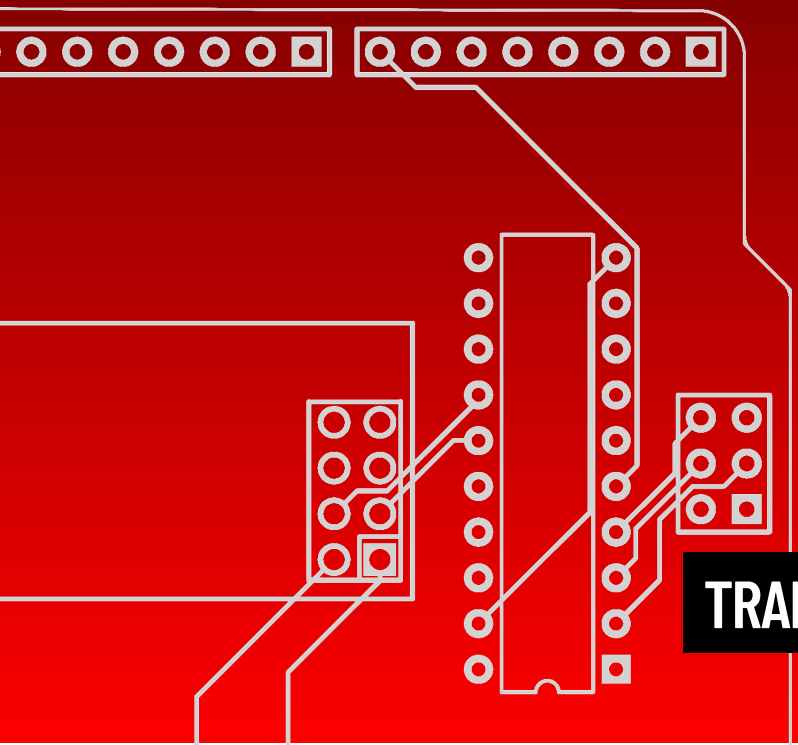
MORSE MESSENGER

MLC

.·-. -·-· · ··· ··-· / -- --- .-· ··· · / -- · ··· ···



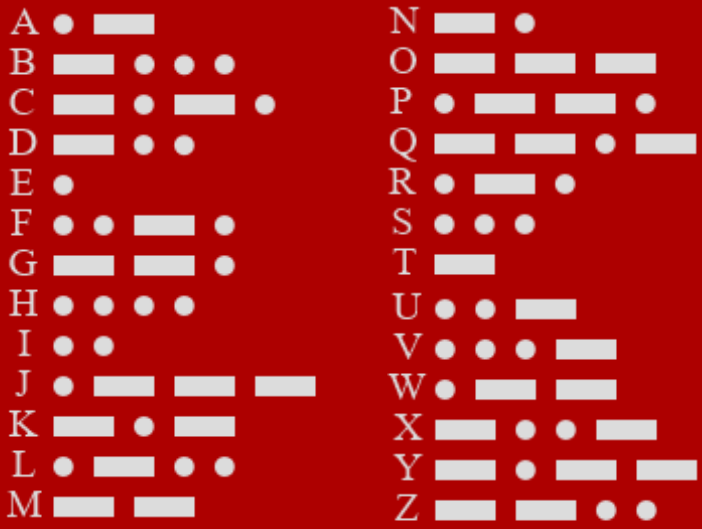
TRANSMITTING

HARDER

TRANSMITTING

Before the world had smartphones, email, or even the humble landline, there was Morse code. Developed in the early 19th century, this revolutionary system of dots and dashes - often used by spies - was a way of encoding data such that it could be transmitted across oceans, via radio, and decoded by humans.

The table below shows how each character of the alphabet was encoded by a series of dots and dashes. Notice how the letter 'E' is a single dot, as this is the most frequent letter used in commination. On this page alone there are 62 occurrences of 'e'.



YOUR TASK

As a transmitter you will be sending a message to your partner for them to decode, but we first need a way for our processor to know what message you wish to send.

To get started we're going to go back about 150 years and use Morse code. This relied on the timings of each pulse of light to represent either a dot or a dash, a dot being a short pulse and a dash being a longer one. A small delay was used to represent a space between two characters which when written out is represented by a forwards slash. For an example, look at the front cover.

STEP 1:

Firstly download the code stubs and dependent libraries from the following URL. If you already have an instance of a given library on your device, you won't need to install it again!

HTTPS://EXAMPLE-URL/RESOURCE.ORG

MLC |

STEP 2:

Open the code in the Arduino IDE and find the following subroutines, you will be editing all 3.

```
String InputMorse();
```

```
String AppendNewMorseSegment();
```

```
int DetermineInput();
```

Morse code relied entirely on timings to distinguish between a dot and a dash and we're going to use that same principle.

The first subroutine, **InputMorse()**, returns the full message to be transmitted after a specified time of user inactivity has passed. Using the pre-set values for timings, the device must be inactive for 6 seconds before sending the message.

The second subroutine, **AppendNewMorseSegment()**, is called by InputMorse() and returns a String of the Morse segment the user has just entered. This is added to the full transmission after 2 seconds of inactivity.

The final subroutine, **DetermineInput()**, is called by AppendNewMorseSegment() and is used to determine if the user is entering a short press (a dot) or a long press (a dash).

TIMER CLASS

Inside the UKESFMorse.h header file, there is a pre-written Timer() class. You can create instances of a timer and call the respective methods anywhere in the program scope. The syntax is show below:

```
Timer myTimer;

//Instantiates a new instance of a timer, identified by “myTimer”, this timer has started counting up
```

```
myTimer.Read();

//Returns how long the timer has been counting for in milliseconds
```

```
myTimer.Reset();

//Resets the timer to 0
```

STEP 3:

Lets work through the first subroutine, InputMorse(), together.

```
String InputMorse(){

  String ToReturn = "";
  #define LongWait 4000

  while(0)
  {
    if(0)
    {
        ToReturn += AppendNewMorseSegment();
        CenterMessage("");
    }
  }

  return ToReturn;
}
```

For this subroutine, while the device has been inactive (received no input) for less than 4 seconds (LongWait), we want to check if the button is being pressed, and if so add the input to our transmission.

For this we'll first need to instantiate a new timer outside of any loops and use it in our while() loop condition.

Task 1: Replace the 0 in while(0) with a coded version of "while the timer reads less than LongWait".

Task 2: We now want to read if the button has been pressed. You may have noticed in the Setup() function we set the ButtonPin pin mode to INPUT_PULLUP, this stops the pin from "floating" using an internal pullup resistor and means that when we press the button, its state reads LOW.

See if you can make some changes to the code below and add it to your subroutine in place of if(0)!

```
if(digitalRead(ButtonPin)==State){
}
```

Task 3: The final part of this subroutine is resetting the timer so that it doesn't forever read as inactive.
Where should you use the .Reset() method in this subroutine? Bearing in mind that the timer is used to measure how long the device has received no button input.

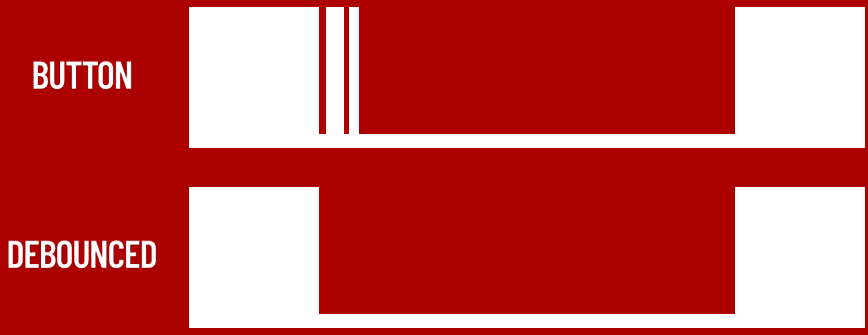
STEP 4:

Good job! You seem to be getting the hang of this, so we'll give you a little bit less guidance for the second subroutine.

You will again need to instantiate a new timer and use this in the while() loop condition and reset it if a button input is successfully read.

It might help to first look at the DetermineType() subroutine and notice that if it returns -1, this is classed as an invalid button press and should be ignored.

This is because of a feature of the button meaning that when its pressed it might get stuck and accidently record a single button press as two, that's where debouncing comes in.



Debouncing an input refers to filtering out very short presses sometimes with a capacitor or in this case with some code.

In the DetermineInput() subroutine you'll notice that if the duration of our ,press adjusted for with the DebounceError, is less than 0, we reject it and that's the same as filtering out these short presses that occur because of how the button is manafactured.

STEP 5:

This final subroutine, DetermineType(), is a little different. We still need to instantiate a timer but this time it won't come up in our while() loop.

Instead while the button is pressed we just want the timer to count, and when its finished we want to calculate how long it was pressed for, whilst accounting for our DebounceError and then assign this duration to the variable "Period".

READY TO TEST

Make sure both you and your partner are using the same address (any 5 bit binary number) and the same channel (any integer between 0 and 255)

```
const byte address[6] = "00001";
const int Channel = 100;
```

These are the default values chosen for you, but change, at least, the channel to a value that your other classmates aren't using, this will ensure only your partner receive the intended message, not your classmates.

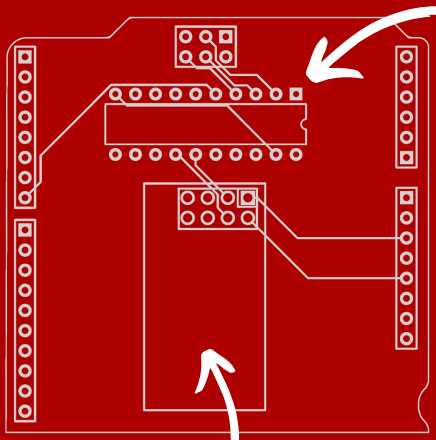
MLC |

EXTENSION:

Try playing around with the values of LongWait, InputTime and ShortPress. See how this affects the device! Or maybe set up some additional Morse characters with your partner, like a space, exclamation mark, or even emoji!

If you're interested in the rest of the code click on the UKESFMorse.h tab near the top of the IDE, here you can see where other libraries and subroutines have been implemented!

UNDERSTANDING THE CIRCUIT



74LVC245AN

This is a logic level converter which allows the radio module to communicate with the microprocessor beneath it. See if you can find out why the NRF24L01 radio module requires a logic level converter when used with the Seeeduino Lotus

NRF24L01

This is the NRF24L01 transceiver. It's a 2.4GHz radio module used for communication, some older phones would've used a similar circuit!

2.4GHz RADIO

Radio waves at 2.4GHz are part of the microwave spectrum, widely used in wireless communication. The “GHz” measures frequency—2.4GHz equals 2.4 billion cycles per second! Signals at this frequency have become essential in modern technologies.

One common application is Wi-Fi, which often operates in the 2.4GHz band. This frequency is favoured for its ability to travel farther and penetrate obstacles like walls, compared to higher frequencies. Bluetooth devices also utilize 2.4GHz signals to connect wirelessly, making it integral to smart gadgets and wearable tech.

However, because many devices use 2.4GHz, interference can sometimes occur, affecting signal performance. But by using different channels which use very slightly different frequencies or bands, we can filter out the noise by only listening to the frequency we want!

MLC