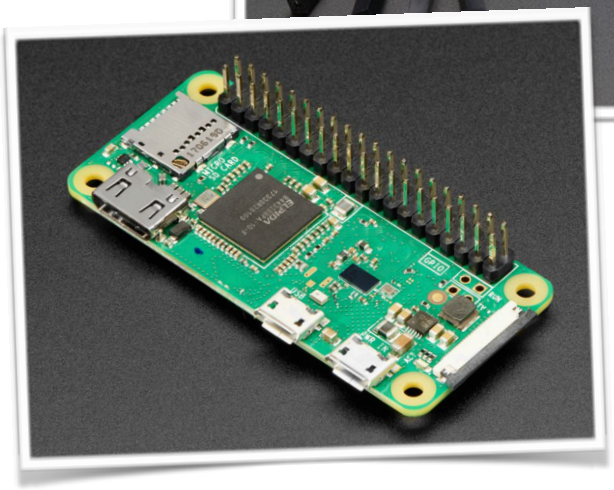
EECS 2031A 2018

Lab #3 Sending Morse Code



EECS 2031 Lab #3

Sending Morse code

This lab involves writing code that reads in (from the keyboard) a line of text and outputting that text as Morse code through a blinking LED. This lab thus involves a couple of different technical components

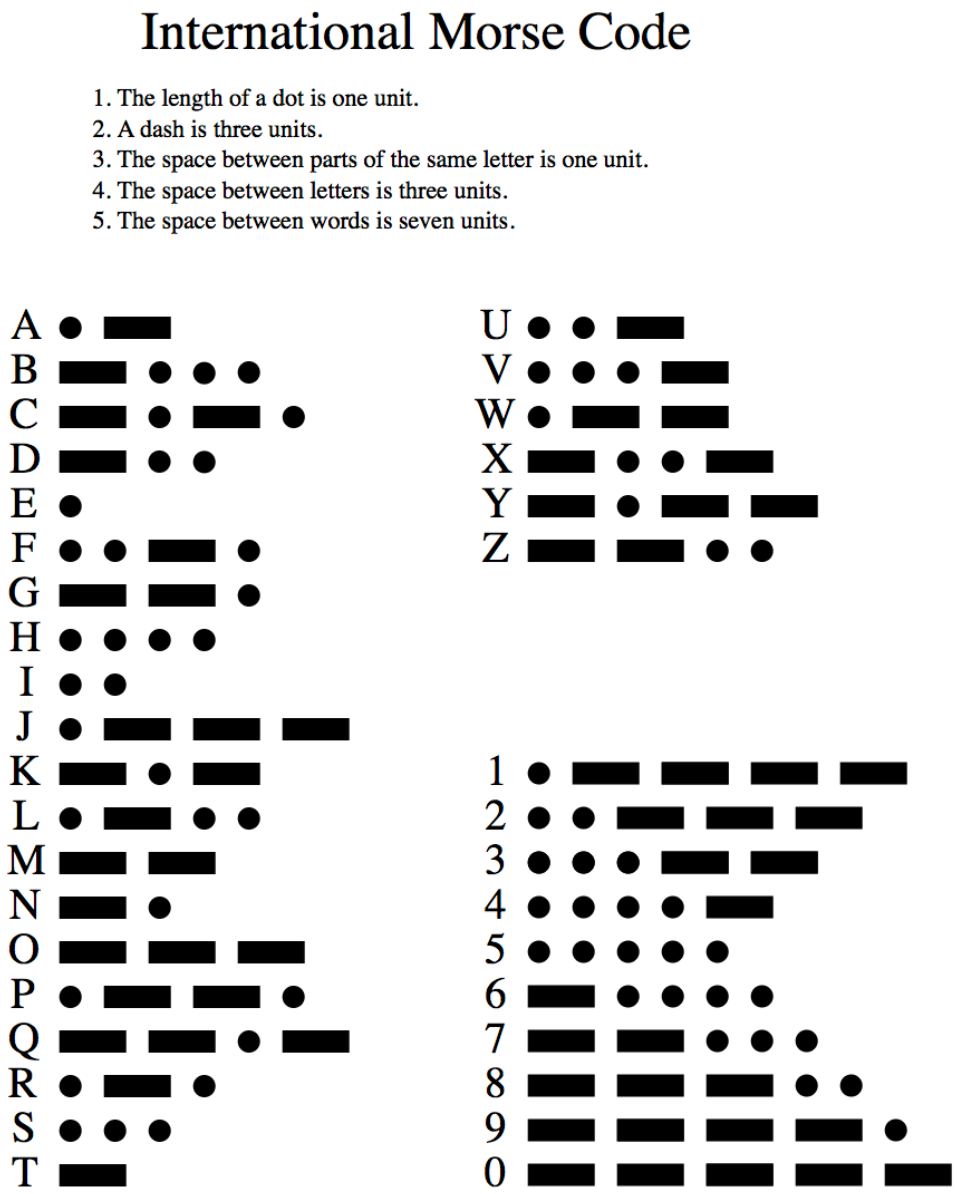
- Controlling a LED under software control. This was mostly covered in the previous lab but we will augment the code that was done there slightly.
- Obtaining a line of text and processing it character by character.
- Extracting individual letters in the line of text and obtaining the necessary pattern of dots and dashes to encode them to blink the LED.

You will write the code to do the first and last of these tasks. You will be provided with code that does the middle portion.

In order to complete this lab and obtain a grade for it you will have to demonstrate to your TA that you have completed the various steps in this lab. This lab, like all labs in this course, are intended to be experiential. You are encouraged to ask for help from both the TA's and your fellow classmates, and to help you fellow classmates in completing the lab. That being said, you need to master the material being presented here.

Morse code

Morse code is a mechanism for transmitting textual information as a sequence of dots and dashes, typically provided by light or electrical



signal. Developed in 1836 by Samuel Morse, information in Morse code is sent as a series of short signals (referred to as dits and represented as dots) and long signals (referred to as dahs and represented as dashes). Morse code relies on a consistent clock which defines a unit of time. Dots are 'on' for one unit of time, 'dashes' are on for three units of time. There

is one unit of time between symbols, three units of time between letters and seven letters of time between words. Upper case letters and numbers have a standard and well known translation into Morse code as shown before.

To make this all perhaps a bit clearer, imagine translating the sequence ‘hello wo’ into morse code, that would translate as shown below with _ being a unit of time equal to the duration of a dot, * being a dot and = being a dash.

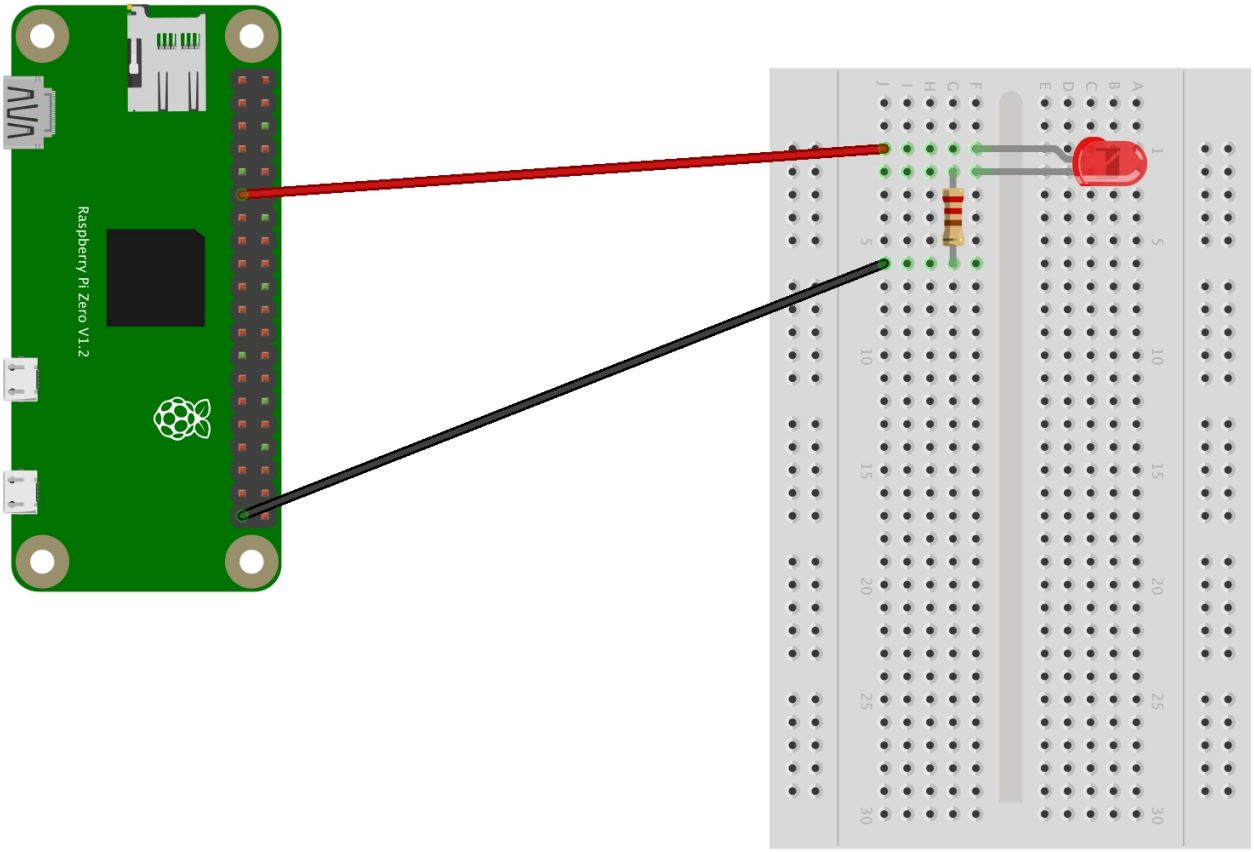
```
*_*_*_*___*___*_=_*_*___*_=_*_*___=_=_=_______*_=_=___=_=_=
```

Blinking a LED

The circuit given below is the same as the circuit that you used in the previous lab. Re-build it using your hardware. Remember that the resistor is required in the circuit and the LEDs have a polarity that must be followed. You can verify that your circuit is correct by re-running your code from last week. (Hope you kept it.)

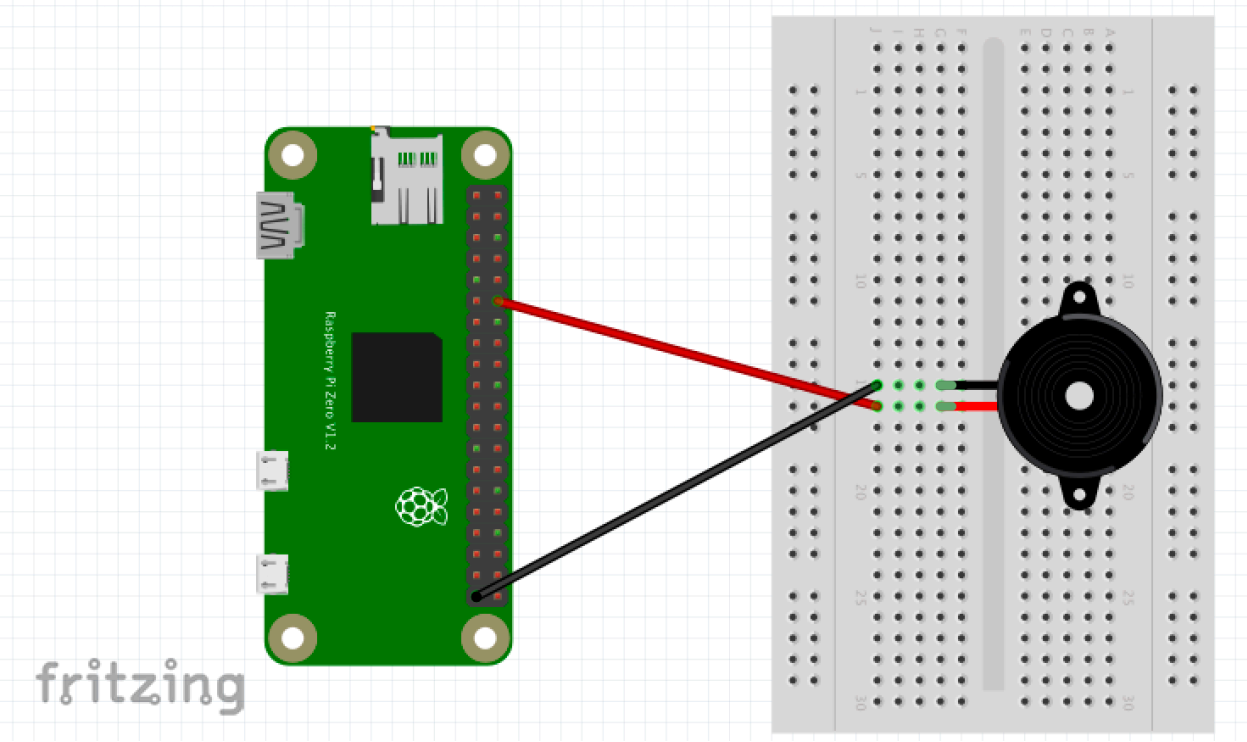
The blink.c file that you used last week is included below for your reference.

```
#include <wiringPi.h>
int main (int argc, char *argv[])
{
  wiringPiSetup () ;
  pinMode (0, OUTPUT) ;
  for (;;)
  {
    digitalWrite (0, HIGH) ; delay (1000) ;
    digitalWrite (0,  LOW) ; delay (500) ;
  }
  return 0 ;
}
```



Playing a buzzer

In order to wake up the Morse operator, we will provide a short beep to the operator so they will attend to the signal. (You will likely want to comment out your code that does this once you have it working.) There is a small buzzer in your kit. It is a small black cylinder with two wiring pins, one slightly longer than the other. The longer end connects to power, the shorter end connects to ground. It takes 3.3V to make a buzzing signal. It does not need a resistor to limit the current running through it. The circuit diagram below shows how you might wire it up. Connect it to wiringPi pin number 1 and to ground. Modify the blink code you used in the last lab to provide intermittent power to wiringPi pin number 1.



Note that the grounds are common, so you can connect them in lots of ways. Also observe that your buzzer is probably a lot smaller than the buzzer shown in the sketch above.

Key to the generation of Morse code is the ability to output dots and dashes. So lets start writing some code. First, create a directory in Documents called lab03 and change directory there. Associated with this lab are a couple of files you will find helpful.

The Code

The code provided in lab3, as will be the case for all code provided throughout the rest of this course, will be provided through github. There are many reasons for this (learning git and using GitHub being two), but its also a very convenient way of distributing the code to you. Make sure

that your RaspberryPi is connected to the Internet, change directory to lab03 and in that directory type

git clone https://github.com/michaeljenkin/morsecode.git

After a few moments you will have a copy of the code that can be found on the GitHub server under michaeljenkin. You will find you have the following files

- Makefile - this is a Makefile for the application
- main.c - the main program (you should not have to modify this)
- morsecode.c - this is the code that you will modify
- morsecode.h - an include file that defines the functionality of morsecode.c

So it is worth looking through the code here to understand what it does. Lets start with main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "morsecode.h"

int main(int argc, char *argv[])
{
  char *p;
  if(argc != 2) {
    fprintf(stderr,"usage: %s text\n", argv[0]);
    exit(1);
  }
  init_morse();
  play_buzzer(500);
  wait_word();
```

```
  p = argv[1];
  while(*p) {
    if(*p == ' ')
      wait_word();
    else {
      char *v = char2morse(*p);
      printf("character %c translates to %s\n", *p, v);
      while(*v) {
        switch(*v) {
        case '*' : send_dot(); break;
        case '=' : send_dash(); break;
        default:
          /*NOTREACHED*/
          fprintf(stderr,"internal logic error\n");
          exit(1);
        }
        if(*(v+1) != '\0')
          wait_dot();
        else if(*(p+1) == ' ')
          wait_word();
        else if(*(p+1) != '\0')
          wait_letter();
        v++;
      }
    }
    p++;
  }
  return 0;
}
```

So this is the main function, execution starts here. What this code does is look at its one argument (and complain if there is not exactly one argument) and then processes it character by character. For each character it calls the function char2morse() with that character and gets back a string. (This is the Morse code version of that character.) It process this returned string which consists of exactly the characters '*', which

represents dots, and '=', which represents dashes. It then calls send_dot() for each dot and send_dash() for each dash. It then does some clever logic to put the proper amount of space around individual symbols, letters and word breaks.

You will note that the code includes <wiringPi.h> and "morsecode.h". You should have some experience with wiringPi now, but its documentation can be found on the web. morsecode.h defines the resources that you can find in morsecode.c. This code is given below

```
#include <stdio.h>
#include <wiringPi.h>
#include "morsecode.h"

#define TIME_UNIT 250
#define DOT_TIME (TIME_UNIT)
#define DASH_TIME (TIME_UNIT*3)
#define LETTER_SPACE_TIME (TIME_UNIT*3)
#define WORD_SPACE_TIME (TIME_UNIT*7)


void init_morse(void)
{
}

void send_dot(void)
{
}

void send_dash(void)
{
}

void wait_letter(void)
{
  printf("wait for letter\n");
```

```
}

void wait_dot(void)
{
  printf("wait for dot\n");
}

void wait_word(void)
{
  printf("wait for word\n");
}

void play_buzzer(int msec)
{
}

char *char2morse(char c)
{
  return "***===***";
}
```

You will see that these are basically empty definitions for the functions that you need to use. Now to compile this, you could type

gcc -Wall -ansi -pedantic main.c morsecode. -lwiringPi -o main

And this will work. Give it a try. The code skeleton as provided compiles and runs, but it doesn't do anything really useful, yet.

If you only edited one of the files this seems a bit of a waste. And it is. A standard tool in Unix is the Makefile (you have one in this project). It contains the following

```
OBJS=main.o morsecode.o
LDFLAGS=-lwiringPi
```

```
CC=gcc
CFLAGS=-Wall -pedantic

%.o: %.c morsecode.h
     $(CC) $(CFLAGS) -c -ansi $<


main:    ${OBJS}
     $(CC) $(CFLAGS) ${OBJS} -o main $(LDFLAGS)
```

To execute a Makefile, you type ‘make’. But to make this interesting, first execute

touch *.c

And then type make. You will find that the individual modules are compiled and then linked together. Type ‘make’ again, and the Makefile system will tell you that everything is up to date. Try touching just one of main.c and morsecode.c, and then typing make. You will find that everything works as you would like it to.

With everything up to date, type make to see that everything is good. Now touch morsecode.h and type make again. You will see that the make system ‘knew’ that if the include file had been updated, then all of the C code should be re-compiled. How did it know that?

So it knows it because of the structure of the Makefile. First, there is one target ‘main’. To make ‘main’, all of the ${OBJS} files need to be up to date. ${OBJS} is defined towards the top of the file. So how does the make system know how to check if a file is up to date? That is given by the line %o: %c morsecode.h. This rule says that a .o file depends on its corresponding .c file and morsecode.h, and to update the .o file execute

the line below this. This (re)-compiles the appropriate .c file. Finally, back at the 'main' rule once everything is up to date, the various .o files are linked together with their corresponding libraries.

Makefiles are very useful and can be difficult to construct. We will go through the make system in class and (almost) all labs from now on will have a Makefile.

So your job here is to complete the application by filling in the functions in morsecode.c. To make things easier, a couple of points

- You only need to support the symbols 0 through 9. Note: you can make the code simple or much too difficult when you write the function char2morse(). Keep it simple.
- The code is executed by typing ./main "101 100 10101" This will cause your code, when it is working, to beep, then output in morse 101 100 10101 with the appropriate spaces between words.
- Test individual parts as your complete them.
- Once you have the buzzer working, comment it out until you are ready to have your solution marked.

LINUX tools introduced in this lab

- git - use the github repository system
- make - create (make) a directory
- Compiling multiple files into a single application

- touch - touches a file (updates its last access date among other things).

Grading

To obtain a grade for this lab you must show your TA your Raspberry Pi running the Morse code program. You can do this in your lab, during any of the TA's office hours before your next lab, or at the very start of your next lab. Late solutions will not be accepted.

Extras

SOS is an internationally recognized signal in Morse Code. Surprisingly it does not mean 'Save Our Ships' or 'Save Our Souls'. It is just easily recognized when heard.

SOS is also the name of an ABBA Song.

There is a good article on Samuel Morse here.