BBC micro:bit
Bit:Commander Buzzer

Introduction

The buzzer on the Bit:Commander is connected to pin 0. This is convenient because that is the default pin. The minor inconvenience is that the potentiometer shares the same pin. Although you can use both in the same project, you need to think carefully about what is happening at any point in the program for that to make sense.

If you are using the Bit:Commander as a controller for a micro:bit game, then you have the buzzer for the game sound effects. If you are using the Bit:Commander as a controller for a project, then it's good as an alert. You can also use it to play a different tone for each button press.

Programming

It's pretty easy to get sound going when the buzzer is connected for you. Everything on the page in the components page is going to work without any alteration to the code. This program runs through the built-in tunes.

```
from microbit import *
import music
built_in_tunes = [music.DADADADUM, music.ENTERTAINER, music.PRELUDE,
music.ODE, music.NYAN, music.RINGTONE, music.FUNK, music.BLUES,
music.BIRTHDAY, music.WEDDING, music.FUNERAL, music.PUNCHLINE,
music.PYTHON, music.BADDY, music.CHASE, music.BA_DING,
music.WAWAWAWAA, music.JUMP_UP, music.JUMP_DOWN, music.POWER_UP,
music.POWER_DOWN]
while True:
    for tune in built_in_tunes:
        music.play(tune)
        sleep(5000)
```

Horrid Noise

This program lets the user press the red button and move the joystick on the y axis to control the pitch of the tone. It's pretty difficult to play as it is.

```
from microbit import *
import music


while True:
    # read y axis of joystick
    y = pin2.read_analog()+440
    btnred = pin12.read_digital()
    if btnred:
        music.pitch(y,-1)
    else:
        music.stop()
```

```
    print(y)
    sleep(10)
```

This can be improved a lot by choosing a range of frequencies that correspond to notes covering a smaller total range of notes. This could be a couple of octaves of the note from a key you want to play. This is a nice project to take on for a musical person. If you can get something that you can feel in control of enough to be able to play a simple tune, a good job has been done.

Simple Arpeggiator

This is a dodgy version of the concept but it might just be enough to give the more musical reader an idea. As with all my 'music' projects, my sincerest apologies go to those with musical talent.

Arpeggiators are tools built into synthesizers that allow the player to choose a group of notes by playing a chord. Those notes are then played back in some pleasing order.

This program was about getting the musical part just about right. There is a flaw with the input, mentioned later, that needs some thought to take the idea forward.

The **notes** list in the program is the frequency for every note from C0 to B8. C0 is at position 0, C4 is at 48.

The idea is to take some notes that sound pleasing when played together. A musical person suggested to me that I start with notes 1, 3 and 5 from a scale. The program plays them in a repeating cycle going up and down. The idea was to allow a user to vary the tempo and pitch of the note sequence. Varying the tempo just means changing the length of the notes - that's easy enough. Changing the pitch needed more thought.

When you play a sequence of notes, what makes them sound the way they do together is how far apart they are from one another in our list of notes. In scales, the notes are not evenly spaced. For example, the first and second notes of the major scale are 2 steps apart in our list. The third and fourth notes are only one step apart in our list. The way the frequencies are calculated for the list means that they are evenly spaced out. When we raise or lower the pitch, we need to preserve the spacing between the notes that we play. For a major scale, the notes we want are positioned at [0,4,7,4] relative to the base note (add these numbers to the base note to get the position in the list of the note we want). In this program, we start with a base note of middle C (48 in our list). Pressing the red and green buttons allows the pitch to be raised or lowered. This changes the value of the base note. When the sequence is played, the positions of the notes are worked out from the base note each time. That means that the sequence 'sounds the same but higher or lower'.

```
from microbit import *
import music

notes = [0x0010,0x0011,0x0012,0x0013,0x0015,0x0016,0x0017,0x0018,
        0x001A,0x001B,0x001D,0x001F,0x0021,0x0023,0x0025,0x0027,
        0x0029,0x002C,0x002E,0x0031,0x0034,0x0037,0x003A,0x003E,
        0x0041,0x0045,0x0049,0x004E,0x0052,0x0057,0x005C,0x0062,
        0x0068,0x006E,0x0075,0x007B,0x0083,0x008B,0x0093,0x009C,
        0x00A5,0x00AF,0x00B9,0x00C4,0x00D0,0x00DC,0x00E9,0x00F7,
        0x0106,0x0115,0x0126,0x0137,0x014A,0x015D,0x0172,0x0188,
        0x019F,0x01B8,0x01D2,0x01EE,0x020B,0x022A,0x024B,0x026E,
        0x0293,0x02BA,0x02E4,0x0310,0x033F,0x0370,0x03A4,0x03DC,
        0x0417,0x0455,0x0497,0x04DD,0x0527,0x0575,0x05C8,0x0620,
        0x067D,0x06E0,0x0749,0x07B8,0x082D,0x08A9,0x092D,0x09B9,
        0x0A4D,0x0AEA,0x0B90,0x0C40,0x0CFA,0x0DC0,0x0E91,0x0F6F,
        0x105A,0x1153,0x125B,0x1372,0x149A,0x15D4,0x1720,0x1880,
        0x19F5,0x1B80,0x1D23,0x1EDE]

# read the buttons and use binary 4 bits to represent
# the button states
def get_btns():
    pattern = 0
    for i,p in enumerate([pin12,pin15,pin14,pin16]):
        pattern += p.read_digital() << i
    return pattern


base = 48
# major
# pattern = [0,4,7,4]
# minor
pattern = [0,3,7,3]
# difference between highest note and base
limit = max(pattern)

delay = 100
last = 0

while True:
    b = get_btns()
    e = [(b >> i & 1) for i in range(4)]
    if e[0]: base += 1
    if e[1]: delay += 20
    if e[2]: base -= 1
    if e[3]: delay -= 20
    base = max(0, min(base, len(notes)-limit-1))
    delay = max(0, min(delay, 1000))
    arp = [notes[base+i] for i in pattern]
    for n in arp:
        music.pitch(n, delay)
        sleep(delay//10)
    last = b
```

The flaw in this program is that the timing of the button reading gives a variable experience to the user. If the speed is high, you can alter the pitch quickly. If it's low, it takes longer. This is because of the timing of the main loop and how little time is taken for the button readings compared to the note playing. There are a few solutions to this, as well as the prospect of a far grander project.

One way to deal with the button press delay is to 'manually' start and stop each pitch. That means removing the delays from the main loop and using running_time to determine whether or not to start or stop a tone. That is a bit trickier but it means that the buttons are polled more regularly and button presses are not missed.

A simpler, but less satisfying approach is to use the micro:bit buttons and the was_pressed() method. You can check for a button press after each note is played. Have two modes for your project. The user can be setting the speed and pitch or listening to the arpeggio.

Once the interactions of the main playing loop are cracked, the next step is to allow the user to input the notes that form the arpeggio by lighting dots on the matrix. The input would be a little abstract, with each dot being a semitone, but you could reliably choose a sequence of notes.

The order in which the notes are played can be varied to get different effects. The example program just goes up and down a sequence of 3 notes. You can play them in one straight order, or zigzag and jump about the list in more interesting ways. This could also be something that you could choose, perhaps whilst the arpeggio is playing. You could make it so that the arpeggio plays whilst one of the buttons is being held down. Program a different order for each button, for example, and you have a nice little item.