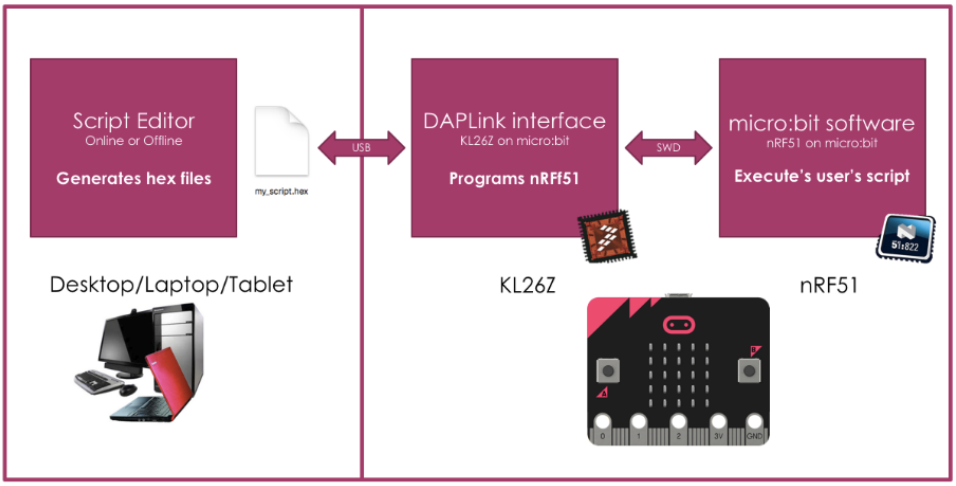
Software Design Process

Objective

Create a method to display student's code on a micro:bit onto a larger mega:bit for demonstration without affecting the micro:bit's performance. This must be done while taking into account the sensors and interactions with additional features such as radio communication. This has to be implemented in a manner such that any student's work can be displayed simply onto the mega:bit's LED matrix without need for additional coding or soldering from the teacher or student.

Design Context



Two software environments need be taken into account, the first being the editor that runs on the computer, and the second that runs on the micro:bit. The script editor that is online generates the hex files which are then uploaded to the micro:bit to be executed. The script editor is what has to be changed for the mega:bit for its functions such as being able to detect the presence of mega:bit once it is connected to the micro:bit.

Since the micro:bit can be programmed by several languages, such as Python, C++, Javascript and MakeCode Blocks in the online editor, our solution needs to take these into account.

Design Selection Process

1) Accessibility Pin: One of the GPIO pins (p12) is a marked as a ‘reserved accessibility pin’ which by default is not accessible to the user. This was created to leave the Micro:Bit Foundation the option for expansion modules. The theory was for any future peripherals which required data communication this pin would provide a UART connection for data transfer. The Mega:Bit suits this function perfectly.

2) Neopixels: These are addressable multicoloured LEDs that are supported by the micro:bit neopixel library. They can be attached simply with 3 crocodile clip connections for data input, power and ground. A large number of neopixels cannot be powered by the micro:bit and need to

be powered externally. The brightness of every LED can be adjusted independently in the software. These could be used for the mega:bit's display however they require strict timings which would require the use of interrupts. This has the potential to interfere with other low level activity on the micro:bit which also require strict timings and interrupts (for example the Bluetooth and RF radios).

3) I2C: The larger LED matrix could be controlled via I2C communication. An I2C LED display driver could be used to drive a matrix of LEDs which would be simple to implement as I2C is natively supported. I2C also has less strict timing requirements as the master can drive the clock at any frequency it would like (up to 400 kHz) in fact if the master has background tasks to do while transmitting it can 'stretch' the clock signal to greater allow for these. I2C bus utilisation must be investigated as we do not want to interfere with the other sensors on the I2C bus.

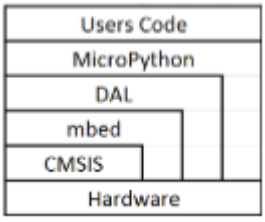
4) USB: Possibly provide two-way communication to control the mega:bit through the USB port or 'clone' the code onto the mega:bit. Two-way communication through the USB port is not feasible but cloning the code potentially is. The code is stored in a particular place in the on-chip memory which could be read and copied to the mega:bit. This however requires a level of 'intelligence' on the mega:bit side, for instance like a Raspberry Pi. This would obviously add cost and be more expensive than the alternative methods.

Execution of I2C Method

The I2C method was chosen as it provided the simplest and cleanest solution however the accessibility pin may still be used to provide a handshake to confirm when the micro:bit is plugged into the mega:bit.

Working with the Micro:bit runtime - Device Abstraction Layer (DAL)[1]

Several layers exist to translate the user code down to the hardware level. Each 'sits' on the top of the next and is procedurally stepped down to the hardware control. However, in some case, mbed, DAL, and microPython can access the hardware directly. We will concentrate on the online MakeCode editor which differs from the diagram above. Instead of the MicroPython layer there are two different layers connect the user code to the DAL which are called the 'pxt-microbit' and 'microbit' layers.



The micro:bit runtime or device abstraction layer (DAL) contains all the hardware drivers for the micro:bit such as display, radio etc and runtime mechanisms that allows flexible programming of the micro:bit. We will insert our code here as it will apply to all user code formats.

[1] "Lancaster-University/Microbit-Dal". *Github*, https://github.com/lancaster-university/microbit-dal/issues/256.

Micro:Bit I2C Functionality

The I2C communication standard allows for many I2C slaves. The micro:bit has two existing devices on the I2C bus, the compass and accelerometer, we must ensure our use of the bus does not affect the existing functionality. This is done by ensuring there are no addresses clashes with the existing sensors.

It is important to note that the 7-bit addresses found in the device datasheets have to translated with a binary shift left as an 8-bit address is to be used with the micro:bit. For example, the adafruit HT16K33 backpack has an address 0x70, that must be translated to 0xE0.

Driving LED Matrix using I2C

Components used: Adafruit 0.8" 8x8 Matrix, Backpack HT16K33 IC, micro:bit edge connector

An Adafruit backpack[2] was used for convenience as a comprehensive library exists. This means we can quickly and easily create a proof of concept before moving on to creating our own PCBs. The I2C pins of the micro:bit, pin 19 and 20 are accessed using an edge connector. The connections are made as follows:

Microbit Pin	Backpack Pin
Pin 19	SCA
Pin 20	SDA
GND	GND
3V	VCC+

The DAL (Device Abstraction Layer) code was modified in order to implement the communication with the I2C external LED matrix.

The listing below shows the initialisation of the global variables needed to set up the LED display. The general structure of the code to interface with the HT16K33 LED driver was translated to C++ from a Python library for the backpack[3].

[2] Microbit Playground. (2016). *Control a 8x8 LED matrix with an I2C backpack on the microbit*. [online] Available at: https://microbit-playground.co.uk/components/8x8-matrix-HT16K33-microbit.

[3] https://bitbucket.org/thesheep/microbit-ht16k33/src/a5b7961f0b57ba226ab98edc2c5f8d95d8954d00/ht16k33.py

```
//------------------------------------------------------------------------------------------------------------------------
#include "MicroBitI2C.h"
#include "MicroBitDisplay.h"

MicroBitI2C i2c(I2C_SDA0, I2C_SCL0);

char buffer[17] = {};
uint8_t address = 0xE0;
uint8_t _HT16K33_OSCILATOR_ON = 0x21;
uint8_t _HT16K33_BLINK_CMD = 0x80;
uint8_t _HT16K33_BLINK_DISPLAYON = 0x01;
uint8_t _HT16K33_CMD_BRIGHTNESS = 0xE0;
bool i2cPresent = false;
```

Configuring Display on LED Matrix

The listing below shows the initialisation process of the LED display. After having checked whether the I2C ports are being used (i.e. if a mega:bit is connected), the buffer is filled with zeroes, the oscillator is turned on and an initial blink rate and brightness are set.

```
class StartUp
{
public:
        StartUp()
        {

                if ( i2c.read(0xE0, 0, 1) == 0 ) { i2cPresent = true; }

                if (i2cPresent) {
                        buffer[0] = 0x00;
                        for (uint8_t i = 0; i < 16; i++) {
                                buffer[i + 1] = 0x00;
                        }
                        char tmp[1] = { _HT16K33_OSCILATOR_ON };
                        i2c.write(address, tmp, 1);
                        set_blink_rate(0);
                        set_Brightness(15);
                }
        }
};
```

Eight main display control functions have been implemented:

Function	Description
set_blink_rate	Sets the blink rate of the display
set_Brightness	Sets the brightness of the display
update_Brightness	Updates the brightness of the display to a new value
fill	Fills the display
initialise	Initialises the display (similar to the StartUp class above)
_pixel	Plots single pixels on the matrix
pixel	Converts x coordinates and calls _pixel function
bufferClear	Clears the buffer

Replicating All Functionality

The micro:bit display is updated to match a 'buffer' which stores the image/text to be displayed. It works on an interrupt basis and is updated several times per second. The I2C display holds the previous state so does not need to be constantly updated, this would also take up unnecessary processing power and bus utilisation. Therefore, we only update the I2C display whenever the buffer is modified, this is the most efficient way of implementing the code.

Further References:

"Microbit-RTCC-MCP7941X - Program To Interface Bbc Microbit To A MCP79410... | Mbed". *Os.Mbed.Com*, https://os.mbed.com/users/euxton/code/microbit-RTCC-MCP7941X/file/92208e3aae5a/main.cpp/.

"Driving Adafruit I2C 8X8 LED Matrices With The BBC Micro:Bit". *Smythe-Consulting.Com*, 2017, http://www.smythe-consulting.com/2017/03/driving-adafruit-i2c-8x8-led-matrices.html.