Lecture 4 CS3514 21/09/17

Revisiting the LED toggle code.

File Edit Sketch Tools Help

New

sketch_sep24a

```
boolean ledOn = false;

void setup(){
  pinMode(8, INPUT);
  pinMode (13, OUTPUT);
  digitalWrite(13, LOW);
}

void loop(){
  if(digitalRead(8))
    digitalWrite(13, (ledOn = !ledOn));
  delay(5);
}
```

Note on Code Shown above

Code to toggle the LED

NOTE 1) In C there is no type boolean — this is an arduino addition.

Note 2) In C, all non zero values are TRUE and zero is FALSE.

Thus we can write things like:

```
j = 10;
while (j--){ --- }
```

Note 3) In C, an assignment statement is an expression whose value is the value assigned to the variable. Thus

(ledOn = !ledOn) is a value in itself equal to the value of the on the left hand side

This is an example of an expression with a side-effect.

Sample code to debounce switch input

Limitation

toggle_led_delay_debounce

```
//Toggle output pin with a noisey input.
//Switching transition, from LOW to HIGH or from HIGH to LOW can be specified.
//The success of this solution depends on the characteristics of the device generating
//the noisey input. Here it is assumed to be a switch with will bounce for no more than
//MAX_BOUNCE_TIME milliseconds. In that respect, this code is no strictly portable
//although it may work for a variety of input with this character of "noisiness"
//John P. Morrison 26/9/13
```

A goal of this code is to exploit the

principle of least privilege

That is, each part of the code can only access resources that are directly relevant to it.

Thus:

1) There are no global variables

2) State information is exchanged to different parts of the program via function call/return

```
boolean debouncedRead(int pin, boolean state){
  if (digitalRead(pin) != state){
    delay(MAX_BOUNCE_TIME);
    return digitalRead(pin);
  }
  return state;
}
```

```
currentInputState = debouncedRead(inPin, currentInputState);
```

3) Use is made of Static local variables

```
void toggleOutPinWithInPin(int outPin, int inPin, int from, int to){
  static int currentInputState = LOW;
  static int prevInputState    = LOW;
  static int state             = LOW;
```

this ensures that CurrentInputState, prevInputState & state are not exposed to the outside environment, where they can be deliberately, or inadvertently, changed.

The static modifier results in space for the local variables being created on the HEAP rather than on the stack.

Without this modifier, local variables are created dynamically on the System stack when their associated function is called

This space is reused when the function terminates & its contents is lost.

The static modifier is used in order to preserve state between function calls without making the associate variable available outside of the function.

It does this by

(a) Creation on HEAP

(b) Restricting scope of access to the body of the function.

On re-entering the function, all of its static variables will have the same value that they had when the function was last used.

Static variables that are initialised as part of their declaration will be initialised only <u>once</u> – the first time the function is called.

4)

```
digitalWrite(outPin, (state = !state));
```

illustrates that C assignment statements are also expression. Their value is the contents of the L.H.S after assignment.

This is an example of an expression with a side-effect: evaluating the expression changes the state of the machine

In general, this is not the case and we say that those latter expressions are referentially transparent (the same net thing) with its result.

Is it a good thing to use expressions with side-effects?

It depend!

```
  if (prevInputState == from && currentInputState == to)
    digitalWrite(outPin, (state = !state));
  prevInputState = currentInputState;
}
```

In this case, it means we can write one statement rather than two (and do away with the need for the '{ }'

— Saving 3 lines of real-estate. This can improve readability.

It can bind a number of semantically related statements together into a single statement: thus ensuring that they always 'happen' together.

This reduces/eliminates the danger of the different actions becoming separated in subsequent edits, or of them being moved and giving rise to potential subtle errors.

Another advantage is that a good compiler can usually exploit the locality exposed in these expressions to generate better code.