Teach-In 2016

Exploring the Arduino

Part 2: Relay and switch control

by Mike and Richard Tooley

Welcome to *Teach-In 2016 – Exploring the Arduino*. This exciting new series has been designed for electronics enthusiasts wanting to get to grips with the immensely popular Arduino microcontroller as well as coding enthusiasts who want to explore hardware and interfacing. So, whether you are considering what to do with your Arduino, or maybe have an idea for a project but don't know how to turn it into reality, our new *Teach-In 2016* series will provide you with a one-stop source of ideas and practical information.

In last month's *Teach-In 2016* we took a first look at the Arduino, explaining why this particular microcontroller has become so popular. *Arduino Workshop* dealt with installing and running the Arduino's simple but powerful integrated development environment (IDE) while *Coding Quickstart* introduced the different types of data that you will encounter in an Arduino environment. *Get Real* examined interfacing switches and LEDs to the Arduino's digital I/O ports and provided you with some simple example 'sketches' to get you started with C coding. For good measure, we introduced UnoArduSim, an easy-to-use, but very useful Arduino simulator that will allow you to develop and test your code without needing to have a real Arduino to hand.

This month

In this month's *Teach-In 2016* we will look at methods of connecting real world hardware to the Arduino Uno. To this end, *Arduino Workshop* deals with driving external loads while *Arduino World* looks at some handy low-cost interface boards that will allow you to drive high-current and high-voltage loads, such as mains operated lamps and motors. *Coding Quickstart*, our regular programming feature, explains the structure and layout of program code as well as introducing decisions and how to make them. Finally, *Get Real* will show you how to design and construct a simple but effective Arduino-based security system.

Arduino Workshop: Driving external loads

Last month, we mentioned that the ATmega328 processor provides a total of 23 input/output (I/O) lines. These lines are made available at one or more of the Uno's I/O connectors (see Fig.2.1). We explained that the I/O pins can be given different functions depending on the software configuration. The I/O port lines can be individually configured as inputs or outputs using the `pinMode()` function to configure the port direction (ie, input or output) and `digitalWrite()` to turn the respective line on or off. Note that the action is latching, so that, once an output is turned on or off it will remain in that state until a further command is generated to change the state.

Relay outputs

The Arduino's I/O pins can source or sink a maximum current of 40mA at standard 5V logic levels. This is sufficient for a lot of purposes (including illuminating an LED) but not enough to operate actuators, motors, lamps and many other 'real-world' output devices. Fortunately, the problem of driving the vast majority of high voltage and high current loads can be easily solved

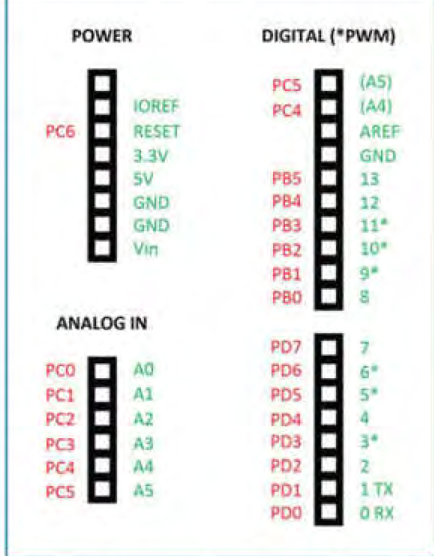


Fig.2.1. The Arduino Uno's I/O pin assignment

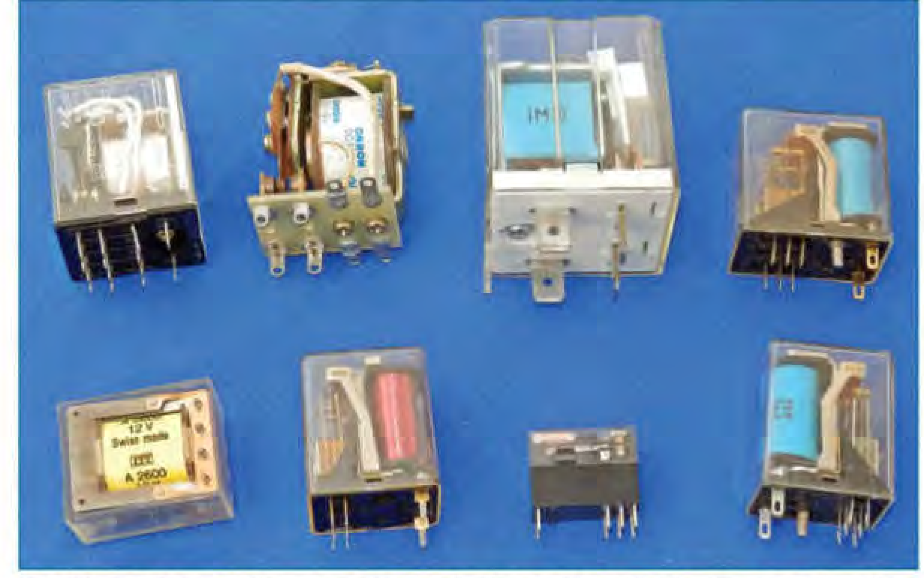


Fig.2.2. A selection of common types of relays with contact ratings ranging from 1A to 20A

Parameter	Value
Nominal operating voltage	5V DC
Nominal operating current	73mA
Maximum load rating	AC 250V 10A, DC 30V 10A
Pull-in voltage (typical)	3.8V
DC coil resistance	70Ω
Power consumption (typical)	0.36W
Operating time (max.)	10ms
Release time (max.)	5ms
Contact resistance (max.)	0.11Ω
Operating life	100,000 operations at rated load
Maximum switching rate	30 operations per second

Table 2.1 Electrical specifications of a typical miniature PCB-mounting relay

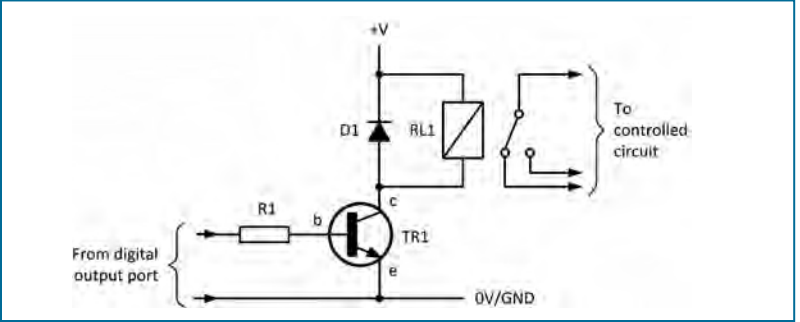


Fig.2.4. Simple single-transistor relay interface

using one or more miniature relays (see Fig.2.2). These electromechanical devices comprise a coil wound on a high-permeability core and a moving armature mechanically linked to a set of contacts that make and break when the device is actuated (see Fig.2.3). When sufficient current is applied to the coil of the relay the resulting magnetic field will cause the soft iron armature to pull-in and this in turn will open or close the relay's electrical contacts. A typical miniature PCB-mounted relay will operate from a 5V DC supply and its contacts will pull-in at typically 75% of this value. The specifications of such a relay are listed in Table 2.1.

It is important to note from Table 2.2 that the relay coil requires an operating current that's well beyond the output drive capability of the Arduino. We therefore need an interface that will provide the extra current required. Fortunately, this can often be little more than a low-power transistor and a handful of other components, as shown in Fig.2.4.

In Fig.2.4 the transistor can be almost any NPN type with a current gain of around 100, or more. Diode D1 counters the effects of the induced voltage that will appear across the relay coil as the current (and consequently the magnetic flux) collapses when the transistor reverts from a conducting to a non-conducting state. A typical value for R1 would be 2.2kΩ when using a relay coil that requires less than 200mA to operate (eg, a 700Ω coil rated at 12V). This value for R1 is sufficiently small to ensure that TR1 is driven into saturation when a high-state output voltage appears on the digital I/O line, but large enough to reduce the demand on the I/O port to around 2mA.

A neater alternative to using discrete components is the use of an integrated circuit output driver, such as the popular ULN2803. We will be looking at this chip in a future *Teach-In 2016* article, but for the moment, if you only have a couple of high current/high voltage loads to drive then a simple discrete circuit like the one shown in Fig.2.4 is all that you need.

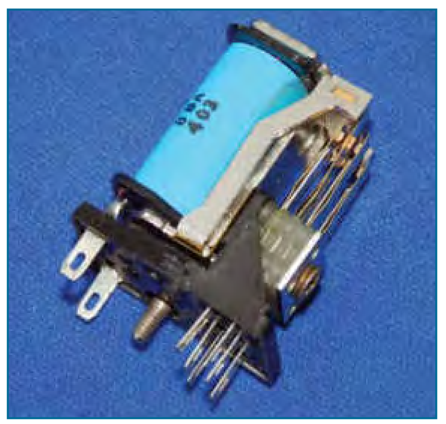


Fig.2.3. Internal arrangement of a typical relay showing the coil, armature and contacts

Fig.2.5. A four-channel relay interface board

Arduino World: Relay boards

Many simple control projects can be based on a ready-made relay board, avoiding the need to construct your own interface circuit. Fortunately, there are quite a few to choose from and the two most common types are fitted with either four or eight relays, with each relay having its own driver circuit.

Four-channel relay board

Fig.2.5 shows a typical four-channel relay board. The board has a transistor driver and an opto-isolator for each output channel. Similar boards can be purchased very cheaply (often less than £5) and so it is invariably more cost effective to purchase one of these boards rather than attempt to build one yourself.

Individual relays are normally fitted with single-pole changeover contacts (equivalent to an SPDT switch) and are commonly rated 250V AC at 10A or 30V DC at 10A. Inputs are usually TTL compatible and active *low* (in other words, they require a logic 0 output from the Arduino to operate).

Isolation

Many relay boards (like the one shown in Figs. 2.6) incorporate opto-isolators and this makes it possible to isolate the relay driver circuitry from the Arduino's circuitry (see Fig.2.7). However, in many applications this feature will not be required and the relay driver circuitry can then be operated from the same supply and ground connection as used by the Arduino itself.

Relay board connection

In Fig.2.8(a) the +5V and GND connections are common between the Arduino and the relay board. In this arrangement the relays must be 5V types and the only isolation between the Arduino and the load will be that afforded by the relay alone. This will usually be perfectly adequate for most applications, including switching mains loads at currents of up to several amps.

Fig.2.8(b) shows how higher-voltage (eg, 12V or 24V) relays can be used, while retaining a common ground (GND)

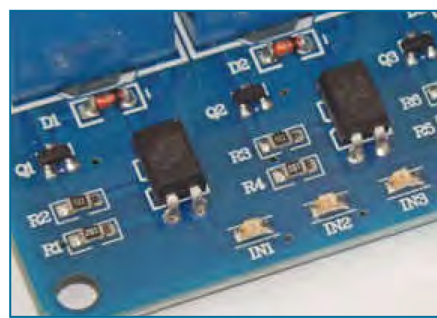


Fig.2.6. Transistor drivers and optical isolators on the four-channel relay board

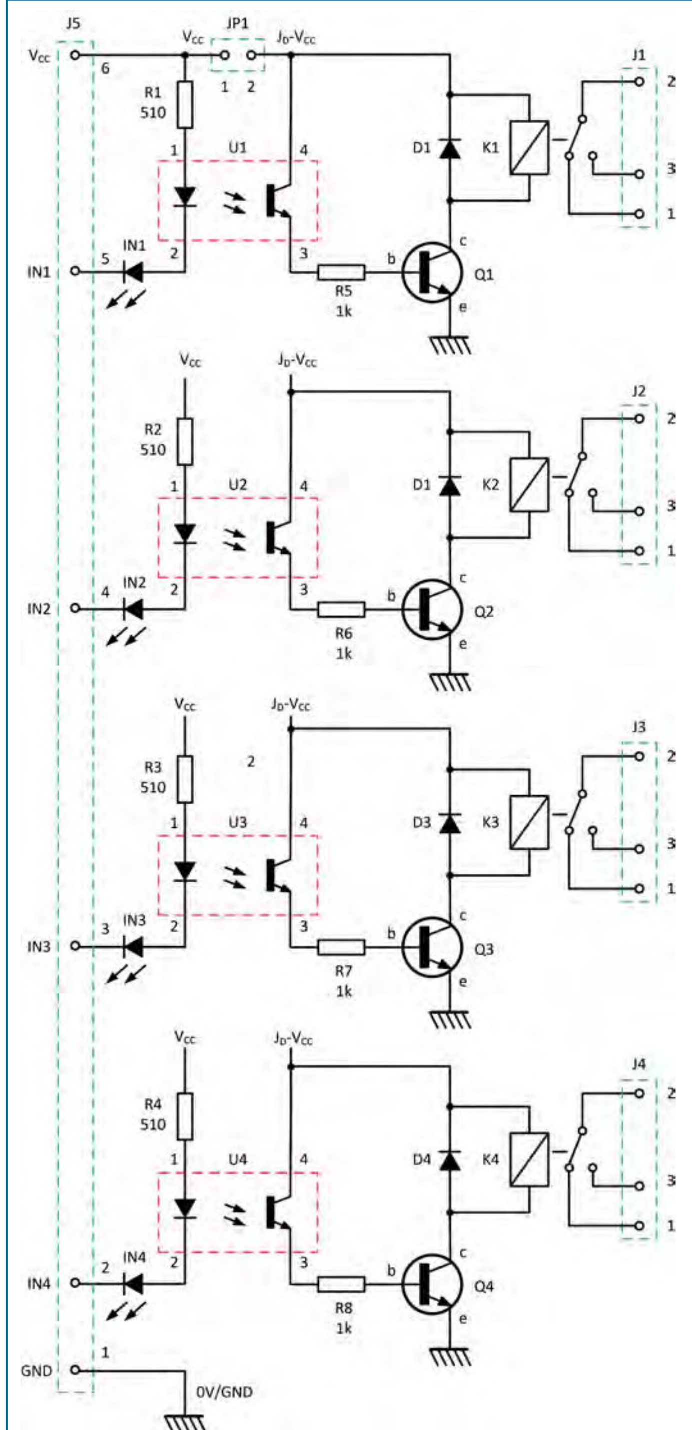


Fig.2.7. Complete circuit of a four-channel relay board

connection between the Arduino and the relay board. Higher power relays can be used for applications that involve switching currents of up to 20A. In all cases it is important to check the specifications of the relay that you plan to use and verify its suitability for use in a particular application.

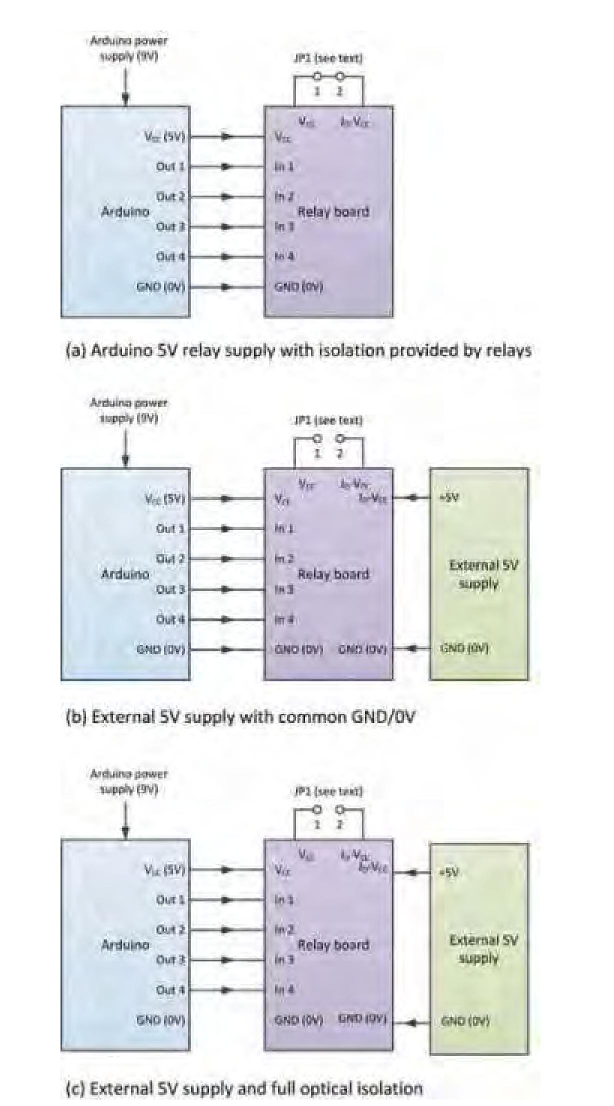


Fig.2.8. Three possible relay board configurations providing different amounts of isolation

In Fig.2.8(c) we have shown a fully optically isolated arrangement in which there is no common ground connection between the Arduino and the relay board. This arrangement offers the greatest amount of isolation, together with improved noise immunity.

Eight-channel relay board

Fig.2.9 shows an eight-channel relay board. Like the four-channel board that we've just described, boards of this type are also available at low-cost from several sources. At under £10 they offer an extremely cost-effective means of controlling up to eight loads and at a cost that's considerably less than the cost of purchasing the individual components.

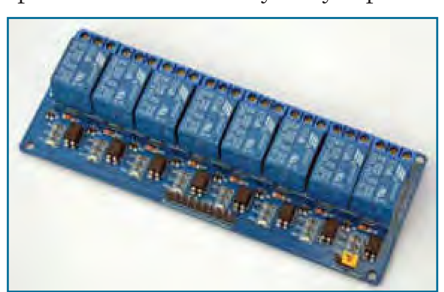


Fig.2.9. An eight-channel relay interface board

Coding relay outputs

Fortunately, it's very easy to control one or more relays using just a few lines of simple code. First, you will need to make sure that you define the digital output pins to which the relays are connected using a line of the form:

```
int pump = 5;  // Pump connected via a relay on digital pin-5
int heater = 6;  // Heater connected via a relay on digital pin-6
```

Next, you will need to add a couple of lines into the `setup()` code block, as follows:

```
pinMode(pump, OUTPUT);  // Pump is configured as an output
pinMode(heater, OUTPUT);  // Heater is configured as an output
```

The relays and their respective loads can be turned on and off incorporating the following lines of code at appropriate points in the main program loop:

```
digitalWrite(pump, LOW);  // Turn the pump off

digitalWrite(pump, HIGH);  // Turn the pump on

digitalWrite(heater, LOW);  // Turn the heater off

digitalWrite(heater, HIGH);  // Turn the pump on
```

Listing 1 shows a complete example where a fluid is heated and pumped in a continuous 30s cycle. Note that the main loop repeats indefinitely and can only be interrupted by using the Arduino's reset button.

Coding Quickstart : Understanding data types

Making decisions based on what's happening and then acting on this information in different ways is an essential pre-requisite of any programming language. C provides you with a variety of different conditional constructs that allow you to do this. Simple decisions can be made using nothing more than `if` and `else`, and loops can be controlled using `while`, `do while`, `for` and `loop`. We will look at all of these starting with `if` and `else`.

The `if` construct

The `if` construct is the most simple of all the conditional constructs. It is used when a statement (or a series of statements) should be executed when a particular condition prevails. The basic syntax is as follows:

```
if (conditional expression)
   // code to be executed if true,
   // each statement ending with ;
```

An example might be counting items into batches of 10 on a conveyor. Let's assume that we need to operate an LED when the count reaches (or exceeds) ten items. The following fragment of code would do this:

```
if (count >=10)
   digitalWrite(fullLED, HIGH);
```

If the value of count is less than 10 the condition evaluates false and the statement following the condition is then simply ignored. However, if the value of count is 10 or greater then the condition evaluates true and the statement following the condition is executed. In some applications it can be appropriate to use a series of `if` statements to detect various conditions and to act on them accordingly.

The `if ... else` construct

The `if ... else` construct is straightforward; its syntax is:

```
if (conditional expression)
   // code to be executed if true,
   // each statement ending with ;
else
   // code to be executed if false
   // each statement ending with ;
```

Here's a simple example. Let's assume that we are monitoring an analogue voltage and wish to set a threshold of 512 as the value at which a green LED should become illuminated and, below this value, we would like a red LED to be turned on. Our `if … else` construct would then look something like this:

```
if (inVoltage >=128)
   digitalWrite(greenLED, HIGH);
else
   digitalWrite(redLED, HIGH);
```

Unfortunately, this isn't quite the whole story. The red and green status LEDs should be mutually exclusive and so we might need to ensure that, when one LED is turned on the other LED is turned off. There are various ways that we could do this. We could either set them both off before we arrive at the `if else` construct or we could turn one on and the other off within a construct containing more than one statement (ie, a *compound* construct). Using the first method we might have:

```
// start with both LEDs off
digitalWrite(redLED, LOW);
digitalWrite(greenLED, LOW);
// now decide which LED to put on
if (inVoltage >=512)
   digitalWrite(greenLED, HIGH);
else
   digitalWrite(redLED, HIGH);
```

The other possibility is:

```
// Read the input voltage and
// put the red or green LED on
if (inVoltage >=512){
   digitalWrite(greenLED, HIGH);
   digitalWrite(redLED, LOW);
}
else{
   digitalWrite(redLED, HIGH);
   digitalWrite(greenLED, LOW);
}
```

Notice how in this example we've compounded several statements after the `if` and `else` and that that we've introduced curly braces, { and }, to make the logic clear and unambiguous. Of course, at some point earlier in the code we would have to define the pins that we are using to control the two LEDs and initialise the variables (`inVoltage`, `redLED` and `greenLED`).

Listing 2.2 provides you with a complete example that you can run using the excellent Arduino Uno simulator that we introduced last month. Fig.2.10 shows the code running in the simulator. Notice how we've entered the LED pin numbers as well as the pin used for the analogue voltage input in the appropriate simulator boxes. You can vary the input voltage by using the slider in the bottom right of the screen, noting how the status LEDs respond, changing when the input voltage reaches the 512 threshold. Finally, it's worth noting how UnoArduSim reports the current values of all of the variables in the bottom left-hand window.

Listing 2.1: Example of typical relay board code

```
/* Hot fluid cycle: heat for 24s and then pump for 6s */

int pump = 5;  // Pump connected via a relay on digital pin-5
int heater = 6;  // Heater connected via a relay on digital pin-6

void setup() {
   pinMode(pump, OUTPUT);  // Pump is configured as an output
   pinMode(heater, OUTPUT);  // Heater is configured as an output
}

void loop() {
   digitalWrite(pump, LOW);  // Turn the pump off
   digitalWrite(heater, HIGH);  // Turn the heater on
   delay(24000);  // Wait 24s
   digitalWrite(heater, LOW);  // Turn the heater off
   digitalWrite(pump, HIGH);  // Turn the pump on
   delay(6000);  // Wait 6s
}
```

Listing 2.2: Using a compound `if ... then` *construct*

```
/* Simple decision making using a compound if..then
   construct
*/

int redLED = 13;      // red LED on digital pin 13
int greenLED = 12;    // green LED on digital pin 12
int inAnalogue = A0;  // analogue input pin 0
int inVoltage = 0;    // initialise the variable

void setup()
{
      pinMode(redLED, OUTPUT);
      pinMode(greenLED, OUTPUT);
}

void loop()
{
      // get the input voltage
      inVoltage = analogRead(inAnalogue);
      // illuminate the green or red status LEDs
      if (inVoltage >= 512){
            digitalWrite(redLED, HIGH);
            digitalWrite(greenLED, LOW);
      }
      else{
            digitalWrite(greenLED, HIGH);
            digitalWrite(redLED, LOW);
      }
}
```

Conditions

In the last example you should have noticed the `>=` condition that we used to find out whether the input voltage has exceeded the threshold value of 512. The 'greater than or equal to' condition isn't the only one that we have to play with, as Table 2.2 shows.

Code	Meaning	Notes
a == b	a is equal to b	True if a and b have the same value
a != b	a is not equal to b	True if a and b have different values
a > b	a is greater than b	True if a is larger than b (but not true if they have the same value)
a < b	a is less than b	True if a is smaller than b (but not true if they have the same value)
a >= b	a is greater than or equal to b	True if a is larger than b (and also true if they have the same value)
a <= b	a is less than or equal to b	True if a is smaller than b (and also true if they have the same value)

Table 2.2 Conditions

The `while` construct

The `while` construct provides you with a means of continuously executing one or more statements until a condition evaluates false. The loop containing the statement (or statements) will continue to be executed as long as the condition remains true – but, as soon as it becomes false the loop will terminate and execution will continue with the next subsequent statement. The basic syntax is:

```
while (conditional expression){
   // statements to be executed if true,
   // each ending with ;
}
```

Here's an example that shows how a belt motor could be controlled using a while loop. The belt motor will run for as long as it takes for an item placed on the belt to reach a limit switch. Note that we must check the status of the limit switch inside the loop. If we forget to do this the motor will run forever!

```
while (limitSwitchStatus == LOW){
    // Limit not reached so run the motor
    digitalWrite(motorRun, HIGH);
// Check to see if anything has changed?
limitSwitchStatus = digitalRead(limitSwitch);
}
```

By making the conditional expression dependent on the value of a counter modified inside the loop we have a simple means of performing one or more statements a predetermined number of times, as follows:

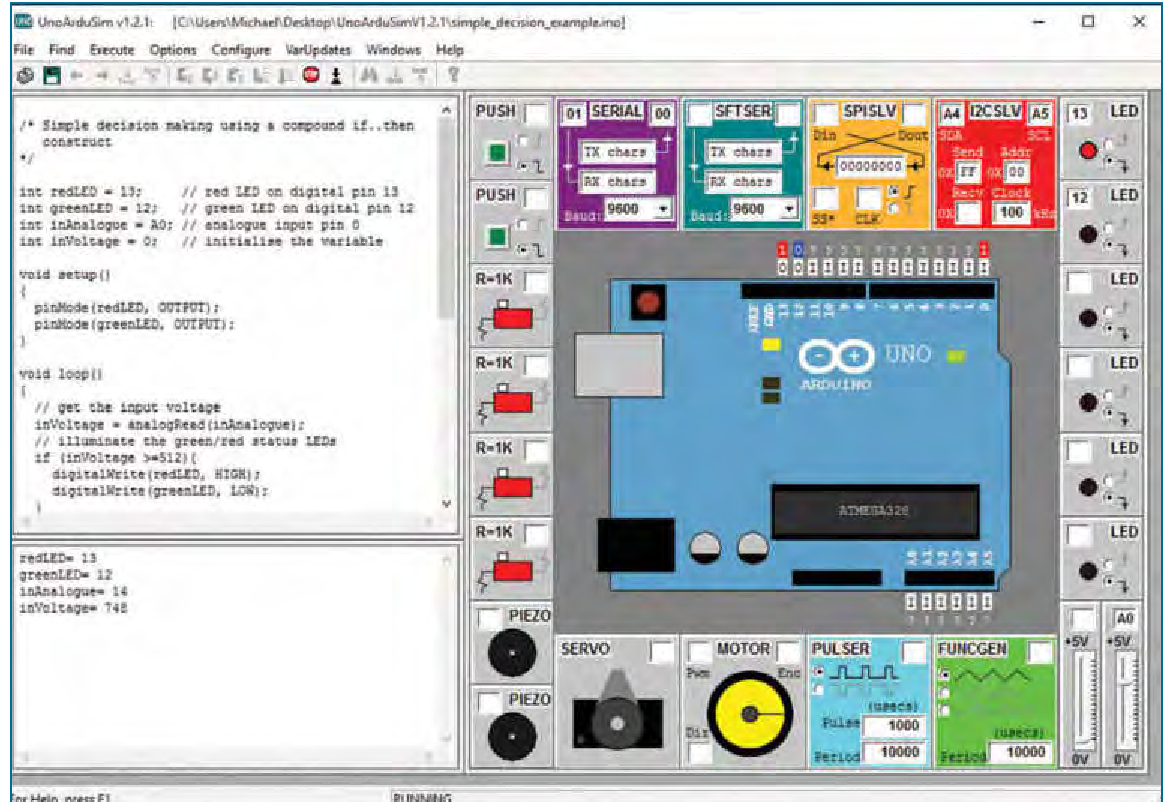


Fig.2.10. Using UnoArduSim to simulate the execution of Listing 2.2

```
count = 0;
while(count < 50){
  // code to be executed 50 times
  // each statement ending with ;
  count = count+1;
}
```

In this `wait` loop we increment the counter on every pass through the loop until it reaches 50, at which point the conditional expression evaluates to false and execution continues with the next statement in the code. Note this neater way of incrementing the count value, as follows:

```
count = 0;
while(count < 50){
  // code to be executed 50 times
  // each statement ending with ;
  count++;
}
```

In this case, `count++` is used to 'post-increment' the value of `count`. In other words, it takes the current value of `count`, adds one to it and places the new value back into the `count` variable.

Finally, here's an example showing how a simple `wait` loop could be used to flash an alarm LED ten times:

```
flashCount = 0;
while(flashCount < 10){
   digitalWrite(alarmLED, HIGH);
   delay(500); // wait half a second
   digitalWrite(alarmLED, LOW);
   delay(500); // wait half a second
   flashCount++;
}
```

The `do ... while` construct

The `do ... while` loop works in a similar fashion to the while loop, but with the exception that the condition is tested at the end of the loop, not the beginning. This means that the statements within the loop will always be executed at least once. The syntax is as follows:

```
do {
    // code to be executed at least once,
// each statement ending with ;
} while (conditional expression);
```

Here's an example of reading a pressure sensor and allowing it a short time for its output to reach a steady value:

```
do {
    delay(100);
    // wait for the value to settle
cp = readPressure();
// read the pressure sensor
}
while (x < 10);
```

In this example we are calling the `readPressure()` function 10 times before arriving at the final value returned from the sensor.

The `for loop` construct

The `for` loop is widely used in almost every computer language, and C is no exception. The construct is used to repeat a statement (or series of statements) whenever a condition evaluates true. If the condition evaluates false then the loop is exited and execution continues with the statement that immediately follows the loop. The `for` loop must be initialised at the outset and thus is a little more complex than the `while` loop. The basic syntax is:

```
for(loop initialization, conditional expression, increment){
   // statements to be executed if true,
   // each ending with ;
}
```

As with the while loop, a counter is often used to control the loop and this is incremented or decremented each time round the loop. This makes the construct ideal for use in any repetitive application, for example, checking the status of a number of I/O lines. It is important to remember that loop initialisation occurs only once and before the loop is executed for the first time.

The next code fragment shows how the ASCII character set can be sent to the serial printer. Note that, for this to run, we would first need to initialise the serial port interface using a line such as `Serial.begin(9600)`. Note also that we have declared the count variable, `i`, within the loop initialisation itself.

```
for (int i = 0; i <= 63; i++)
{
     testValue = 64 + i;
     Serial.print(testValue);
     Serial.print("\n");
     delay(100);
}
```

Program structure and layout

By now you should have gained some idea of what Arduino code looks like and how it is structured but before we go any further it is well worth explaining the layout of a C program in a little more detail. You may have noticed that the first few lines of code in a program usually take the form of a heading enclosed between pairs of characters, `/*` and `*/`, which constitute a comment block. Everything between these two characters is taken as plain text and, since this has no effect on program execution you can use as many lines of text here as you want.

The title comment block is usually followed by a number of variable declarations. The reason behind this is simply that, in the C language, variables *must always be declared* before they are used. In fact, declarations don't have to be placed at the beginning of the program code but the point at which they are declared (ie, their position in the program) can impose restrictions on the scope over which they can be used. However, since we often need to use variables on a global basis (ie, anywhere in our program code) we will often place them *before* any of the other code. Declarations involve assigning a variable type (see last month), a name and (optionally) an initial value.

Next follows code that's used for setting up. This code is placed in a function called `setup()` and it is executed at the beginning and only once. The `setup()` function is often used to specify the pin modes (ie, input or output) and to configure the Arduino's serial monitor, but if they are not being used the `setup()` function can simply be left empty.

The main program code is written inside a loop that executes forever (or until the reset button is pressed or the power is removed). This `loop()` function contains the functions that will execute when the program is being run. Each function takes the form of a block of code that is executed whenever the function is called. Functions can be the ones that are built into the language or they can be user-defined. This feature allows us to extend the basic language for our own needs with our user-defined functions calling other functions (both user-defined and in-built) as and when required. Function declarations take the form of one or more statements enclosed between curly braces, `{` and `}`. Note that each individual statement must end with a semi-colon, `;`.

As well as the block comments that we mentioned earlier, comments can be placed in-line. These consist of plain text appearing after two `//` characters and added at the end of the line to which they apply. As previously mentioned, comments provide us with a useful reminder of what's going on in the code and they can be invaluable when maintaining and debugging a program.

D1 (green)	D2 (red)	Condition
off	off	Alarm waiting to be set
on	off	Alarm set and waiting to be triggered
off	on	Alarm triggered and waiting to be cancelled

Table 2.3: Status indication for the simple Arduino-based security alarm

Listing 2.3: Code for the simple Arduino-based security system

```
* Single zone alarm with SET and CANCEL buttons */

int triggerInput = 7; // Break to trigger alarm
int setButton = 11; // Alarm SET button
int cancelButton = 12; // Alarm cancel button
int alarmSound = 4; // Siren
int setLED = 5; // Alarm SET LED
int alarmLED = 6;  // Alarm triggered LED
int setStatus = LOW; // SET button status
int cancelStatus = LOW; // CANCEL button status
int triggerStatus = LOW; // Trigger status

void setup()
{
      pinMode(triggerInput, INPUT);
      pinMode(setButton, INPUT);
      pinMode(cancelButton, INPUT);
      pinMode(alarmSound, OUTPUT);
      pinMode(setLED, OUTPUT);
      pinMode(alarmLED, OUTPUT);
      digitalWrite(alarmSound, LOW);
}

void loop()
{
      // Wait for set Button
      setStatus = LOW;
      while (setStatus == LOW && triggerStatus == LOW)
            // Loop must be closed to set the alarm
      {
            // Check to see if SET button has been pressed
            setStatus = digitalRead(setButton);
      }

      // Alarm has been set
      digitalWrite(setLED, HIGH);

      while (triggerStatus == LOW)
      {
            // Check if the alarm has been triggered
            triggerStatus = digitalRead(triggerInput);
      }

      //Alarm has been triggered
      digitalWrite(setLED, LOW);
      digitalWrite(alarmLED, HIGH);
      digitalWrite(alarmSound,HIGH);

      while (cancelStatus == LOW)
      {
            // Check if the CANCEL button has been pressed
            cancelStatus = digitalRead(cancelButton);
      }

      // Alarm has been cancelled
      triggerStatus = LOW;
      cancelStatus = LOW;
      // stop the alarm sound
      digitalWrite(alarmSound,LOW);
      // and also reset the LED indicators
      digitalWrite(alarmLED, LOW);
      digitalWrite(setLED, LOW);
}
```

Indenting (often by three or four spaces) is used to assist with program readability. This becomes particularly important when functions are enclosed within other functions. Finally, Fig.2.11 illustrates the various structural and layout features discussed here.

Get Real : A simple Arduino-based security systems

In this month's *Get Real* we are going to use the Arduino Uno as the basis of a very simple security system. Once again, we've minimised the need for anything much in the way of additional hardware, so you will only need a mini-breadboard and a few commonly available components to try it out.

You will need:

Arduino Uno with power supply
USB Type-A to Type-B cable
Computer with an available powered USB port
Mini-breadboard with a selection of coloured connecting leads
1 standard red LED (D2)
1 standard green LED (D1)
2 330Ω resistors (R1 and R2)
2 10kΩ resistors (R3 and R4)
1 4.7kΩ resistor (R5)
2 miniature push-button switches (S1 and S2)
Proximity switches (as required)
1 piezoelectric sounder (PZ1) (must be DC type)

Circuit

The complete circuit of our simple Arduino-based security system is shown in Fig.2.12. It uses six of the Uno's digital I/O lines; three of these are configured as inputs and three as outputs.

Two momentary action push-button switches, S1 and S2, are used to SET and CANCEL the alarm. The former of these switches is sensed by digital I/O pin-11, while the latter is sensed by digital I/O pin-12. The input loop is connected between digital I/O pin-7 and ground with R5 acting as a pull-up resistor so that the input will go 'high' whenever the loop is broken.

The green (SET) status indicator, D1, is driven by digital I/O pin-5 and the red (ALARM) status indicator, D2, is connected to digital I/O pin-6. The piezoelectric sounder (buzzer) is connected to digital I/O in-4. Note that the sounder needs to be a DC-operated component (not one that requires AC excitation).

Physical layout

The components are shown mounted on the mini-breadboard in Fig.2.13. The two status indicator LEDs will need to be connected with the correct polarity (see Fig.1.18 in last month's *Teach-In 2016*). If the layout proves problematic you could use a larger breadboard or a prototype shield (more of this in a future *Teach-In 2016*),

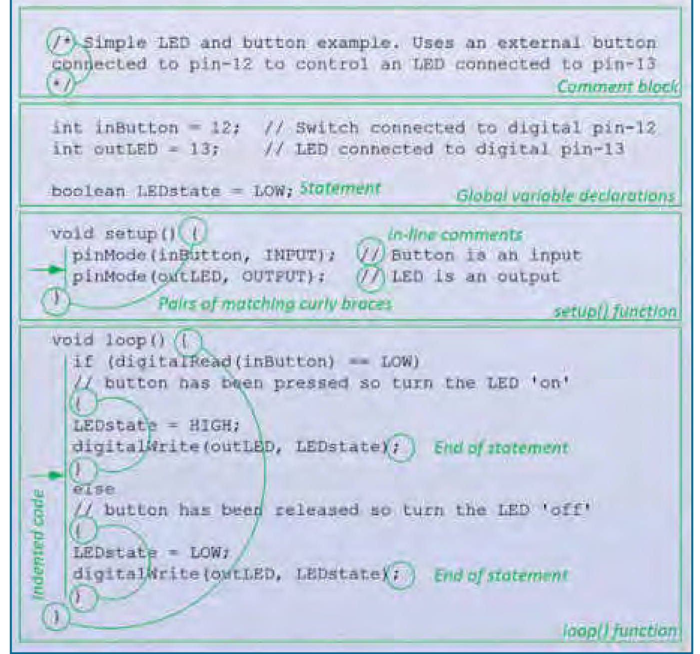


Fig.2.11. A simple Arduino program with various structural features identified

Fig.2.14. Actual breadboard arrangement

The input to digital I/O pin-7 is effectively a closed-circuit loop which, when broken, triggers the alarm. In a retail environment this can take the form of a continuous loop of insulated wire attached to any products that need to be protected. In order to remove an item the loop must be broken and this, in turn, will trigger the alarm. In other applications the loop can comprise one or more magnetically operated proximity switches, as shown in Fig.2.15. These are designed for discrete protection of doors and windows and they comprise a pair of moulded parts that need to be mounted adjacent to one another when the door or window to which they are attached is in the closed position. A permanent magnet is enclosed in one of the mouldings and a magnetic reed switch is mounted in the other. Proximity switches normally have a sensing range of between 10mm and 15mm and they are ideal for use in a range of basic security applications.

Code

Listing 2.3 shows the complete code for the simple Arduino-based security system. To help you understand what's going on we've included numerous comments in the code. Note that the main loop contains three `while` loops. The first of these waits for the alarm to be set (using S1). The second waits for the alarm to be triggered (when the zone loop is broken) and the third waits for the alarm to be cancelled (using S2). The two LEDs indicate as shown Table 2.3.

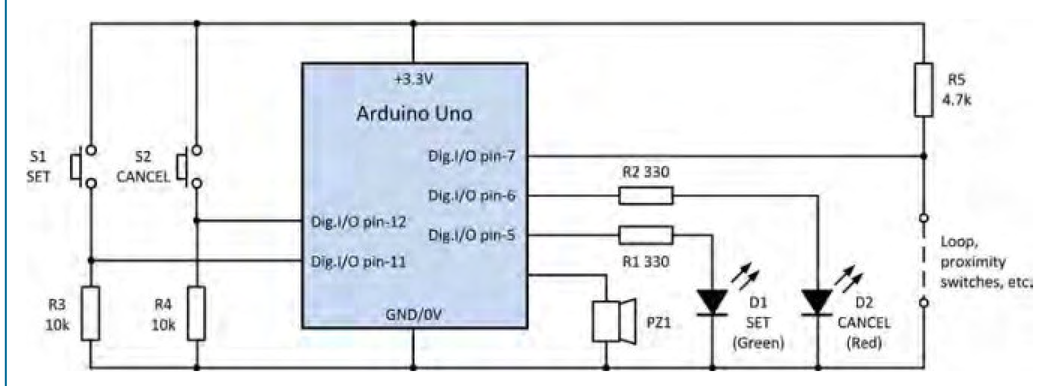
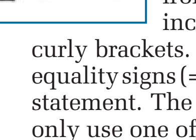


Fig.2.12. Circuit of the simple Arduino-based security system



As before, the code should be entered using the Arduino's IDE and then saved before compiling and uploading it to the Uno, as described in last month's *Arduino Workshop.* Don't forget to save your work by clicking on 'File' and 'Save' or 'Save As…' when you finish.

Next, click on 'Sketch' and 'Verify/Compile'. Where errors occur during compilation they often arise from missing semi-colons or incorrectly matched pairs of curly brackets. Note also the use of two equality signs (==) in the conditional loop statement. The compiler will fail if you only use one of them.

Testing

When you've corrected any coding errors that the compiler reports you will be ready to upload your code to the Uno. Just click on the upload arrow and watch

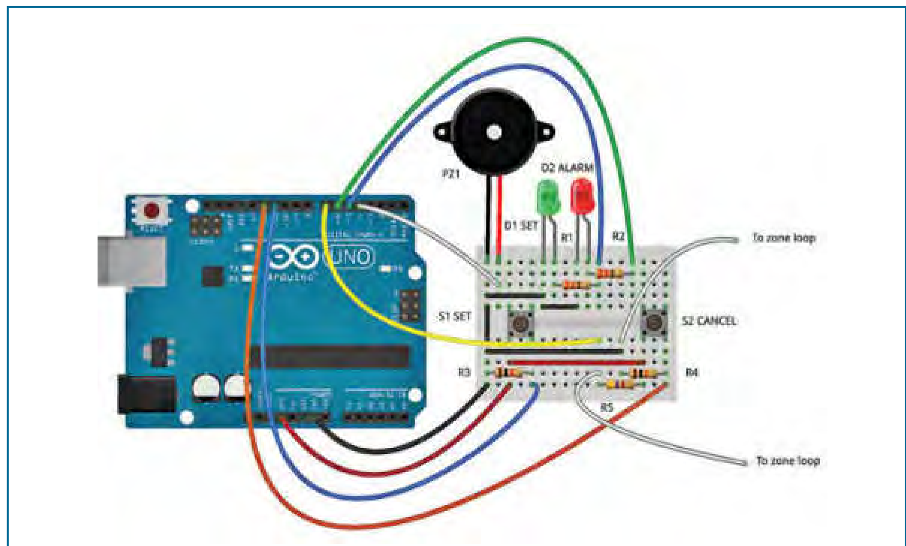


Fig.2.13. Fritzing breadboard arrangement for Fig.2.12

Fig.2.15. Some common types of proximity switch

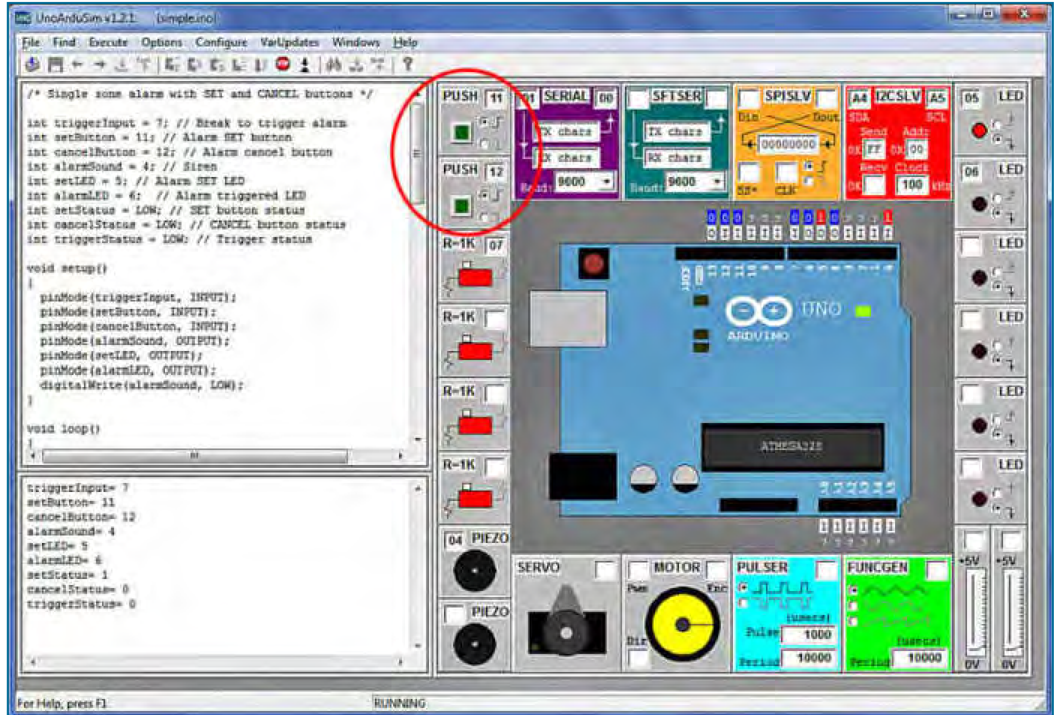


Fig.2.16. Using UnoArduSim to simulate the execution of Listing 2.3 (note that we've selected rising-edge triggering for digital inputs 11 and 12)

the progress report – but, before you do this it is important to make sure that the input loop is closed. The LEDs on the Uno should flash and the code should begin to execute. At this point neither of the status indicators, D1 and D2, should become illuminated. If you now press the SET button the green LED, D1, should become lit. This indicates that the alarm has been SET.

If you now break the loop the alarm will be triggered. In this condition the red status LED should be illuminated and the piezoelectric sounder should be operating. To reset the alarm you can press the CANCEL button, S2. Note that the alarm cannot be cancelled if the loop is still broken. To re-instate the alarm you will need to close the loop again, press the CANCEL button and, if all is well the red LED will go out and the circuit will then be ready to be put back into the SET state.

Going further

There's a great deal of scope for going further with our simple Arduino-based security alarm. The most obvious enhancement would be the addition of several more zones, each with an LED to indicate which of the zones has been triggered. All this needs is more of the digital I/O lines configured as inputs and outputs (one pair for each additional zone) together with some code that will poll each of the loops in turn to see if any of them have been triggered.

Another useful modification would be an entry/exit delay that would operate on the zone associated with access. This would allow an entry door to be opened and closed without triggering the alarm for a short period after pressing the SET button.

For applications in which a mains-operated sounder or floodlighting is to be controlled, the output from the piezoelectric sounder can be connected to a relay interface or a ready-made relay board like those described earlier in this month's *Teach In 2016*. All of this makes this simple project an excellent candidate for further experimentation.

Next month

In next month's *Teach-In 2016* we will look at displays and keyboards that can be used with the Arduino. To this end, *Arduino Workshop* deals with interfacing an alphanumeric LCD display and *Arduino World* looks at keypads and buttons. Our programming feature, *Coding Quickstart*, introduces string and string manipulation and the functions that you will need to read and print lines of text. Finally, *Get Real* will show you how to build a simple entry/access control system.