the given information? Did you sketch any diagrams to help frame the problem?
- Is there more than one way to solve this problem? Which method seems best to you?
- Show the work you did in solving this problem, even if the solution is incomplete or incorrect.
- What would you say was the most challenging part of this problem, and why was it so?
- Was any important information missing from the problem which you had to research or recall?
- Was there any extraneous information presented within this problem? If so, what was it and why did it not matter?
- Examine someone else's solution to identify where they applied fundamental laws or principles.
- Simplify the problem from its given form and show how to solve this simpler version of it. Examples include eliminating certain variables or conditions, altering values to simpler (usually whole) numbers, applying a limiting case (i.e. altering a variable to some extreme or ultimate value).
- For quantitative problems, identify the real-world meaning of all intermediate calculations: their units of measurement, where they fit into the scenario at hand. Annotate any diagrams or illustrations with these calculated values.
- For quantitative problems, try approaching it qualitatively instead, thinking in terms of "increase" and "decrease" rather than definite values.
- For qualitative problems, try approaching it quantitatively instead, proposing simple numerical values for the variables.
- Were there any assumptions you made while solving this problem? Would your solution change if one of those assumptions were altered?
- Identify where it would be easy for someone to go astray in attempting to solve this problem.
- Formulate your own problem based on what you learned solving this one.

General follow-up challenges for experiments or projects

- In what way(s) was this experiment or project easy to complete?
- Identify some of the challenges you faced in completing this experiment or project.

- Show how thorough documentation assisted in the completion of this experiment or project.
- Which fundamental laws or principles are key to this system's function?
- Identify any way(s) in which one might obtain false or otherwise misleading measurements from test equipment in this system.
- What will happen if (component X) fails (open/shorted/etc.)?
- What would have to occur to make this system unsafe?

7.1 Conceptual reasoning

These questions are designed to stimulate your analytic and synthetic thinking[3]. In a Socratic discussion with your instructor, the goal is for these questions to prompt an extended dialogue where assumptions are revealed, conclusions are tested, and understanding is sharpened. Your instructor may also pose additional questions based on those assigned, in order to further probe and refine your conceptual understanding.

Questions that follow are presented to challenge and probe your understanding of various concepts presented in the tutorial. These questions are intended to serve as a guide for the Socratic dialogue between yourself and the instructor. Your instructor's task is to ensure you have a sound grasp of these concepts, and the questions contained in this document are merely a means to this end. Your instructor may, at his or her discretion, alter or substitute questions for the benefit of tailoring the discussion to each student's needs. The only absolute requirement is that each student is challenged and assessed at a level equal to or greater than that represented by the documented questions.

It is far more important that you convey your *reasoning* than it is to simply convey a correct answer. For this reason, you should refrain from researching other information sources to answer questions. What matters here is that *you* are doing the thinking. If the answer is incorrect, your instructor will work with you to correct it through proper reasoning. A correct answer without an adequate explanation of how you derived that answer is unacceptable, as it does not aid the learning or assessment process.

You will note a conspicuous lack of answers given for these conceptual questions. Unlike standard textbooks where answers to every other question are given somewhere toward the back of the book, here in these learning modules students must rely on other means to check their work. The best way by far is to debate the answers with fellow students and also with the instructor during the Socratic dialogue sessions intended to be used with these learning modules. Reasoning through challenging questions with other people is an excellent tool for developing strong reasoning skills.

Another means of checking your conceptual answers, where applicable, is to use circuit simulation software to explore the effects of changes made to circuits. For example, if one of these conceptual questions challenges you to predict the effects of altering some component parameter in a circuit, you may check the validity of your work by simulating that same parameter change within software and seeing if the results agree.

[3] *Analytical* thinking involves the "disassembly" of an idea into its constituent parts, analogous to dissection. *Synthetic* thinking involves the "assembly" of a new idea comprised of multiple concepts, analogous to construction. Both activities are high-level cognitive skills, extremely important for effective problem-solving, necessitating frequent challenge and regular practice to fully develop.

7.1.1 Reading outline and reflections

"Reading maketh a full man; conference a ready man; and writing an exact man" – Francis Bacon

Francis Bacon's advice is a blueprint for effective education: reading provides the learner with knowledge, writing focuses the learner's thoughts, and critical dialogue equips the learner to confidently communicate and apply their learning. Independent acquisition and application of knowledge is a powerful skill, well worth the effort to cultivate. To this end, students should read these educational resources closely, journal their own reflections on the reading, and discuss in detail their findings with classmates and instructor(s). You should be able to do all of the following after reading any instructional text:

☑ Briefly SUMMARIZE THE TEXT in the form of a journal entry documenting your learning as you progress through the course of study. Share this summary in dialogue with your classmates and instructor. Journaling is an excellent self-test of thorough reading because you cannot clearly express what you have not read or did not comprehend.

☑ Demonstrate ACTIVE READING STRATEGIES, including verbalizing your impressions as you read, simplifying long passages to convey the same ideas using fewer words, annotating text and illustrations with your own interpretations, working through mathematical examples shown in the text, cross-referencing passages with relevant illustrations and/or other passages, identifying problem-solving strategies applied by the author, etc. Technical reading is a special case of problem-solving, and so these strategies work precisely because they help solve any problem: paying attention to your own thoughts (metacognition), eliminating unnecessary complexities, identifying what makes sense, paying close attention to details, drawing connections between separated facts, and noting the successful strategies of others.

☑ Identify IMPORTANT THEMES, especially GENERAL LAWS and PRINCIPLES, expounded in the text and express them in the simplest of terms as though you were teaching an intelligent child. This emphasizes connections between related topics and develops your ability to communicate complex ideas to anyone.

☑ Form YOUR OWN QUESTIONS based on the reading, and then pose them to your instructor and classmates for their consideration. Anticipate both correct and incorrect answers, the incorrect answer(s) assuming one or more plausible misconceptions. This helps you view the subject from different perspectives to grasp it more fully.

☑ Devise EXPERIMENTS to test claims presented in the reading, or to disprove misconceptions. Predict possible outcomes of these experiments, and evaluate their meanings: what result(s) would confirm, and what would constitute disproof? Running mental simulations and evaluating results is essential to scientific and diagnostic reasoning.

☑ Specifically identify any points you found CONFUSING. The reason for doing this is to help diagnose misconceptions and overcome barriers to learning.

7.1.2 Foundational concepts

Correct analysis and diagnosis of electric circuits begins with a proper understanding of some basic concepts. The following is a list of some important concepts referenced in this module's full tutorial. Define each of them in your own words, and be prepared to illustrate each of these concepts with a description of a practical example and/or a live demonstration.

???

???

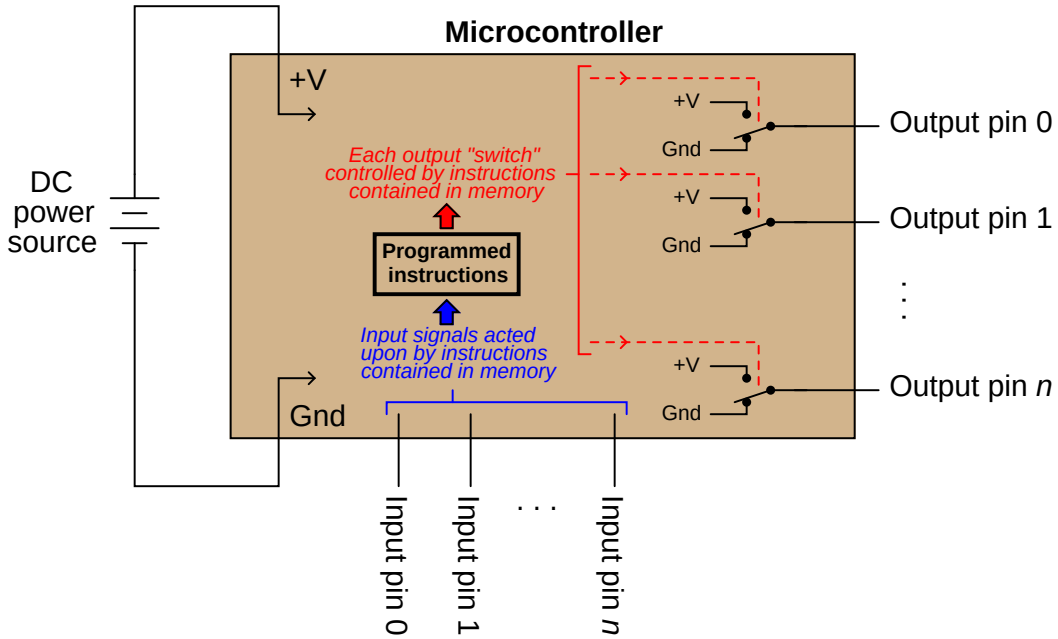
???

???

7.1.3 Microcontroller advantages

A *microcontroller unit*, or *MCU*, is a specialized type of digital computer used to provide automatic sequencing or control of a system. Microcontrollers differ from ordinary digital computers in being very small (typically a single integrated circuit chip), with several dedicated pins for input and/or output of digital signals, and limited memory. Instructions programmed into the microcontroller's memory tell it how to react to input conditions, and what types of signals to send to the outputs.

The simplest type of signal "understood" by a microcontroller is a discrete voltage level: either "high" (approximately +V) or "low" (approximately ground potential) measured at a specified pin on the chip. Transistors internal to the microcontroller produce these "high" and "low" signals at the output pins, their actions being modeled by SPDT switches for simplicity's sake:



Microcontrollers may be programmed to emulate the functions of digital logic gates (AND, OR, NAND, NOR, etc.) in addition to a wide variety of combinational and multivibrator functions. The only real limits to what a microcontroller can do are memory (how large of a program may be stored) and input/output pins on the MCU chip.

However, microcontrollers are themselves made up of many thousands (or millions!) of logic gate circuits. Why would it make sense to use a microcontroller to perform a logic function that a small fraction of its constituent gates could accomplish directly? In other words, why would anyone bother to program a microcontroller to perform a digital function when they could build the logic network they needed out of fewer gate circuits?

Challenges

- What might be some disadvantages of using a microcontroller instead of hard-wired logic gates to perform some function.

7.1.4 Second conceptual question

Challenges

- ???.
- ???.
- ???.

7.1.5 Applying foundational concepts to ???

Identify which foundational concept(s) apply to each of the declarations shown below regarding the following circuit. If a declaration is true, then identify it as such and note which concept supports that declaration; if a declaration is false, then identify it as such and note which concept is violated by that declaration:

(Under development)

- ???
- ???
- ???
- ???

Here is a list of foundational concepts for your reference: **Conservation of Energy**, **Conservation of Electric Charge**, **behavior of sources vs. loads**, **Ohm's Law**, **Joule's Law**, **effects of open faults**, **effect of shorted faults**, **properties of series networks**, **properties of parallel networks**, **Kirchhoff's Voltage Law**, **Kirchhoff's Current Law**. More than one of these concepts may apply to a declaration, and some concepts may not apply to any listed declaration at all. Also, feel free to include foundational concepts not listed here.

Challenges

- ???.
- ???.
- ???.

7.1.6 Explaining the meaning of calculations

Challenges

- ???.
- ???.
- ???.

7.2 Quantitative reasoning

These questions are designed to stimulate your computational thinking. In a Socratic discussion with your instructor, the goal is for these questions to reveal your mathematical approach(es) to problem-solving so that good technique and sound reasoning may be reinforced. Your instructor may also pose additional questions based on those assigned, in order to observe your problem-solving firsthand.

Mental arithmetic and estimations are strongly encouraged for all calculations, because without these abilities you will be unable to readily detect errors caused by calculator misuse (e.g. keystroke errors).

You will note a conspicuous lack of answers given for these quantitative questions. Unlike standard textbooks where answers to every other question are given somewhere toward the back of the book, here in these learning modules students must rely on other means to check their work. My advice is to use circuit simulation software such as SPICE to check the correctness of quantitative answers. Refer to those learning modules within this collection focusing on SPICE to see worked examples which you may use directly as practice problems for your own study, and/or as templates you may modify to run your own analyses and generate your own practice problems.

Completely worked example problems found in the Tutorial may also serve as "test cases[4]" for gaining proficiency in the use of circuit simulation software, and then once that proficiency is gained you will never need to rely[5] on an answer key!

[4] In other words, set up the circuit simulation software to analyze the same circuit examples found in the Tutorial. If the simulated results match the answers shown in the Tutorial, it confirms the simulation has properly run. If the simulated results disagree with the Tutorial's answers, something has been set up incorrectly in the simulation software. Using every Tutorial as practice in this way will quickly develop proficiency in the use of circuit simulation software.

[5] This approach is perfectly in keeping with the instructional philosophy of these learning modules: *teaching students to be self-sufficient thinkers*. Answer keys can be useful, but it is even more useful to your long-term success to have a set of tools on hand for checking your own work, because once you have left school and are on your own, there will no longer be "answer keys" available for the problems you will have to solve.

7.2.1 Miscellaneous physical constants

Note: constants shown in **bold** type are *exact*, not approximations. Values inside of parentheses show one standard deviation (σ) of uncertainty in the final digits: for example, the magnetic permeability of free space value given as $1.25663706212(19) \times 10^{-6}$ H/m represents a center value (i.e. the location parameter) of $1.25663706212 \times 10^{-6}$ Henrys per meter with one standard deviation of uncertainty equal to $0.0000000000019 \times 10^{-6}$ Henrys per meter.

Avogadro's number (N_A) = **6.02214076** $\times 10^{23}$ **per mole** (mol^{-1})

Boltzmann's constant (k) = **1.380649** $\times 10^{-23}$ **Joules per Kelvin** (J/K)

Electronic charge (e) = **1.602176634** $\times 10^{-19}$ **Coulomb** (C)

Faraday constant (F) = **96,485.33212...** $\times 10^{4}$ **Coulombs per mole** (C/mol)

Magnetic permeability of free space (μ_0) = $1.25663706212(19) \times 10^{-6}$ Henrys per meter (H/m)

Electric permittivity of free space (ϵ_0) = $8.8541878128(13) \times 10^{-12}$ Farads per meter (F/m)

Characteristic impedance of free space (Z_0) = 376.730313668(57) Ohms (Ω)

Gravitational constant (G) = $6.67430(15) \times 10^{-11}$ cubic meters per kilogram-seconds squared ($m^3/kg\text{-}s^2$)

Molar gas constant (R) = **8.314462618... Joules per mole-Kelvin** (J/mol-K) = 0.08205746(14) liters-atmospheres per mole-Kelvin

Planck constant (h) = **6.62607015** $\times 10^{-34}$ **joule-seconds** (J-s)

Stefan-Boltzmann constant (σ) = **5.670374419...** $\times 10^{-8}$ **Watts per square meter-Kelvin**^{4} ($W/m^2 \cdot K^4$)

Speed of light in a vacuum (c) = **299,792,458 meters per second** (m/s) = 186282.4 miles per second (mi/s)

Note: All constants taken from NIST data "Fundamental Physical Constants – Complete Listing", from `http://physics.nist.gov/constants`, National Institute of Standards and Technology (NIST), 2018 CODATA Adjustment.

7.2.2 Introduction to spreadsheets

A powerful computational tool you are encouraged to use in your work is a *spreadsheet*. Available on most personal computers (e.g. Microsoft Excel), *spreadsheet* software performs numerical calculations based on number values and formulae entered into cells of a grid. This grid is typically arranged as lettered columns and numbered rows, with each cell of the grid identified by its column/row coordinates (e.g. cell `B3`, cell `A8`). Each cell may contain a string of text, a number value, or a mathematical formula. The spreadsheet automatically updates the results of all mathematical formulae whenever the entered number values are changed. This means it is possible to set up a spreadsheet to perform a series of calculations on entered data, and those calculations will be re-done by the computer any time the data points are edited in any way.

For example, the following spreadsheet calculates average speed based on entered values of distance traveled and time elapsed:

	A	B	C	D
1	Distance traveled	46.9	Kilometers	
2	Time elapsed	1.18	Hours	
3	Average speed	= B1 / B2	km/h	
4				
5				

Text labels contained in cells `A1` through `A3` and cells `C1` through `C3` exist solely for readability and are not involved in any calculations. Cell `B1` contains a sample distance value while cell `B2` contains a sample time value. The formula for computing speed is contained in cell `B3`. Note how this formula begins with an "equals" symbol (`=`), references the values for distance and speed by lettered column and numbered row coordinates (`B1` and `B2`), and uses a forward slash symbol for division (`/`). The coordinates `B1` and `B2` function as *variables*[6] would in an algebraic formula.

When this spreadsheet is executed, the numerical value `39.74576` will appear in cell `B3` rather than the formula `= B1 / B2`, because 39.74576 is the computed speed value given 46.9 kilometers traveled over a period of 1.18 hours. If a different numerical value for distance is entered into cell `B1` or a different value for time is entered into cell `B2`, cell `B3`'s value will automatically update. All you need to do is set up the given values and any formulae into the spreadsheet, and the computer will do all the calculations for you.

Cell `B3` may be referenced by other formulae in the spreadsheet if desired, since it is a variable just like the given values contained in `B1` and `B2`. This means it is possible to set up an entire chain of calculations, one dependent on the result of another, in order to arrive at a final value. The arrangement of the given data and formulae need not follow any pattern on the grid, which means you may place them anywhere.

[6] Spreadsheets may also provide means to attach text labels to cells for use as variable names (Microsoft Excel simply calls these labels "names"), but for simple spreadsheets such as those shown here it's usually easier just to use the standard coordinate naming for each cell.

Common[7] arithmetic operations available for your use in a spreadsheet include the following:

- Addition (`+`)
- Subtraction (`-`)
- Multiplication (`*`)
- Division (`/`)
- Powers (`^`)
- Square roots (`sqrt()`)
- Logarithms (`ln()` , `log10()`)

Parentheses may be used to ensure[8] proper order of operations within a complex formula. Consider this example of a spreadsheet implementing the *quadratic formula*, used to solve for roots of a polynomial expression in the form of $ax^2 + bx + c$:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

	A	B
1	x_1	= (-B4 + sqrt((B4^2) - (4*B3*B5))) / (2*B3)
2	x_2	= (-B4 - sqrt((B4^2) - (4*B3*B5))) / (2*B3)
3	a =	9
4	b =	5
5	c =	-2

This example is configured to compute roots[9] of the polynomial $9x^2 + 5x - 2$ because the values of 9, 5, and -2 have been inserted into cells `B3`, `B4`, and `B5`, respectively. Once this spreadsheet has been built, though, it may be used to calculate the roots of *any* second-degree polynomial expression simply by entering the new a, b, and c coefficients into cells `B3` through `B5`. The numerical values appearing in cells `B1` and `B2` will be automatically updated by the computer immediately following any changes made to the coefficients.

[7] Modern spreadsheet software offers a bewildering array of mathematical functions you may use in your computations. I recommend you consult the documentation for your particular spreadsheet for information on operations other than those listed here.

[8] Spreadsheet programs, like text-based programming languages, are designed to follow standard order of operations by default. However, my personal preference is to use parentheses even where strictly unnecessary just to make it clear to any other person viewing the formula what the intended order of operations is.

[9] Reviewing some algebra here, a *root* is a value for x that yields an overall value of zero for the polynomial. For this polynomial ($9x^2 + 5x - 2$) the two roots happen to be $x = 0.269381$ and $x = -0.82494$, with these values displayed in cells `B1` and `B2`, respectively upon execution of the spreadsheet.

Alternatively, one could break up the long quadratic formula into smaller pieces like this:

$$y = \sqrt{b^2 - 4ac} \qquad z = 2a$$

$$x = \frac{-b \pm y}{z}$$

	A	B	C
1	x_1	= (-B4 + C1) / C2	= sqrt((B4^2) - (4*B3*B5))
2	x_2	= (-B4 - C1) / C2	= 2*B3
3	a =	9	
4	b =	5	
5	c =	-2	

Note how the square-root term (y) is calculated in cell C1, and the denominator term (z) in cell C2. This makes the two final formulae (in cells B1 and B2) simpler to interpret. The positioning of all these cells on the grid is completely arbitrary[10] – all that matters is that they properly reference each other in the formulae.

Spreadsheets are particularly useful for situations where the same set of calculations representing a circuit or other system must be repeated for different initial conditions. The power of a spreadsheet is that it automates what would otherwise be a tedious set of calculations. One specific application of this is to simulate the effects of various components within a circuit failing with abnormal values (e.g. a shorted resistor simulated by making its value nearly zero; an open resistor simulated by making its value extremely large). Another application is analyzing the behavior of a circuit design given new components that are out of specification, and/or aging components experiencing drift over time.

[10] My personal preference is to locate all the "given" data in the upper-left cells of the spreadsheet grid (each data point flanked by a sensible name in the cell to the left and units of measurement in the cell to the right as illustrated in the first distance/time spreadsheet example), sometimes coloring them in order to clearly distinguish which cells contain entered data versus which cells contain computed results from formulae. I like to place all formulae in cells below the given data, and try to arrange them in logical order so that anyone examining my spreadsheet will be able to figure out *how* I constructed a solution. This is a general principle I believe all computer programmers should follow: *document and arrange your code to make it easy for other people to learn from it.*

7.2.3 First quantitative problem

Challenges

- ???.
- ???.
- ???.

7.2.4 Second quantitative problem

Challenges

- ???.
- ???.
- ???.

7.3 Diagnostic reasoning

These questions are designed to stimulate your deductive and inductive thinking, where you must apply general principles to specific scenarios (deductive) and also derive conclusions about the failed circuit from specific details (inductive). In a Socratic discussion with your instructor, the goal is for these questions to reinforce your recall and use of general circuit principles and also challenge your ability to integrate multiple symptoms into a sensible explanation of what's wrong in a circuit. Your instructor may also pose additional questions based on those assigned, in order to further challenge and sharpen your diagnostic abilities.

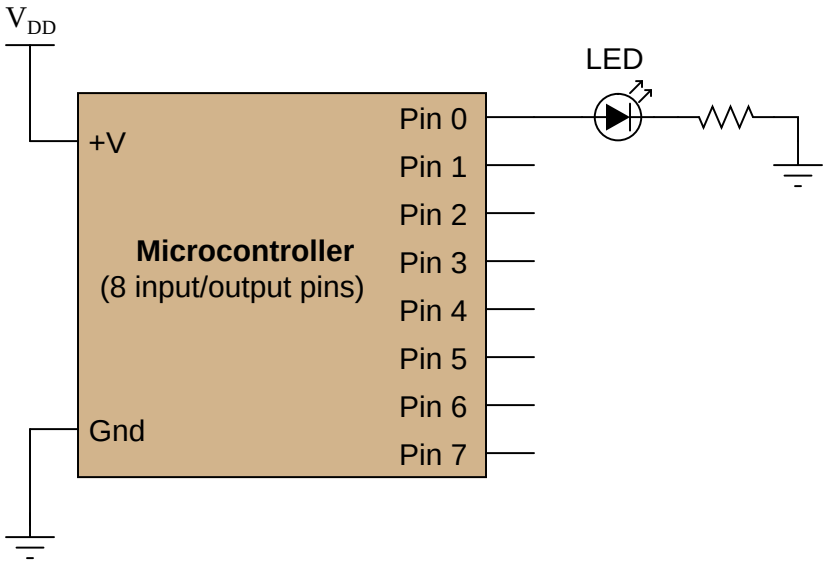
As always, your goal is to fully *explain* your analysis of each problem. Simply obtaining a correct answer is not good enough – you must also demonstrate sound reasoning in order to successfully complete the assignment. Your instructor's responsibility is to probe and challenge your understanding of the relevant principles and analytical processes in order to ensure you have a strong foundation upon which to build further understanding.

You will note a conspicuous lack of answers given for these diagnostic questions. Unlike standard textbooks where answers to every other question are given somewhere toward the back of the book, here in these learning modules students must rely on other means to check their work. The best way by far is to debate the answers with fellow students and also with the instructor during the Socratic dialogue sessions intended to be used with these learning modules. Reasoning through challenging questions with other people is an excellent tool for developing strong reasoning skills.

Another means of checking your diagnostic answers, where applicable, is to use circuit simulation software to explore the effects of faults placed in circuits. For example, if one of these diagnostic questions requires that you predict the effect of an open or a short in a circuit, you may check the validity of your work by simulating that same fault (substituting a very high resistance in place of that component for an open, and substituting a very low resistance for a short) within software and seeing if the results agree.

7.3.1 Faulty light-blinking program

A student decides to build a light-flasher circuit using a microcontroller instead of a 555 timer or some other hard-wired astable circuit. Unfortunately, there is a problem somewhere. When first powered up, the LED lights on for 1 second, then turns off and never turns back on. The only way the LED ever comes back on is if the MCU is reset or its power is cycled off and on:



Pseudocode listing

```
Declare Pin0 as an output

BEGIN
  Set Pin0 HIGH
  Pause for 1 second
  Set Pin0 LOW
END
```

A fellow student, when asked for help, modifies the program listing and re-sends it from the personal computer where it is being edited to the microcontroller, through a programming cable. The program listing now reads as such:

Pseudocode listing

```
Declare Pin0 as an output

LOOP
  Set Pin0 HIGH
  Pause for 1 second
  Set Pin0 LOW
ENDLOOP
```

When the MCU is reset with the new program, the LED starts blinking on and off . . . sort of. The LED is "on" most of the time, but once every second it turns off and then immediately comes back on. In fact, the "off" period is so brief it is barely noticeable.

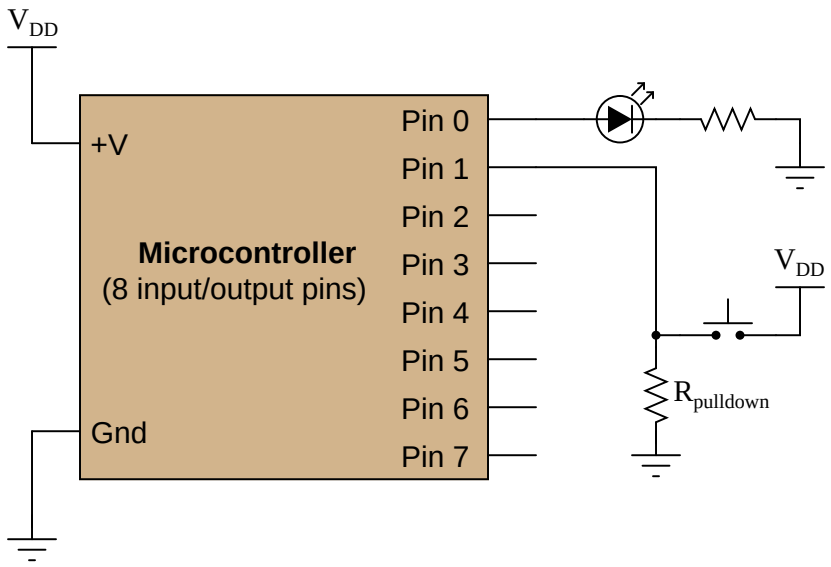
What the student wanted was a 50% duty cycle: "on" for 1 second, then "off" for 1 second, repeating that cycle indefinitely. First, explain the significance of the classmate's program modification, and then modify the program listing again so that the LED does what the student wants it to.

Challenges

- ???.
- ???.
- ???.

7.3.2 Another faulty light-blinking program

A student decides to build a light-flasher circuit using a microcontroller. The LED is supposed to blink on and off only when the pushbutton switch is depressed. It is supposed to turn off when the switch is released:



Pseudocode listing

```
Declare Pin0 as an output
Declare Pin1 as an input

WHILE Pin1 is HIGH
  Set Pin0 HIGH
  Pause for 0.5 seconds
  Set Pin0 LOW
  Pause for 0.5 seconds
ENDWHILE
```

The LED blinks on and off just fine as long as the pushbutton switch is held when the MCU is powered up or reset. As soon as the switch is released, the LED turns off and never comes back on. If the switch was never pressed during start-up, the LED never comes on! Explain what is happening, and modify the program as necessary to fix this problem.

Challenges

- What purpose does the resistor $R_{pulldown}$ serve in the pushbutton circuit?

Chapter 8

Projects and Experiments

The following project and experiment descriptions outline things you can build to help you understand circuits. With any real-world project or experiment there exists the potential for physical harm. *Electricity can be very dangerous in certain circumstances, and you should follow proper safety precautions at all times!*

8.1 Recommended practices

This section outlines some recommended practices for all circuits you design and construct.

8.1.1 Safety first!

Electricity, when passed through the human body, causes uncomfortable sensations and in large enough measures[1] will cause muscles to involuntarily contract. The overriding of your nervous system by the passage of electrical current through your body is particularly dangerous in regard to your heart, which is a vital muscle. Very large amounts of current can produce serious internal burns in addition to all the other effects.

Cardio-pulmonary resuscitation (CPR) is the standard first-aid for any victim of electrical shock. This is a very good skill to acquire if you intend to work with others on dangerous electrical circuits. You should never perform tests or work on such circuits unless someone else is present who is proficient in CPR.

As a general rule, any voltage in excess of 30 Volts poses a definitive electric shock hazard, because beyond this level human skin does not have enough resistance to safely limit current through the body. "Live" work of any kind with circuits over 30 volts should be avoided, and if unavoidable should only be done using electrically insulated tools and other protective equipment (e.g. insulating shoes and gloves). If you are unsure of the hazards, or feel unsafe at any time, stop all work and distance yourself from the circuit!

A policy I strongly recommend for students learning about electricity is to *never come into electrical contact*[2] *with an energized conductor, no matter what the circuit's voltage*[3] *level!* Enforcing this policy may seem ridiculous when the circuit in question is powered by a single battery smaller than the palm of your hand, but it is precisely this instilled habit which will save a person from bodily harm when working with more dangerous circuits. Experience has taught me that students who learn early on to be careless with safe circuits have a tendency to be careless later with dangerous circuits!

In addition to the electrical hazards of shock and burns, the construction of projects and running of experiments often poses other hazards such as working with hand and power tools, potential

[1] Professor Charles Dalziel published a research paper in 1961 called "The Deleterious Effects of Electric Shock" detailing the results of electric shock experiments with both human and animal subjects. The threshold of perception for human subjects holding a conductor in their hand was in the range of 1 milliampere of current (less than this for alternating current, and generally less for female subjects than for male). Loss of muscular control was exhibited by half of Dalziel's subjects at less than 10 milliamperes alternating current. Extreme pain, difficulty breathing, and loss of all muscular control occurred for over 99% of his subjects at direct currents less than 100 milliamperes and alternating currents less than 30 milliamperes. In summary, it doesn't require much electric current to induce painful and even life-threatening effects in the human body! Your first and best protection against electric shock is maintaining an insulating barrier between your body and the circuit in question, such that current from that circuit will be unable to flow through your body.

[2] By "electrical contact" I mean either directly touching an energized conductor with any part of your body, or indirectly touching it through a conductive tool. The only physical contact you should ever make with an energized conductor is via an electrically insulated tool, for example a screwdriver with an electrically insulated handle, or an insulated test probe for some instrument.

[3] Another reason for consistently enforcing this policy, even on low-voltage circuits, is due to the dangers that even some low-voltage circuits harbor. A single 12 Volt automobile battery, for example, can cause a surprising amount of damage if short-circuited simply due to the high current levels (i.e. very low internal resistance) it is capable of, even though the voltage level is too low to cause a shock through the skin. Mechanics wearing metal rings, for example, are at risk from severe burns if their rings happen to short-circuit such a battery! Furthermore, even when working on circuits that are simply too low-power (low voltage and low current) to cause any bodily harm, touching them while energized can pose a threat to the circuit components themselves. In summary, it generally wise (and *always* a good habit to build) to "power down" *any* circuit before making contact between it and your body.

contact with high temperatures, potential chemical exposure, etc. You should never proceed with a project or experiment if you are unaware of proper tool use or lack basic protective measures (e.g. personal protective equipment such as safety glasses) against such hazards.

Some other safety-related practices should be followed as well:

- All power conductors extending outward from the project must be *firmly* strain-relieved (e.g. "cord grips" used on line power cords), so that an accidental tug or drop will not compromise circuit integrity.
- All electrical connections must be sound and appropriately made (e.g. soldered wire joints rather than twisted-and-taped; terminal blocks rather than solderless breadboards for high-current or high-voltage circuits). Use "touch-safe" terminal connections with recessed metal parts to minimize risk of accidental contact.
- Always provide overcurrent protection in any circuit you build. *Always.* This may be in the form of a fuse, a circuit breaker, and/or an electronically current-limited power supply.
- Always ensure circuit conductors are rated for more current than the overcurrent protection limit. *Always.* A fuse does no good if the wire or printed circuit board trace will "blow" before it does!
- Always bond metal enclosures to Earth ground for any line-powered circuit. *Always.* Ensuring an equipotential state between the enclosure and Earth by making the enclosure electrically common with Earth ground ensures no electric shock can occur simply by one's body bridging between the Earth and the enclosure.
- Avoid building a high-energy circuit when a low-energy circuit will suffice. For example, I always recommend beginning students power their first DC resistor circuits using small batteries rather than with line-powered DC power supplies. The intrinsic energy limitations of a dry-cell battery make accidents highly unlikely.
- Use line power receptacles that are GFCI (Ground Fault Current Interrupting) to help avoid electric shock from making accidental contact with a "hot" line conductor.
- Always wear eye protection when working with tools or live systems having the potential to eject material into the air. Examples of such activities include soldering, drilling, grinding, cutting, wire stripping, working on or near energized circuits, etc.
- Always use a step-stool or stepladder to reach high places. Never stand on something not designed to support a human load.
- When in doubt, *ask an expert.* If anything even seems remotely unsafe to you, do not proceed without consulting a trusted person fully knowledgeable in electrical safety.

8.1.2 Other helpful tips

Experience has shown the following practices to be very helpful, especially when students make their own component selections, to ensure the circuits will be well-behaved:

- Avoid resistor values less than 1 kΩ or greater than 100 kΩ, unless such values are definitely necessary[4]. Resistances below 1 kΩ may draw excessive current if directly connected to a voltage source of significant magnitude, and may also complicate the task of accurately measuring current since any ammeter's non-zero resistance inserted in series with a low-value circuit resistor will significantly alter the total resistance and thereby skew the measurement. Resistances above 100 kΩ may complicate the task of measuring voltage since any voltmeter's finite resistance connected in parallel with a high-value circuit resistor will significantly alter the total resistance and thereby skew the measurement. Similarly, AC circuit impedance values should be between 1 kΩ and 100 kΩ, and for all the same reasons.

- Ensure all electrical connections are low-resistance and physically rugged. For this reason, one should avoid *compression splices* (e.g. "butt" connectors), solderless breadboards[5], and wires that are simply twisted together.

- Build your circuit with **testing** in mind. For example, provide convenient connection points for test equipment (e.g. multimeters, oscilloscopes, signal generators, logic probes).

- Design permanent projects with **maintenance** in mind. The more convenient you make maintenance tasks, the more likely they will get done.

- **Always document and save your work**. Circuits lacking schematic diagrams are more difficult to troubleshoot than documented circuits. Similarly, circuit construction is simpler when a schematic diagram precedes construction. Experimental results are easier to interpret when comprehensively recorded. Consider modern videorecording technology for this purpose where appropriate.

- **Record your steps** when troubleshooting. **Talk to yourself** when solving problems. These simple steps clarify thought and simplify identification of errors.

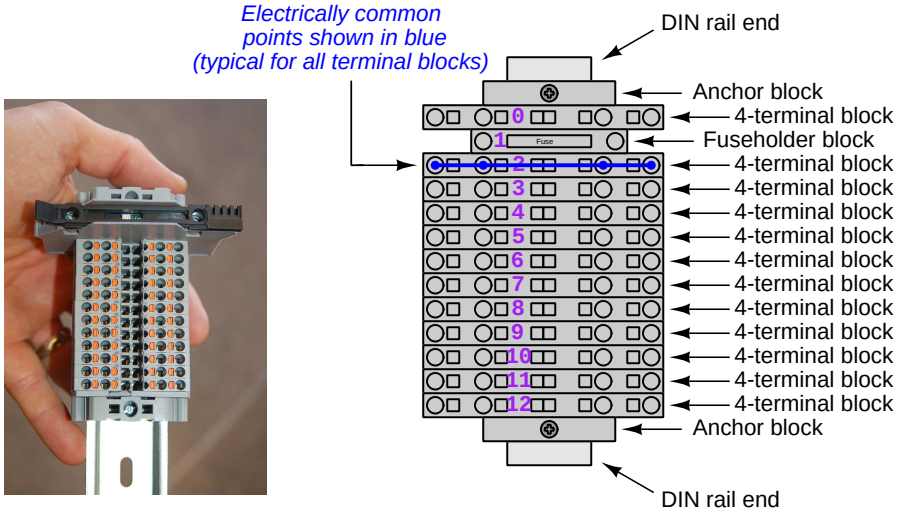
[4] An example of a necessary resistor value much less than 1 kΩ is a *shunt resistor* used to produce a small voltage drop for the purpose of sensing current in a circuit. Such shunt resistors must be low-value in order not to impose an undue load on the rest of the circuit. An example of a necessary resistor value much greater than 100 kΩ is an electrostatic *drain resistor* used to dissipate stored electric charges from body capacitance for the sake of preventing damage to sensitive semiconductor components, while also preventing a path for current that could be dangerous to the person (i.e. shock).

[5] Admittedly, solderless breadboards are very useful for constructing complex electronic circuits with many components, especially DIP-style integrated circuits (ICs), but they tend to give trouble with connection integrity after frequent use. An alternative for projects using low counts of ICs is to solder IC sockets into prototype printed circuit boards (PCBs) and run wires from the soldered pins of the IC sockets to terminal blocks where reliable temporary connections may be made.

8.1.3 Terminal blocks for circuit construction

Terminal blocks are the standard means for making electric circuit connections in industrial systems. They are also quite useful as a learning tool, and so I highly recommend their use in lieu of solderless breadboards[6]. Terminal blocks provide highly reliable connections capable of withstanding significant voltage and current magnitudes, and they force the builder to think very carefully about component layout which is an important mental practice. Terminal blocks that mount on standard 35 mm DIN rail[7] are made in a wide range of types and sizes, some with built-in disconnecting switches, some with built-in components such as rectifying diodes and fuseholders, all of which facilitate practical circuit construction.

I recommend every student of electricity build their own terminal block array for use in constructing experimental circuits, consisting of several terminal blocks where each block has at least 4 connection points all electrically common to each other[8] and at least one terminal block that is a fuse holder for overcurrent protection. A pair of anchoring blocks hold all terminal blocks securely on the DIN rail, preventing them from sliding off the rail. Each of the terminals should bear a number, starting from 0. An example is shown in the following photograph and illustration:



Screwless terminal blocks (using internal spring clips to clamp wire and component lead ends) are preferred over screw-based terminal blocks, as they reduce assembly and disassembly time, and also minimize repetitive wrist stress from twisting screwdrivers. Some screwless terminal blocks require the use of a special tool to release the spring clip, while others provide buttons[9] for this task which may be pressed using the tip of any suitable tool.

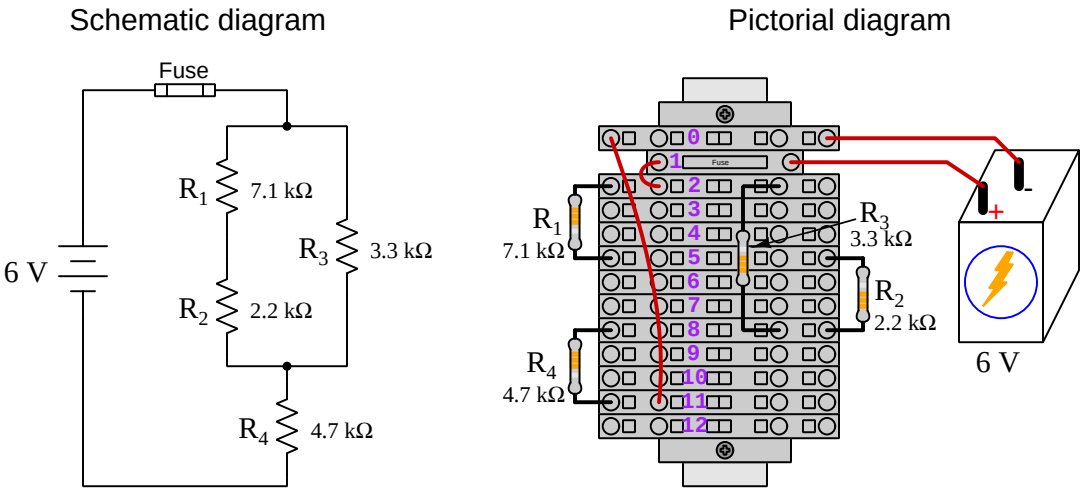
[6] Solderless breadboard are preferable for complicated electronic circuits with multiple integrated "chip" components, but for simpler circuits I find terminal blocks much more practical. An alternative to solderless breadboards for "chip" circuits is to solder chip sockets onto a PCB and then use wires to connect the socket pins to terminal blocks. This also accommodates *surface-mount* components, which solderless breadboards do not.

[7] DIN rail is a metal rail designed to serve as a mounting point for a wide range of electrical and electronic devices such as terminal blocks, fuses, circuit breakers, relay sockets, power supplies, data acquisition hardware, etc.

[8] Sometimes referred to as *equipotential*, *same-potential*, or *potential distribution* terminal blocks.

[9] The small orange-colored squares seen in the above photograph are buttons for this purpose, and may be actuated by pressing with any tool of suitable size.

The following example shows how such a terminal block array might be used to construct a series-parallel resistor circuit consisting of four resistors and a battery:



Numbering on the terminal blocks provides a very natural translation to SPICE[10] netlists, where component connections are identified by terminal number:

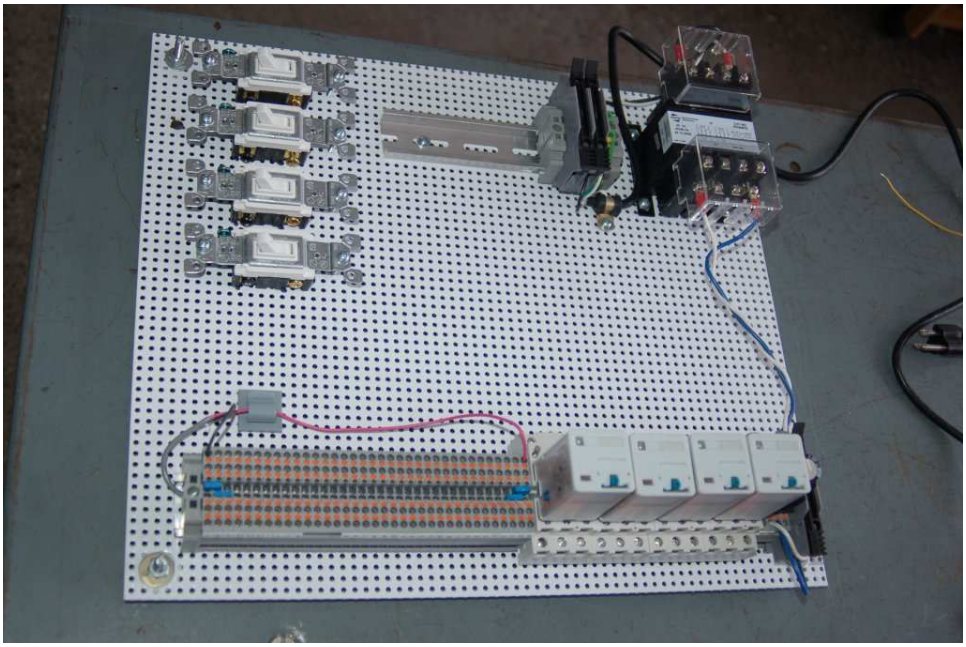
```
* Series-parallel resistor circuit
v1 1 0 dc 6
r1 2 5 7100
r2 5 8 2200
r3 2 8 3300
r4 8 11 4700
rjmp1 1 2 0.01
rjmp2 0 11 0.01
.op
.end
```

Note the use of "jumper" resistances `rjmp1` and `rjmp2` to describe the wire connections between terminals 1 and 2 and between terminals 0 and 11, respectively. Being resistances, SPICE requires a resistance value for each, and here we see they have both been set to an arbitrarily low value of 0.01 Ohm realistic for short pieces of wire.

Listing all components and wires along with their numbered terminals happens to be a useful documentation method for any circuit built on terminal blocks, independent of SPICE. Such a "wiring sequence" may be thought of as a *non-graphical description* of an electric circuit, and is exceptionally easy to follow.

[10] SPICE is computer software designed to analyze electrical and electronic circuits. Circuits are described for the computer in the form of *netlists* which are text files listing each component type, connection node numbers, and component values.

An example of a more elaborate terminal block array is shown in the following photograph, with terminal blocks and "ice-cube" style electromechanical relays mounted to DIN rail, which is turn mounted to a perforated subpanel[11]. This "terminal block board" hosts an array of thirty five undedicated terminal block sections, four SPDT toggle switches, four DPDT "ice-cube" relays, a step-down control power transformer, bridge rectifier and filtering capacitor, and several fuses for overcurrent protection:



Four plastic-bottomed "feet" support the subpanel above the benchtop surface, and an unused section of DIN rail stands ready to accept other components. Safety features include electrical bonding of the AC line power cord's ground to the metal subpanel (and all metal DIN rails), mechanical strain relief for the power cord to isolate any cord tension from wire connections, clear plastic finger guards covering the transformer's screw terminals, as well as fused overcurrent protection for the 120 Volt AC line power and the transformer's 12 Volt AC output. The perforated holes happen to be on $\frac{1}{4}$ inch centers with a diameter suitable for tapping with 6-32 machine screw threads, their presence making it very easy to attach other sections of DIN rail, printed circuit boards, or specialized electrical components directly to the grounded metal subpanel. Such a "terminal block board" is an inexpensive[12] yet highly flexible means to construct physically robust circuits using industrial wiring practices.

[11] An electrical *subpanel* is a thin metal plate intended for mounting inside an electrical enclosure. Components are attached to the subpanel, and the subpanel in turn bolts inside the enclosure. Subpanels allow circuit construction outside the confines of the enclosure, which speeds assembly. In this particular usage there is no enclosure, as the subpanel is intended to be used as an open platform for the convenient construction of circuits on a benchtop by students. In essence, this is a modern version of the traditional *breadboard* which was literally a wooden board such as might be used for cutting loaves of bread, but which early electrical and electronic hobbyists used as platforms for the construction of circuits.

[12] At the time of this writing (2019) the cost to build this board is approximately $250 US dollars.

8.1.4 Conducting experiments

An *experiment* is an exploratory act, a test performed for the purpose of assessing some proposition or principle. Experiments are the foundation of the *scientific method*, a process by which careful observation helps guard against errors of speculation. All good experiments begin with an *hypothesis*, defined by the American Heritage Dictionary of the English Language as:

> An assertion subject to verification or proof, as (a) A proposition stated as a basis for argument or reasoning. (b) A premise from which a conclusion is drawn. (c) A conjecture that accounts, within a theory or ideational framework, for a set of facts and that can be used as a basis for further investigation.

Stated plainly, an hypothesis is an *educated guess* about cause and effect. The correctness of this initial guess matters little, because any well-designed experiment will reveal the truth of the matter. In fact, *incorrect* hypotheses are often the most valuable because the experiments they engender lead us to surprising discoveries. One of the beautiful aspects of science is that it is more focused on the process of *learning* than about the status of *being correct*[13]. In order for an hypothesis to be valid, it must be testable[14], which means it must be a claim possible to refute given the right data. Hypotheses impossible to critique are useless.

Once an hypothesis has been formulated, an experiment must be designed to test that hypothesis. A well-designed experiment requires careful regulation of all relevant variables, both for personal safety and for prompting the hypothesized results. If the effects of one particular variable are to be tested, the experiment must be run multiple times with different values of (only) that particular variable. The experiment set up with the "baseline" variable set is called the *control*, while the experiment set up with different value(s) is called the *test* or *experimental*.

For some hypotheses a viable alternative to a physical experiment is a *computer-simulated experiment* or even a *thought experiment*. Simulations performed on a computer test the hypothesis against the physical laws encoded within the computer simulation software, and are particularly useful for students learning new principles for which simulation software is readily available[15].

[13] Science is more about clarifying our view of the universe through a systematic process of error detection than it is about proving oneself to be right. Some scient*ists* may happen to have large egos – and this may have more to do with the ways in which large-scale scientific research is *funded* than anything else – but scientific *method* itself is devoid of ego, and if embraced as a practical philosophy is quite an effective stimulant for humility. Within the education system, scientific method is particularly valuable for helping students break free of the crippling fear of *being wrong*. So much emphasis is placed in formal education on assessing correct retention of facts that many students are fearful of saying or doing anything that might be perceived as a mistake, and of course making mistakes (i.e. having one's hypotheses disproven by experiment) is an indispensable tool for learning. Introducing science in the classroom – *real* science characterized by individuals forming actual hypotheses and testing those hypotheses by experiment – helps students become self-directed learners.

[14] This is the principle of *falsifiability*: that a scientific statement has value only insofar as it is liable to disproof given the requisite experimental evidence. Any claim that is unfalsifiable – that is, a claim which can *never* be disproven by any evidence whatsoever – could be completely wrong and we could never know it.

[15] A very pertinent example of this is learning how to analyze electric circuits using simulation software such as SPICE. A typical experimental cycle would proceed as follows: (1) Find or invent a circuit to analyze; (2) Apply your analytical knowledge to that circuit, predicting all voltages, currents, powers, etc. relevant to the concepts you are striving to master; (3) Run a simulation on that circuit, collecting "data" from the computer when complete; (4) Evaluate whether or not your hypotheses (i.e. predicted voltages, currents, etc.) agree with the computer-generated results; (5) If so, your analyses are (provisionally) correct – if not, examine your analyses and the computer simulation again to determine the source of error; (6) Repeat this process as many times as necessary until you achieve mastery.

Thought experiments are useful for detecting inconsistencies within your own understanding of some subject, rather than testing your understanding against physical reality.

Here are some general guidelines for conducting experiments:

- The clearer and more specific the hypothesis, the better. Vague or unfalsifiable hypotheses are useless because they will fit *any* experimental results, and therefore the experiment cannot teach you anything about the hypothesis.
- Collect as much data (i.e. information, measurements, sensory experiences) generated by an experiment as is practical. This includes the time and date of the experiment, too!
- *Never* discard or modify data gathered from an experiment. If you have reason to believe the data is unreliable, write notes to that effect, but never throw away data just because you think it is untrustworthy. It is quite possible that even "bad" data holds useful information, and that someone else may be able to uncover its value even if you do not.
- Prioritize *quantitative* data over *qualitative* data wherever practical. Quantitative data is more specific than qualitative, less prone to subjective interpretation on the part of the experimenter, and amenable to an arsenal of analytical methods (e.g. statistics).
- Guard against your own bias(es) by making your experimental results available to others. This allows other people to scrutinize your experimental design and collected data, for the purpose of detecting and correcting errors you may have missed. Document your experiment such that others may independently replicate it.
- Always be looking for sources of error. No physical measurement is perfect, and so it is impossible to achieve *exact* values for any variable. Quantify the amount of uncertainty (i.e. the "tolerance" of errors) whenever possible, and be sure your hypothesis does not depend on precision better than this!
- Always remember that scientific confirmation is provisional – no number of "successful" experiments will prove an hypothesis true for all time, but a single experiment can disprove it. Put into simpler terms, *truth is elusive but error is within reach.*
- Remember that scientific method is about *learning*, first and foremost. An unfortunate consequence of scientific triumph in modern society is that science is often viewed by non-practitioners as an unerring source of truth, when in fact science is an ongoing process of challenging existing ideas to probe for errors and oversights. This is why it is perfectly acceptable to have a failed hypothesis, and why the only truly failed experiment is one where nothing was learned.

The following is an example of a well-planned and executed experiment, in this case a physical experiment demonstrating Ohm's Law.

```
Planning Time/Date = 09:30 on 12 February 2019

HYPOTHESIS: the current through any resistor should be exactly proportional
to the voltage impressed across it.

PROCEDURE: connect a resistor rated 1 k Ohm and 1/4 Watt to a variable-voltage
DC power supply.  Use an ammeter in series to measure resistor current and
a voltmeter in parallel to measure resistor voltage.

RISKS AND MITIGATION: excessive power dissipation may harm the resistor and/
or pose a burn hazard, while excessive voltage poses an electric shock hazard.
30 Volts is a safe maximum voltage for laboratory practices, and according to
Joule's Law a 1000 Ohm resistor will dissipate 0.25 Watts at 15.81 Volts
(P = V^2 / R), so I will remain below 15 Volts just to be safe.

Experiment Time/Date = 10:15 on 12 February 2019

DATA COLLECTED:
  (Voltage)    (Current)         (Voltage)   (Current)
   0.000 V   =  0.000 mA          8.100    =  7.812 mA
   2.700 V   =  2.603 mA          10.00 V  =  9.643 mA
   5.400 V   =  5.206 mA          14.00 V  =  13.49 mA

Analysis Time/Date = 10:57 on 12 February 2019

ANALYSIS: current definitely increases with voltage, and although I expected
exactly one milliAmpere per Volt the actual current was usually less than
that.  The voltage/current ratios ranged from a low of 1036.87 (at 8.1 Volts)
to a high of 1037.81 (at 14 Volts), but this represents a variance of only
-0.0365% to +0.0541% from the average, indicating a very consistent
proportionality -- results consistent with Ohm's Law.

ERROR SOURCES: one major source of error is the resistor's value itself.  I
did not measure it, but simply assumed color bands of brown-black-red meant
exactly 1000 Ohms.  Based on the data I think the true resistance is closer
to 1037 Ohms.  Another possible explanation is multimeter calibration error.
However, neither explains the small positive and negative variances from the
average.  This might be due to electrical noise, a good test being to repeat
the same experiment to see if the variances are the same or different.  Noise
should generate slightly different results every time.
```

The following is an example of a well-planned and executed *virtual* experiment, in this case demonstrating Ohm's Law using a computer (SPICE) simulation.

```
Planning Time/Date = 12:32 on 14 February 2019

HYPOTHESIS: for any given resistor, the current through that resistor should be
exactly proportional to the voltage impressed across it.

PROCEDURE: write a SPICE netlist with a single DC voltage source and single
1000 Ohm resistor, then use NGSPICE version 26 to perform a "sweep" analysis
from 0 Volts to 25 Volts in 5 Volt increments.

    * SPICE circuit
    v1 1 0 dc
    r1 1 0 1000
    .dc v1 0 25 5
    .print dc v(1) i(v1)
    .end

RISKS AND MITIGATION: none.

DATA COLLECTED:
       DC transfer characteristic  Thu Feb 14 13:05:08  2019
    ------------------------------------------------------------
    Index   v-sweep         v(1)            v1#branch
    ------------------------------------------------------------
    0       0.000000e+00    0.000000e+00    0.000000e+00
    1       5.000000e+00    5.000000e+00    -5.00000e-03
    2       1.000000e+01    1.000000e+01    -1.00000e-02
    3       1.500000e+01    1.500000e+01    -1.50000e-02
    4       2.000000e+01    2.000000e+01    -2.00000e-02
    5       2.500000e+01    2.500000e+01    -2.50000e-02

Analysis Time/Date = 13:06 on 14 February 2019

ANALYSIS: perfect agreement between data and hypothesis -- current is precisely
1/1000 of the applied voltage for all values.  Anything other than perfect
agreement would have probably meant my netlist was incorrect.  The negative
current values surprised me, but it seems this is just how SPICE interprets
normal current through a DC voltage source.

ERROR SOURCES: none.
```

As gratuitous as it may seem to perform experiments on a physical law as well-established as Ohm's Law, even the examples listed previously demonstrate opportunity for real learning. In the physical experiment example, the student should identify and explain why their data does not perfectly agree with the hypothesis, and this leads them naturally to consider sources of error. In the computer-simulated experiment, the student is struck by SPICE's convention of denoting regular current through a DC voltage source as being *negative* in sign, and this is also useful knowledge for future simulations. Scientific experiments are most interesting when things *do not* go as planned!

Aside from verifying well-established physical laws, simple experiments are extremely useful as educational tools for a wide range of purposes, including:

- Component familiarization (e.g. *Which terminals of this switch connect to the NO versus NC contacts?*)
- System testing (e.g. *How heavy of a load can my AC-DC power supply source before the semiconductor components reach their thermal limits?*)
- Learning programming languages (e.g. *Let's try to set up an "up" counter function in this PLC!*)

Above all, the priority here is to inculcate the habit of hypothesizing, running experiments, and analyzing the results. This experimental cycle not only serves as an excellent method for self-directed learning, but it also works exceptionally well for troubleshooting faults in complex systems, and for these reasons should be a part of every technician's and every engineer's education.

8.1.5 Constructing projects

Designing, constructing, and testing projects is a very effective means of practical education. Within a formal educational setting, projects are generally chosen (or at least vetted) by an instructor to ensure they may be reasonably completed within the allotted time of a course or program of study, and that they sufficiently challenge the student to learn certain important principles. In a self-directed environment, projects are just as useful as a learning tool but there is some risk of unwittingly choosing a project beyond one's abilities, which can lead to frustration.

Here are some general guidelines for managing projects:

- Define your goal(s) before beginning a project: what do you wish to achieve in building it? What, exactly, should the completed project *do*?
- Analyze your project prior to construction. Document it in appropriate forms (e.g. schematic diagrams), predict its functionality, anticipate all associated risks. In other words, *plan ahead*.
- Set a reasonable budget for your project, and stay within it.
- Identify any deadlines, and set reasonable goals to meet those deadlines.
- Beware of *scope creep*: the tendency to modify the project's goals before it is complete.
- Document your progress! An easy way to do this is to use photography or videography: take photos and/or videos of your project as it progresses. Document failures as well as successes, because both are equally valuable from the perspective of learning.

8.2 Experiment: (first experiment)

Conduct an experiment to . . .

EXPERIMENT CHECKLIST:

- Prior to experimentation:
  - ☑ Write an hypothesis (i.e. a detailed description of what you expect will happen) unambiguous enough that it could be disproven given the right data.
  - ☑ Write a procedure to test the hypothesis, complete with adequate controls and documentation (e.g. schematic diagrams, programming code).
  - ☑ Identify any risks (e.g. shock hazard, component damage) and write a mitigation plan based on best practices and component ratings.
- During experimentation:
  - ☑ Safe practices followed at all times (e.g. no contact with energized circuit).
  - ☑ Correct equipment usage according to manufacturer's recommendations.
  - ☑ All data collected, ideally quantitative with full precision (i.e. no rounding).
- After each experimental run:
  - ☑ If the results fail to match the hypothesis, identify the error(s), correct the hypothesis and/or revise the procedure, and re-run the experiment.
  - ☑ Identify any uncontrolled sources of error in the experiment.
- After all experimental re-runs:
  - ☑ Save all data for future reference.
  - ☑ Write an analysis of experimental results and lessons learned.

Challenges

- Science is an *iterative* process, and for this reason is never complete. Following the results of your experiment, what would you propose for your *next* hypothesis and *next* experimental procedure? Hint: if your experiment produced any unexpected results, exploring those unexpected results is often a very good basis for the next experiment!
- ???.
- ???.

8.3 Project: (first project)

This is a description of the project!

PROJECT CHECKLIST:

- Prior to construction:
  - ☑ Prototype diagram(s) and description of project scope.
  - ☑ Risk assessment/mitigation plan.
  - ☑ Timeline and action plan.
- During construction:
  - ☑ Safe work habits (e.g. no contact made with energized circuit at any time).
  - ☑ Correct equipment usage according to manufacturer's recommendations.
  - ☑ Timeline and action plan amended as necessary.
  - ☑ Maintain the originally-planned project scope (i.e. avoid adding features!).
- After completion:
  - ☑ All functions tested against original plan.
  - ☑ Full, accurate, and appropriate documentation of all project details.
  - ☑ Complete bill of materials.
  - ☑ Written summary of lessons learned.

Challenges

- ???.
- ???.
- ???.

Appendix A

Problem-Solving Strategies

The ability to solve complex problems is arguably one of the most valuable skills one can possess, and this skill is particularly important in any science-based discipline.

- Study principles, not procedures. Don't be satisfied with merely knowing how to compute solutions – learn *why* those solutions work.
- Identify what it is you need to solve, identify all relevant data, identify all units of measurement, identify any general principles or formulae linking the given information to the solution, and then identify any "missing pieces" to a solution. Annotate all diagrams with this data.
- Sketch a diagram to help visualize the problem. When building a real system, always devise a plan for that system and analyze its function *before* constructing it.
- Follow the units of measurement and meaning of every calculation. If you are ever performing mathematical calculations as part of a problem-solving procedure, and you find yourself unable to apply each and every intermediate result to some aspect of the problem, it means you don't understand what you are doing. Properly done, every mathematical result should have practical meaning for the problem, and not just be an abstract number. You should be able to identify the proper units of measurement for each and every calculated result, and show where that result fits into the problem.
- Perform "thought experiments" to explore the effects of different conditions for theoretical problems. When troubleshooting real systems, perform *diagnostic tests* rather than visually inspecting for faults, the best diagnostic test being the one giving you the most information about the nature and/or location of the fault with the fewest steps.
- Simplify the problem until the solution becomes obvious, and then use that obvious case as a model to follow in solving the more complex version of the problem.
- Check for exceptions to see if your solution is incorrect or incomplete. A good solution will work for *all* known conditions and criteria. A good example of this is the process of testing scientific hypotheses: the task of a scientist is not to find support for a new idea, but rather to *challenge* that new idea to see if it holds up under a battery of tests. The philosophical

principle of *reductio ad absurdum* (i.e. disproving a general idea by finding a specific case where it fails) is useful here.

- Work "backward" from a hypothetical solution to a new set of given conditions.
- Add quantities to problems that are qualitative in nature, because sometimes a little math helps illuminate the scenario.
- Sketch graphs illustrating how variables relate to each other. These may be quantitative (i.e. with realistic number values) or qualitative (i.e. simply showing increases and decreases).
- Treat quantitative problems as qualitative in order to discern the relative magnitudes and/or directions of change of the relevant variables. For example, try determining what happens if a certain variable were to increase or decrease before attempting to precisely calculate quantities: how will each of the dependent variables respond, by increasing, decreasing, or remaining the same as before?
- Consider limiting cases. This works especially well for qualitative problems where you need to determine which direction a variable will change. Take the given condition and magnify that condition to an extreme degree as a way of simplifying the direction of the system's response.
- Check your work. This means regularly testing your conclusions to see if they make sense. This does *not* mean repeating the same steps originally used to obtain the conclusion(s), but rather to use some other means to check validity. Simply repeating procedures often leads to *repeating the same errors* if any were made, which is why alternative paths are better.

Appendix B

Instructional philosophy

"The unexamined circuit is not worth energizing" – Socrates (if he had taught electricity)

These learning modules, although useful for self-study, were designed to be used in a formal learning environment where a subject-matter expert challenges students to digest the content and exercise their critical thinking abilities in the answering of questions and in the construction and testing of working circuits.

The following principles inform the instructional and assessment philosophies embodied in these learning modules:

- The first goal of education is to enhance clear and independent thought, in order that every student reach their fullest potential in a highly complex and inter-dependent world. Robust reasoning is *always* more important than particulars of any subject matter, because its application is universal.
- Literacy is fundamental to independent learning and thought because text continues to be the most efficient way to communicate complex ideas over space and time. Those who cannot read with ease are limited in their ability to acquire knowledge and perspective.
- Articulate communication is fundamental to work that is complex and interdisciplinary.
- Faulty assumptions and poor reasoning are best corrected through challenge, not presentation. The rhetorical technique of *reductio ad absurdum* (disproving an assertion by exposing an absurdity) works well to discipline student's minds, not only to correct the problem at hand but also to learn how to detect and correct future errors.
- Important principles should be repeatedly explored and widely applied throughout a course of study, not only to reinforce their importance and help ensure their mastery, but also to showcase the interconnectedness and utility of knowledge.

These learning modules were expressly designed to be used in an "inverted" teaching environment[1] where students first read the introductory and tutorial chapters on their own, then individually attempt to answer the questions and construct working circuits according to the experiment and project guidelines. The instructor never lectures, but instead meets regularly with each individual student to review their progress, answer questions, identify misconceptions, and challenge the student to new depths of understanding through further questioning. Regular meetings between instructor and student should resemble a Socratic[2] dialogue, where questions serve as scalpels to dissect topics and expose assumptions. The student passes each module only after consistently demonstrating their ability to logically analyze and correctly apply all major concepts in each question or project/experiment. The instructor must be vigilant in probing each student's understanding to ensure they are truly *reasoning* and not just *memorizing*. This is why "Challenge" points appear throughout, as prompts for students to think deeper about topics and as starting points for instructor queries. Sometimes these challenge points require additional knowledge that hasn't been covered in the series to answer in full. This is okay, as the major purpose of the Challenges is to stimulate analysis and synthesis on the part of each student.

The instructor must possess enough mastery of the subject matter and awareness of students' reasoning to generate their own follow-up questions to practically any student response. Even completely correct answers given by the student should be challenged by the instructor for the purpose of having students practice articulating their thoughts and defending their reasoning. Conceptual errors committed by the student should be exposed and corrected not by direct instruction, but rather by reducing the errors to an absurdity[3] through well-chosen questions and thought experiments posed by the instructor. Becoming proficient at this style of instruction requires time and dedication, but the positive effects on critical thinking for both student and instructor are spectacular.

An inspection of these learning modules reveals certain unique characteristics. One of these is a bias toward thorough explanations in the tutorial chapters. Without a live instructor to explain concepts and applications to students, the text itself must fulfill this role. This philosophy results in lengthier explanations than what you might typically find in a textbook, each step of the reasoning process fully explained, including footnotes addressing common questions and concerns students raise while learning these concepts. Each tutorial seeks to not only explain each major concept in sufficient detail, but also to explain the logic of each concept and how each may be developed

[1] In a traditional teaching environment, students first encounter new information via *lecture* from an expert, and then independently apply that information via *homework*. In an "inverted" course of study, students first encounter new information via *homework*, and then independently apply that information under the scrutiny of an expert. The expert's role in lecture is to simply *explain*, but the expert's role in an inverted session is to *challenge*, *critique*, and if necessary *explain* where gaps in understanding still exist.

[2] Socrates is a figure in ancient Greek philosophy famous for his unflinching style of questioning. Although he authored no texts, he appears as a character in Plato's many writings. The essence of Socratic philosophy is to leave no question unexamined and no point of view unchallenged. While purists may argue a topic such as electric circuits is too narrow for a true Socratic-style dialogue, I would argue that the essential thought processes involved with scientific reasoning on *any* topic are not far removed from the Socratic ideal, and that students of electricity and electronics would do very well to challenge assumptions, pose thought experiments, identify fallacies, and otherwise employ the arsenal of critical thinking skills modeled by Socrates.

[3] This rhetorical technique is known by the Latin phrase *reductio ad absurdum*. The concept is to expose errors by counter-example, since only one solid counter-example is necessary to disprove a universal claim. As an example of this, consider the common misconception among beginning students of electricity that voltage cannot exist without current. One way to apply *reductio ad absurdum* to this statement is to ask how much current passes through a fully-charged battery connected to nothing (i.e. a clear example of voltage existing without current).

from "first principles". Again, this reflects the goal of developing clear and independent thought in students' minds, by showing how clear and logical thought was used to forge each concept. Students benefit from witnessing a model of clear thinking in action, and these tutorials strive to be just that.

Another characteristic of these learning modules is a lack of step-by-step instructions in the Project and Experiment chapters. Unlike many modern workbooks and laboratory guides where step-by-step instructions are prescribed for each experiment, these modules take the approach that students must learn to closely read the tutorials and apply their own reasoning to identify the appropriate experimental steps. Sometimes these steps are plainly declared in the text, just not as a set of enumerated points. At other times certain steps are implied, an example being assumed competence in test equipment use where the student should not need to be told *again* how to use their multimeter because that was thoroughly explained in previous lessons. In some circumstances no steps are given at all, leaving the entire procedure up to the student.

This lack of prescription is not a flaw, but rather a feature. Close reading and clear thinking are foundational principles of this learning series, and in keeping with this philosophy all activities are designed to *require* those behaviors. Some students may find the lack of prescription frustrating, because it demands more from them than what their previous educational experiences required. This frustration should be interpreted as an unfamiliarity with autonomous thinking, a problem which must be corrected if the student is ever to become a self-directed learner and effective problem-solver. Ultimately, the need for students to read closely and think clearly is more important both in the near-term and far-term than any specific facet of the subject matter at hand. If a student takes longer than expected to complete a module because they are forced to outline, digest, and reason on their own, so be it. The future gains enjoyed by developing this mental discipline will be well worth the additional effort and delay.

Another feature of these learning modules is that they do not treat topics in isolation. Rather, important concepts are introduced early in the series, and appear repeatedly as stepping-stones toward other concepts in subsequent modules. This helps to avoid the "compartmentalization" of knowledge, demonstrating the inter-connectedness of concepts and simultaneously reinforcing them. Each module is fairly complete in itself, reserving the beginning of its tutorial to a review of foundational concepts.

This methodology of assigning text-based modules to students for digestion and then using Socratic dialogue to assess progress and hone students' thinking was developed over a period of several years by the author with his Electronics and Instrumentation students at the two-year college level. While decidedly unconventional and sometimes even unsettling for students accustomed to a more passive lecture environment, this instructional philosophy has proven its ability to convey conceptual mastery, foster careful analysis, and enhance employability so much better than lecture that the author refuses to ever teach by lecture again.

Problems which often go undiagnosed in a lecture environment are laid bare in this "inverted" format where students must articulate and logically defend their reasoning. This, too, may be unsettling for students accustomed to lecture sessions where the instructor cannot tell for sure who comprehends and who does not, and this vulnerability necessitates sensitivity on the part of the "inverted" session instructor in order that students never feel discouraged by having their errors exposed. *Everyone* makes mistakes from time to time, and learning is a lifelong process! Part of the instructor's job is to build a culture of learning among the students where errors are not seen as shameful, but rather as opportunities for progress.

To this end, instructors managing courses based on these modules should adhere to the following principles:

- Student questions are always welcome and demand thorough, honest answers. The only type of question an instructor should refuse to answer is one the student should be able to easily answer on their own. Remember, *the fundamental goal of education is for each student to learn to think clearly and independently.* This requires hard work on the part of the student, which no instructor should ever circumvent. Anything done to bypass the student's responsibility to do that hard work ultimately limits that student's potential and thereby does real harm.

- It is not only permissible, but encouraged, to answer a student's question by asking questions in return, these follow-up questions designed to guide the student to reach a correct answer through their own reasoning.

- All student answers demand to be challenged by the instructor and/or by other students. This includes both correct and incorrect answers – the goal is to practice the articulation and defense of one's own reasoning.

- No reading assignment is deemed complete unless and until the student demonstrates their ability to accurately summarize the major points in their own terms. Recitation of the original text is unacceptable. This is why every module contains an "Outline and reflections" question as well as a "Foundational concepts" question in the Conceptual reasoning section, to prompt reflective reading.

- No assigned question is deemed answered unless and until the student demonstrates their ability to consistently and correctly apply the concepts to *variations* of that question. This is why module questions typically contain multiple "Challenges" suggesting different applications of the concept(s) as well as variations on the same theme(s). Instructors are encouraged to devise as many of their own "Challenges" as they are able, in order to have a multitude of ways ready to probe students' understanding.

- No assigned experiment or project is deemed complete unless and until the student demonstrates the task in action. If this cannot be done "live" before the instructor, video-recordings showing the demonstration are acceptable. All relevant safety precautions must be followed, all test equipment must be used correctly, and the student must be able to properly explain all results. The student must also successfully answer all Challenges presented by the instructor for that experiment or project.

Students learning from these modules would do well to abide by the following principles:

- No text should be considered fully and adequately read unless and until you can express every idea *in your own words, using your own examples.*
- You should always articulate your thoughts as you read the text, noting points of agreement, confusion, and epiphanies. Feel free to print the text on paper and then write your notes in the margins. Alternatively, keep a journal for your own reflections as you read. This is truly a helpful tool when digesting complicated concepts.
- Never take the easy path of highlighting or underlining important text. Instead, *summarize* and/or *comment* on the text using your own words. This actively engages your mind, allowing you to more clearly perceive points of confusion or misunderstanding on your own.
- A very helpful strategy when learning new concepts is to place yourself in the role of a teacher, if only as a mental exercise. Either explain what you have recently learned to someone else, or at least *imagine* yourself explaining what you have learned to someone else. The simple act of having to articulate new knowledge and skill forces you to take on a different perspective, and will help reveal weaknesses in your understanding.
- Perform each and every mathematical calculation and thought experiment shown in the text on your own, referring back to the text to see that your results agree. This may seem trivial and unnecessary, but it is critically important to ensuring you actually understand what is presented, especially when the concepts at hand are complicated and easy to misunderstand. Apply this same strategy to become proficient in the use of *circuit simulation software*, checking to see if your simulated results agree with the results shown in the text.
- Above all, recognize that learning is hard work, and that a certain level of frustration is unavoidable. There are times when you will struggle to grasp some of these concepts, and that struggle is a natural thing. Take heart that it will yield with persistent and varied[4] effort, and never give up!

Students interested in using these modules for self-study will also find them beneficial, although the onus of responsibility for thoroughly reading and answering questions will of course lie with that individual alone. If a qualified instructor is not available to challenge students, a workable alternative is for students to form study groups where they challenge[5] one another.

To high standards of education,

Tony R. Kuphaldt

---

[4] As the old saying goes, "Insanity is trying the same thing over and over again, expecting different results." If you find yourself stumped by something in the text, you should attempt a different approach. Alter the thought experiment, change the mathematical parameters, do whatever you can to see the problem in a slightly different light, and then the solution will often present itself more readily.

[5] Avoid the temptation to simply share answers with study partners, as this is really counter-productive to learning. Always bear in mind that the answer to any question is far less important in the long run than the method(s) used to obtain that answer. The goal of education is to empower one's life through the improvement of clear and independent thought, literacy, expression, and various practical skills.

Appendix C

Tools used

I am indebted to the developers of many open-source software applications in the creation of these learning modules. The following is a list of these applications with some commentary on each.

You will notice a theme common to many of these applications: a bias toward *code*. Although I am by no means an expert programmer in any computer language, I understand and appreciate the flexibility offered by code-based applications where the user (you) enters commands into a plain ASCII text file, which the software then reads and processes to create the final output. Code-based computer applications are by their very nature *extensible*, while WYSIWYG (What You See Is What You Get) applications are generally limited to whatever user interface the developer makes for you.

The `GNU/Linux` computer operating system

> There is so much to be said about Linus Torvalds' `Linux` and Richard Stallman's `GNU` project. First, to credit just these two individuals is to fail to do justice to the *mob* of passionate volunteers who contributed to make this amazing software a reality. I first learned of `Linux` back in 1996, and have been using this operating system on my personal computers almost exclusively since then. It is *free*, it is completely *configurable*, and it permits the continued use of highly efficient `Unix` applications and scripting languages (e.g. shell scripts, Makefiles, `sed`, `awk`) developed over many decades. `Linux` not only provided me with a powerful computing platform, but its open design served to inspire my life's work of creating open-source educational resources.

Bram Moolenaar's `Vim` text editor

> Writing code for any code-based computer application requires a *text editor*, which may be thought of as a word processor strictly limited to outputting plain-ASCII text files. Many good text editors exist, and one's choice of text editor seems to be a deeply personal matter within the programming world. I prefer `Vim` because it operates very similarly to `vi` which is ubiquitous on `Unix`/`Linux` operating systems, and because it may be entirely operated via keyboard (i.e. no mouse required) which makes it fast to use.

Donald Knuth's TEX typesetting system

Developed in the late 1970's and early 1980's by computer scientist extraordinaire Donald Knuth to typeset his multi-volume magnum opus *The Art of Computer Programming*, this software allows the production of formatted text for screen-viewing or paper printing, all by writing plain-text code to describe how the formatted text is supposed to appear. TEX is not just a markup language for documents, but it is also a Turing-complete programming language in and of itself, allowing useful algorithms to be created to control the production of documents. Simply put, TEX *is a programmer's approach to word processing*. Since TEX is controlled by code written in a plain-text file, this means anyone may read that plain-text file to see exactly how the document was created. This openness afforded by the code-based nature of TEX makes it relatively easy to learn how other people have created their own TEX documents. By contrast, examining a beautiful document created in a conventional WYSIWYG word processor such as Microsoft `Word` suggests nothing to the reader about *how* that document was created, or what the user might do to create something similar. As Mr. Knuth himself once quipped, conventional word processing applications should be called WYSIAYG (What You See Is *All* You Get).

Leslie Lamport's LATEX extensions to TEX

Like all true programming languages, TEX is inherently extensible. So, years after the release of TEX to the public, Leslie Lamport decided to create a massive extension allowing easier compilation of book-length documents. The result was LATEX, which is the markup language used to create all ModEL module documents. You could say that TEX is to LATEX as `C` is to `C++`. This means it is permissible to use any and all TEX commands within LATEX source code, and it all still works. Some of the features offered by LATEX that would be challenging to implement in TEX include automatic index and table-of-content creation.

Tim Edwards' `Xcircuit` drafting program

This wonderful program is what I use to create all the schematic diagrams and illustrations (but not photographic images or mathematical plots) throughout the ModEL project. It natively outputs PostScript format which is a true vector graphic format (this is why the images do not pixellate when you zoom in for a closer view), and it is so simple to use that I have never had to read the manual! Object libraries are easy to create for `Xcircuit`, being plain-text files using PostScript programming conventions. Over the years I have collected a large set of object libraries useful for drawing electrical and electronic schematics, pictorial diagrams, and other technical illustrations.

`Gimp` graphic image manipulation program

Essentially an open-source clone of Adobe's `PhotoShop`, I use `Gimp` to resize, crop, and convert file formats for all of the photographic images appearing in the ModEL modules. Although `Gimp` does offer its own scripting language (called `Script-Fu`), I have never had occasion to use it. Thus, my utilization of `Gimp` to merely crop, resize, and convert graphic images is akin to using a sword to slice bread.

`SPICE` circuit simulation program

`SPICE` is to circuit analysis as TEX is to document creation: it is a form of markup language designed to describe a certain object to be processed in plain-ASCII text. When the plain-text "source file" is compiled by the software, it outputs the final result. More modern circuit analysis tools certainly exist, but I prefer `SPICE` for the following reasons: it is *free*, it is *fast*, it is *reliable*, and it is a fantastic tool for *teaching* students of electricity and electronics how to write simple code. I happen to use rather old versions of `SPICE`, version 2g6 being my "go to" application when I only require text-based output. `NGSPICE` (version 26), which is based on Berkeley `SPICE` version 3f5, is used when I require graphical output for such things as time-domain waveforms and Bode plots. In all `SPICE` example netlists I strive to use coding conventions compatible with all `SPICE` versions.

Andrew D. Hwang's `ePiX` mathematical visualization programming library

This amazing project is a `C++` library you may link to any `C/C++` code for the purpose of generating PostScript graphic images of mathematical functions. As a completely free and open-source project, it does all the plotting I would otherwise use a Computer Algebra System (CAS) such as `Mathematica` or `Maple` to do. It should be said that `ePiX` is *not* a Computer Algebra System like `Mathematica` or `Maple`, but merely a mathematical *visualization* tool. In other words, it won't determine integrals for you (you'll have to implement that in your own `C/C++` code!), but it can graph the results, and it does so beautifully. What I really admire about `ePiX` is that it is a `C++` programming library, which means it builds on the existing power and toolset available with that programming language. Mr. Hwang could have probably developed his own stand-alone application for mathematical plotting, but by creating a `C++` library to do the same thing he accomplished something much greater.

`gnuplot` mathematical visualization software

Another open-source tool for mathematical visualization is `gnuplot`. Interestingly, this tool is *not* part of Richard Stallman's GNU project, its name being a coincidence. For this reason the authors prefer "gnu" *not* be capitalized at all to avoid confusion. This is a much "lighter-weight" alternative to a spreadsheet for plotting tabular data, and the fact that it easily outputs directly to an X11 console or a file in a number of different graphical formats (including PostScript) is very helpful. I typically set my `gnuplot` output format to default (X11 on my Linux PC) for quick viewing while I'm developing a visualization, then switch to PostScript file export once the visual is ready to include in the document(s) I'm writing. As with my use of `Gimp` to do rudimentary image editing, my use of `gnuplot` only scratches the surface of its capabilities, but the important points are that it's *free* and that it *works well*.

`Python` programming language

Both Python and C++ find extensive use in these modules as instructional aids and exercises, but I'm listing Python here as a *tool* for myself because I use it almost daily as a *calculator*. If you open a Python interpreter console and type `from math import *` you can type mathematical expressions and have it return results just as you would on a hand calculator. Complex-number (i.e. *phasor*) arithmetic is similarly supported if you include the complex-math library (`from cmath import *`). Examples of this are shown in the Programming References chapter (if included) in each module. Of course, being a fully-featured programming language, Python also supports conditionals, loops, and other structures useful for calculation of quantities. Also, running in a console environment where all entries and returned values show as text in a chronologically-ordered list makes it easy to copy-and-paste those calculations to document exactly how they were performed.

Appendix D

Creative Commons License

Creative Commons Attribution 4.0 International Public License

By exercising the Licensed Rights (defined below), You accept and agree to be bound by the terms and conditions of this Creative Commons Attribution 4.0 International Public License ("Public License"). To the extent this Public License may be interpreted as a contract, You are granted the Licensed Rights in consideration of Your acceptance of these terms and conditions, and the Licensor grants You such rights in consideration of benefits the Licensor receives from making the Licensed Material available under these terms and conditions.

Section 1 – Definitions.

a. **Adapted Material** means material subject to Copyright and Similar Rights that is derived from or based upon the Licensed Material and in which the Licensed Material is translated, altered, arranged, transformed, or otherwise modified in a manner requiring permission under the Copyright and Similar Rights held by the Licensor. For purposes of this Public License, where the Licensed Material is a musical work, performance, or sound recording, Adapted Material is always produced where the Licensed Material is synched in timed relation with a moving image.

b. **Adapter's License** means the license You apply to Your Copyright and Similar Rights in Your contributions to Adapted Material in accordance with the terms and conditions of this Public License.

c. **Copyright and Similar Rights** means copyright and/or similar rights closely related to copyright including, without limitation, performance, broadcast, sound recording, and Sui Generis Database Rights, without regard to how the rights are labeled or categorized. For purposes of this Public License, the rights specified in Section 2(b)(1)-(2) are not Copyright and Similar Rights.

d. **Effective Technological Measures** means those measures that, in the absence of proper authority, may not be circumvented under laws fulfilling obligations under Article 11 of the WIPO Copyright Treaty adopted on December 20, 1996, and/or similar international agreements.

e. **Exceptions and Limitations** means fair use, fair dealing, and/or any other exception or

limitation to Copyright and Similar Rights that applies to Your use of the Licensed Material.

f. **Licensed Material** means the artistic or literary work, database, or other material to which the Licensor applied this Public License.

g. **Licensed Rights** means the rights granted to You subject to the terms and conditions of this Public License, which are limited to all Copyright and Similar Rights that apply to Your use of the Licensed Material and that the Licensor has authority to license.

h. **Licensor** means the individual(s) or entity(ies) granting rights under this Public License.

i. **Share** means to provide material to the public by any means or process that requires permission under the Licensed Rights, such as reproduction, public display, public performance, distribution, dissemination, communication, or importation, and to make material available to the public including in ways that members of the public may access the material from a place and at a time individually chosen by them.

j. **Sui Generis Database Rights** means rights other than copyright resulting from Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, as amended and/or succeeded, as well as other essentially equivalent rights anywhere in the world.

k. **You** means the individual or entity exercising the Licensed Rights under this Public License. **Your** has a corresponding meaning.

Section 2 – Scope.

a. License grant.

1. Subject to the terms and conditions of this Public License, the Licensor hereby grants You a worldwide, royalty-free, non-sublicensable, non-exclusive, irrevocable license to exercise the Licensed Rights in the Licensed Material to:

A. reproduce and Share the Licensed Material, in whole or in part; and

B. produce, reproduce, and Share Adapted Material.

2. Exceptions and Limitations. For the avoidance of doubt, where Exceptions and Limitations apply to Your use, this Public License does not apply, and You do not need to comply with its terms and conditions.

3. Term. The term of this Public License is specified in Section 6(a).

4. Media and formats; technical modifications allowed. The Licensor authorizes You to exercise the Licensed Rights in all media and formats whether now known or hereafter created, and to make technical modifications necessary to do so. The Licensor waives and/or agrees not to assert any right or authority to forbid You from making technical modifications necessary to exercise the Licensed Rights, including technical modifications necessary to circumvent Effective Technological Measures.

For purposes of this Public License, simply making modifications authorized by this Section 2(a)(4) never produces Adapted Material.

5. Downstream recipients.

A. Offer from the Licensor – Licensed Material. Every recipient of the Licensed Material automatically receives an offer from the Licensor to exercise the Licensed Rights under the terms and conditions of this Public License.

B. No downstream restrictions. You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, the Licensed Material if doing so restricts exercise of the Licensed Rights by any recipient of the Licensed Material.

6. No endorsement. Nothing in this Public License constitutes or may be construed as permission to assert or imply that You are, or that Your use of the Licensed Material is, connected with, or sponsored, endorsed, or granted official status by, the Licensor or others designated to receive attribution as provided in Section 3(a)(1)(A)(i).

b. Other rights.

1. Moral rights, such as the right of integrity, are not licensed under this Public License, nor are publicity, privacy, and/or other similar personality rights; however, to the extent possible, the Licensor waives and/or agrees not to assert any such rights held by the Licensor to the limited extent necessary to allow You to exercise the Licensed Rights, but not otherwise.

2. Patent and trademark rights are not licensed under this Public License.

3. To the extent possible, the Licensor waives any right to collect royalties from You for the exercise of the Licensed Rights, whether directly or through a collecting society under any voluntary or waivable statutory or compulsory licensing scheme. In all other cases the Licensor expressly reserves any right to collect such royalties.

Section 3 – License Conditions.

Your exercise of the Licensed Rights is expressly made subject to the following conditions.

a. Attribution.

1. If You Share the Licensed Material (including in modified form), You must:

A. retain the following if it is supplied by the Licensor with the Licensed Material:

i. identification of the creator(s) of the Licensed Material and any others designated to receive attribution, in any reasonable manner requested by the Licensor (including by pseudonym if designated);

ii. a copyright notice;

iii. a notice that refers to this Public License;

iv. a notice that refers to the disclaimer of warranties;

v. a URI or hyperlink to the Licensed Material to the extent reasonably practicable;

B. indicate if You modified the Licensed Material and retain an indication of any previous modifications; and

C. indicate the Licensed Material is licensed under this Public License, and include the text of, or the URI or hyperlink to, this Public License.

2. You may satisfy the conditions in Section 3(a)(1) in any reasonable manner based on the medium, means, and context in which You Share the Licensed Material. For example, it may be reasonable to satisfy the conditions by providing a URI or hyperlink to a resource that includes the required information.

3. If requested by the Licensor, You must remove any of the information required by Section 3(a)(1)(A) to the extent reasonably practicable.

4. If You Share Adapted Material You produce, the Adapter's License You apply must not prevent recipients of the Adapted Material from complying with this Public License.

Section 4 – Sui Generis Database Rights.

Where the Licensed Rights include Sui Generis Database Rights that apply to Your use of the Licensed Material:

a. for the avoidance of doubt, Section 2(a)(1) grants You the right to extract, reuse, reproduce, and Share all or a substantial portion of the contents of the database;

b. if You include all or a substantial portion of the database contents in a database in which You have Sui Generis Database Rights, then the database in which You have Sui Generis Database Rights (but not its individual contents) is Adapted Material; and

c. You must comply with the conditions in Section 3(a) if You Share all or a substantial portion of the contents of the database.

For the avoidance of doubt, this Section 4 supplements and does not replace Your obligations under this Public License where the Licensed Rights include other Copyright and Similar Rights.

Section 5 – Disclaimer of Warranties and Limitation of Liability.

a. Unless otherwise separately undertaken by the Licensor, to the extent possible, the Licensor offers the Licensed Material as-is and as-available, and makes no representations or warranties of any kind concerning the Licensed Material, whether express, implied, statutory, or other. This includes, without limitation, warranties of title, merchantability, fitness for a particular purpose, non-infringement, absence of latent or other defects, accuracy, or the presence or absence of errors,

whether or not known or discoverable. Where disclaimers of warranties are not allowed in full or in part, this disclaimer may not apply to You.

b. To the extent possible, in no event will the Licensor be liable to You on any legal theory (including, without limitation, negligence) or otherwise for any direct, special, indirect, incidental, consequential, punitive, exemplary, or other losses, costs, expenses, or damages arising out of this Public License or use of the Licensed Material, even if the Licensor has been advised of the possibility of such losses, costs, expenses, or damages. Where a limitation of liability is not allowed in full or in part, this limitation may not apply to You.

c. The disclaimer of warranties and limitation of liability provided above shall be interpreted in a manner that, to the extent possible, most closely approximates an absolute disclaimer and waiver of all liability.

Section 6 – Term and Termination.

a. This Public License applies for the term of the Copyright and Similar Rights licensed here. However, if You fail to comply with this Public License, then Your rights under this Public License terminate automatically.

b. Where Your right to use the Licensed Material has terminated under Section 6(a), it reinstates:

1. automatically as of the date the violation is cured, provided it is cured within 30 days of Your discovery of the violation; or

2. upon express reinstatement by the Licensor.

For the avoidance of doubt, this Section 6(b) does not affect any right the Licensor may have to seek remedies for Your violations of this Public License.

c. For the avoidance of doubt, the Licensor may also offer the Licensed Material under separate terms or conditions or stop distributing the Licensed Material at any time; however, doing so will not terminate this Public License.

d. Sections 1, 5, 6, 7, and 8 survive termination of this Public License.

Section 7 – Other Terms and Conditions.

a. The Licensor shall not be bound by any additional or different terms or conditions communicated by You unless expressly agreed.

b. Any arrangements, understandings, or agreements regarding the Licensed Material not stated herein are separate from and independent of the terms and conditions of this Public License.

Section 8 – Interpretation.

a. For the avoidance of doubt, this Public License does not, and shall not be interpreted to, reduce, limit, restrict, or impose conditions on any use of the Licensed Material that could lawfully

be made without permission under this Public License.

b. To the extent possible, if any provision of this Public License is deemed unenforceable, it shall be automatically reformed to the minimum extent necessary to make it enforceable. If the provision cannot be reformed, it shall be severed from this Public License without affecting the enforceability of the remaining terms and conditions.

c. No term or condition of this Public License will be waived and no failure to comply consented to unless expressly agreed to by the Licensor.

d. Nothing in this Public License constitutes or may be interpreted as a limitation upon, or waiver of, any privileges and immunities that apply to the Licensor or You, including from the legal processes of any jurisdiction or authority.

Creative Commons is not a party to its public licenses. Notwithstanding, Creative Commons may elect to apply one of its public licenses to material it publishes and in those instances will be considered the "Licensor." Except for the limited purpose of indicating that material is shared under a Creative Commons public license or as otherwise permitted by the Creative Commons policies published at `creativecommons.org/policies`, Creative Commons does not authorize the use of the trademark "Creative Commons" or any other trademark or logo of Creative Commons without its prior written consent including, without limitation, in connection with any unauthorized modifications to any of its public licenses or any other arrangements, understandings, or agreements concerning use of licensed material. For the avoidance of doubt, this paragraph does not form part of the public licenses.

Creative Commons may be contacted at `creativecommons.org`.

Appendix E

References

Appendix F

Version history

This is a list showing all significant additions, corrections, and other edits made to this learning module. Each entry is referenced by calendar date in reverse chronological order (newest version first), which appears on the front cover of every learning module for easy reference. Any contributors to this open-source document are listed here as well.

9 November 2024 – added an Introduction section on challenging concepts.

4 August 2024 – corrected an error in a pulse waveform graph showing UART signals with parity bits (image_7306).

30-31 July 2024 – minor edit made to the Tutorial section on counter functions, and also added a footnote on Schmitt-trigger logic gates to the Tutorial section on off-delay timer functions. Also fixed a typographical error in the microcontroller pseudocode examples for off-delay and on-delay timing functions.

29 July 2024 – edited the name of that new Tutorial section to be "Serial data communication" rather than "Serial communication".

24 July 2024 – added more assessment methods to the Introduction chapter.

19-20 July 2024 – added new Tutorial section on Serial data communication.

16 July 2024 – added new Tutorial section on Counters, and also added sections to the Introduction chapter (one for students, one for instructors).

27-28 May 2024 – added new Case Tutorial sections showing sample Arduino Nano microcontroller programs.

13 August 2022 – added a section on latching functionality.

3 August 2022 – minor typographical error correction in the Case Tutorial ("essense" should have been spelled "essence"), as well as a couple of other mis-spellings in the Tutorial.

27-31 May 2022 – document first created.

Index