University of Nottingham
UK | CHINA | MALAYSIA

EEEE1002
Applied Engineering Design

I2C Example Code Breakdown

Overview

- This PowerPoint discusses the code contained inside ‘Example 1: Arduino to Arduino I2C Communication’ and ‘Example 2: Master Device to ESP32 Slave I2C Communication’
- Remember that if you want to run Example 2 with an Arduino as the master, you need a voltage logic level shifter on both data lines (SDA, SCL)
- This PowerPoint covers:
  - The operators and syntax used in the code
  - Overview of the data handling and communication used

Data being transmitted and received can be handled in a variety of ways using a variety of data types – these examples present one such method; however feel free to implement your own

Note on the I2C Libraries Between Example 1 & Example 2

- The difference between example 1 and example 2 is that for Arduino I2C communication, the standard Wire.h library can be used
- In example 2, as an ESP32 is setup as a slave device, the ESP32 I2C Slave library (WireSlave.h) is used – however the code functionality and general I2C process is identical to Example 1
- This can be downloaded from: https://github.com/gutierrezps/ESP32_I2C_Slave
- It is recommended you read through and run the example programs contained in this library

Key Operators & Their Syntax

The following operators are used within the example code

Key Operators & Their Syntax

Bitwise Right Shift >>

- **Definition:** The right shifter operator causes the bits of the left operand to be shifted right by the number of positions specified by the right operand
- **Syntax:** variable >> number_of_bits;
- **Example:** *Assuming int has a variable size of 8-bits…*

```
int x = 241;          // 8-bit binary value of 11110001
int y = x >> 3;       // 8-bit binary value of 00011110; note that bits outside of the variable size are lost
```

If no bits are lost, shifting a binary number N bits to the right is equivalent to dividing the denary equivalent by 2^N

Key Operators & Their Syntax

Bitwise Left Shift <<

- **Definition:** The left shift operator causes the bits of the left operand to be shifted left by the number of positions specified by the right operand
- **Syntax:** variable << number_of_bits;
- **Example:** *Assuming int has a variable size of 8-bits…*

```
int x = 241;           // 8-bit binary value of 11110001
int y = x << 3;        // 8-bit binary value of 10000010; note that bits outside of the variable size are lost
```

If no bits are lost, shifting a binary number N bits to the left is equivalent to multiplying the denary equivalent by 2^N

Key Operators & Their Syntax

Bitwise AND &

- **Definition:** The AND operator compares the bits in the same position of the two variables, returning a value of TRUE (i.e. 1) if both of the bit inputs are TRUE – otherwise it returns FALSE (i.e. 0). See the following truth table and logic gate symbol.
- **Syntax:** variable1 & variable2;
- **Example:** *Assuming int has a variable size of 8-bits…*

```
int x = 23;          // binary: 00010111
int y = 24;          // binary: 00011000
int z = x & y;       // binary:  00010000 – equivalent to 16 in denary
```

2 - input AND gate

A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

This will be covered in H61INF: Information & Systems

Its purpose in the code is explained later…

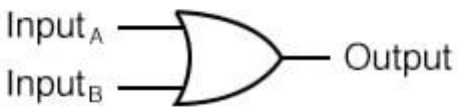
Key Operators & Their Syntax

Bitwise OR |

- **Definition:** The OR operator compares the bits in the same position of the two variables, returning a value of TRUE (i.e. 1) if either, or both, of the bit inputs are TRUE – otherwise it returns FALSE (i.e. 0). See the following truth table and logic gate symbol.
- **Syntax:** variable1 | variable2;
- **Example:** *Assuming int has a variable size of 8-bits…*

```
int x = 23;        // binary: 00010111
int y = 24;        // binary: 00011000
int z = x | y;     // binary:  00011111 – equivalent to 31 in denary
```

2 - input OR gate



A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

This will be covered in H61INF: Information & Systems

Its purpose in the code is explained later…

I2C Communication & Data Handling Overview

I2C Communication Overview

- We will follow the code used in Example 1, except with a simple case of just transferring the variable x
- Remember the communication and data handling process is identical between Example 1 and Example 2
- We will assume that the int variable size is 32-bits, however the principle described is the same – just with a varying amount of bytes (1 byte = 8 bits)
- The int variable size varies depending which microcontroller you are using – research this before you write your code

A Note on Signed Integer Representation

- If we have an N-bit number, it can store 2^N unsigned integers, with the range 0 to $2^N - 1$
- However if we want to represent negative (i.e. signed) integer, we have to use the most significant bit to indicate if the value is positive or negative – this method is called *'Two's Complement'*, do your own research on how use this
- As a result, we can still store 2^N values, however the range is now between -2^{N-1} to $2^{N-1} - 1$
- For example, if we have an 8-bit variable, it can store 128 values. For unsigned integers, the range is 0 to 255. For signed integers, the range is -128 to 127

Integer Representation

```
int x = -210;
```

- Let us take a simple case, with x equal to the integer value of 210
- In binary this is 11010010 (represented by 0b00000000000000000000000011010010 in the code – remember the 32-bit variable size!)
- This is equivalent to four bytes:

000000000000000011010010

00000000	00000000	00000000	11010010
Byte 1 Bits 32-25	Byte 2 Bits 24-17	Byte 3 Bits 16-9	Byte 4 Bits 8-1

Be careful, although bits are always numbered right to left – the starting number can either be 1 or 0, we will start numbering at bit 1

Operating On & Extracting Bytes

```
Wire.write((byte)((x & 0x0000FF00) >> 8));
Wire.write((byte)(x & 0x000000FF));
```

- Starting from inside the brackets and working out…
- We first just want the 8 bits we are interested in
- Like how 0b represents a binary number, 0x represents a hexadecimal number; with F being 15 in denary or 1111 in binary – note that each character in hexadecimal is equivalent to 4 bits
- In order to get bytes 3 and 4, that were labelled in the previous slide, we AND the variable x with the hexadecimal equivalent of 11111111 in the positions of the bits we want; as a 0 ANDed with another bit results in 0 and a 1 ANDed with another bit results in 1

Operating On & Extracting Bytes

```
Wire.write((byte)((x & 0x0000FF00) >> 8));
Wire.write((byte)(x & 0x000000FF));
```

- But what about bytes 1 and 2!
- In order to reduce the amount of data we are sending, we are not extracting these bytes of data – as in our EEEBot, we only need to send values between -255 and 255 (requiring 9 bits – remember Two's Compliment!)
- Bytes 1 and 2 (holding bits 32 to 17) are only needed if we wanted to send signed integers between -2,147,483,648 and -65,536 or between 65,535 and 2,147,483,647 – rather excessive for us!

Operating On & Extracting Bytes

```
Wire.write((byte)((x & 0x0000FF00) >> 8));
Wire.write((byte)(x & 0x000000FF));
```

- With x = 210, `x & 0x0000FF00` looks like:

```
x:        00000000000000000000000011010010   & (AND)
          00000000000000001111111100000000
          --------------------------------
Result:   00000000000000000000000000000000
```

All the bits not of interest are set to 0

- With x = 210, `x & 0x000000FF` looks like:

```
x:        00000000000000000000000011010010   & (AND)
          00000000000000000000000011111111
          --------------------------------
Result:   00000000000000000000000011010010
```

- *The need for this operator is best seen with a negative number…*

Operating On & Extracting Bytes

- If x = -210, the binary representation would instead be 0b11111111111111111111111100101110
- Therefore to get bytes 3 and 4…
- With x = -210, `x & 0x0000FF00` looks like:

```
x:       11111111111111111111111100101110   & (AND)
         00000000000000001111111100000000
         --------------------------------
Result:  00000000000000001111111100000000
```

All the bits not of interest are set to 0

- With x = -210, `x & 0x000000FF` looks like:

```
x:       11111111111111111111111100101110   & (AND)
         00000000000000000000000011111111
         --------------------------------
Result:  00000000000000000000000000101110
```

Operating On & Extracting Bytes

```
Wire.write((byte)((x & 0x0000FF00) >> 8));
Wire.write((byte)(x & 0x000000FF));
```

- Back to x = 210
- As I2C communication can only send information one byte at a time, we need to split the variable, x, into 8-bit segments of data
- Note; as we are now ignoring bytes 1 and 2 of the 32-bit variable, x we will only split x into two bytes (bytes 3 and 4) of data
- As a byte is only 8 bits and `(byte)` ignores bits 9 and above, we need to 'shuffle' our 8 bits of interest (in each case), to the right X times until they are in the positions of 8 to 1
- This is where we use the Bitwise Right Shift operator (>>)

Operating On & Extracting Bytes

```
Wire.write((byte)((x & 0x0000FF00) >> 8));
Wire.write((byte)(x & 0x000000FF));
```

- For `(byte)(x & 0x000000FF)`, the bits of interest, relating to byte 4, are already in positions 8 to 1 so no shifting is required
- However, for `(byte)((x & 0x0000FF00) >> 8)`, the bits of interest, relating to byte 3, are in positions 16 to 9 and so required shifting to the right by 8 bits i.e. subtracting 8 off the bit's position
- The data (byte 3) for `(byte)((x & 0x0000FF00) >> 8)` looks like: `00000000`

Although for this value of x, byte 3 is 0 – this will not always be the case i.e. when x < -128

- The data (byte 4) for `(byte)(x & 0x000000FF)` looks like: `11010010`

Transmitting & Receiving Bytes

```
Wire.write((byte)((x & 0x0000FF00) >> 8));
Wire.write((byte)(x & 0x000000FF));
```

```
uint8_t x16_9 = Wire.read();
uint8_t x8_1 = Wire.read();
```

- Each byte is then sent sequentially along the I2C data lines using `Wire.write` - in Example 2 the bytes are first written in ‘packer’
- In the slave code, each byte is received sequentially, i.e. in the same order as it was sent, and is assigned to a variable using `Wire.read()`
- In Example 2, `WireSlave.read()` is used instead
- Therefore, the first byte received is equivalent to byte 3 and the second is byte 4 – as these are bytes we read them into an 8-bit integer type so the bit-size does not change but they can now be manipulated as integers; rather than bytes

Data Types

- Remember these examples show just one way of handling I2C data, the most suitable data types to use will vary with the code purpose
- Just remember that the incoming data will be one byte long (8-bits)
- This must be considered when handling the data otherwise conversion into other data types may result in unpredictable results with bits either being lost or added
- A quick google of the Arduino IDE data types will show their bit lengths; https://learn.sparkfun.com/tutorials/data-types-in-arduino/all

Reconstructing the Number

```
x = (x16_9 << 8) | x8_1;
```

- As a recap, the data stored in the two variables relating to x is:

Variable	Bits Stored	Byte that the Bits Relate to from the Original Number (Slide 12)
x16_9	00000000	Byte 3
x8_1	11010010	Byte 4

- So now we need to reconstruct the original number that was stored in ‘x’ in the master code…

Reconstructing the Number

```
int16_t x = 0;
x = (x16_9 << 8) | x8_1;
```

- Working from inside the brackets outwards, we first shift the bits relating to byte 3 to the left by 8 bits
- This effectively undoes the right bit shift that was performed in the master code, in order to restore the bits original position in the original integer
- The bits aren't lost as x has been defined as a 16-bit integer which by default has a value of 0 i.e. 16 0-bits
- Therefore, from the result from `x16_9 << 8` looks like `0000000000000000`
- Again, as for x = 210 bits 16 to 9 are all 0, this is best shown with a different example…

Reconstructing the Number

```
int16_t x = 0;

x = (x16_9 << 8) | x8_1;
```

- Referring back to x = -210 from Slide 16…
- Byte 3 has a value of `11111111`
- Therefore, the result from `x16_9 << 8` looks like `1111111100000000`
- Now you can clearly see that the bits have all been shifted to the left by 8-bits, restoring their original position in original number

Reconstructing the Number

```
int16_t x = 0;

x = (x16_9 << 8) | x8_1;
```

- Back to x = 210…
- Now we have a 16-bit value with only bits 16 to 9 with the correct value, and an 8-bit value with only bits 8 to 1 with the correct value
- Note that is we take an 8-bit variable and assign it to a 16-bit variable, by default the extra bits i.e. bits 16 to 9 are set to 0's
- Likewise, if we put an operator between a 16-bit and 8-bit number, the extra 'empty' bits (16 to 9) are treated as 0
- Therefore, in order to combine these two values and restore the bitstream of the original value, we use an OR operator

Operating On & Extracting Bytes

- We use the OR operator as no bits are changed due to the fact that if a ‘1’ bit is ORed with a ‘0’ bit, the result is 1
- We don’t need to worry about when both bits are 1 as these never occur because only the 8 ‘potentially non-zero’ bits in each variable never are in the same position
- The result in x is therefore;

```
x16_9 << 8:    0000000000000000   | (OR)
      x8_1:    0000000011010010
               ----------------
Result in x:   0000000011010010
```

- In denary, this is equivalent to 210 i.e. the original number has been successfully transmitted and received
- Remember x is in Two’s Compliment if you are working in binary, however the int variables handles this automatically

University of Nottingham
UK | CHINA | MALAYSIA

End of I2C Example Code Breakdown