Assistive Technology with the BBC micro:bit

Helping people communicate

Matthew Oppenheim, InfoLab21, Lancaster University, matt.oppenheim@gmail.com

In this article I use the BBC micro:bit [microbit] to create two assistive technology (AT) devices. These are designed to enable people with cerebral palsy to more easily access communication software to create speech. The ideas for the projects came from the Technologists who work at Beaumont College, Lancaster. This college educates around 100 students with a variety of disabilities, many resulting from cerebral palsy. Some of the students use specialist communication software to create speech.

I will describe the micro:bit and the programming tools that are available for it and what makes this board a suitable choice for using in AT. The area of AT that these projects contribute to is called alternative and augmentative communication (AAC). The projects were tested at the College and have been presented at the Communication Matters conference. All code and manuals are freely available on my GitHub site. I created videos showing how to implement the systems.

While I was writing this article in October 2020 a version 2 of the micro:bit board was announced. The new board has a faster processor, more RAM, a speaker and a microphone, all in the same form factor. The projects presented in this article run on both versions of the board.

BBC micro:bit board

Since early 2016, every 11-12 year old in the United Kingdom is given a small embedded microcontroller board to learn how to program. The development of this board was driven by the British Broadcasting Co-operation (BBC) with 29 partners [BBC micro:bit]. Over five million of the boards have been manufactured.

The micro:bit packs a lot into a 4 x 5 cm board. Please see Table 1 for a summary of the hardware specifications for the micro:bit v1, which was used for development of the projects in this article. Figure 1 and Figure 2 show the front and back of the board along with a AAA battery holder.

The micro:bit lacks the 0.1 ” header pins found on most development boards as these might not last long in the hands of the intended user group. Instead, an edge connector allows access to power, ground, the I2C bus and the microcontroller general purpose input-outputs (GPIOs). A set of 4 mm holes along this connector allows banana plugs to connect with some of these signals and the voltage and ground. Crocodile clips can be used with the five larger pads.

A block diagram of the board is shown in Figure 3. This also shows the signals that are connected to the edge connector.

The microcontroller has a built in 12-bit ADC, which can be accessed from the edge connector. Break out connectors are available which the micro:bit slots in to, which enable easy access to all the edge connector pads.

The board is powered by 2AAA batteries which connect using a JST connector, or through the micro-USB connector. The boards are programmed through the same micro-USB connector.

The CPU on the micro:bit v1 is the Nordic nRF51822 with an ARM Cortex-M0 32 bit processor which contains a 2.4 GHz radio module. The radio module can be used for Bluetooth or with a custom radio stack. Only one of these wireless protocols can be used at a time though.

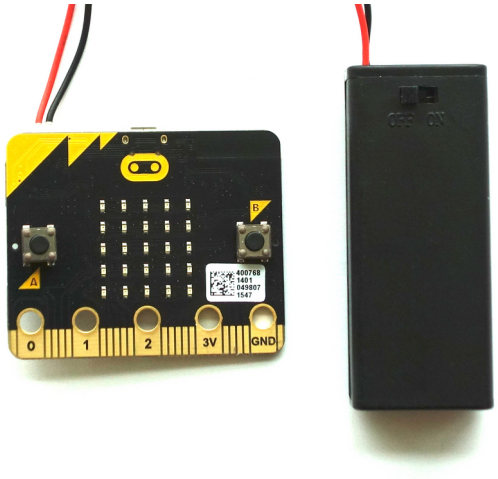

Figure 1: BBC micro:bit v1, front view

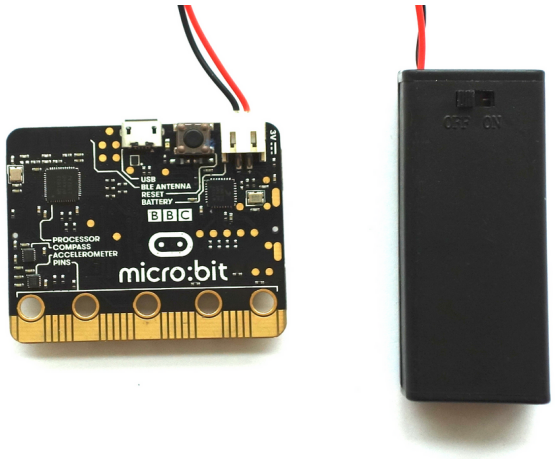

Figure 2: BBC micro:bit v1, back view

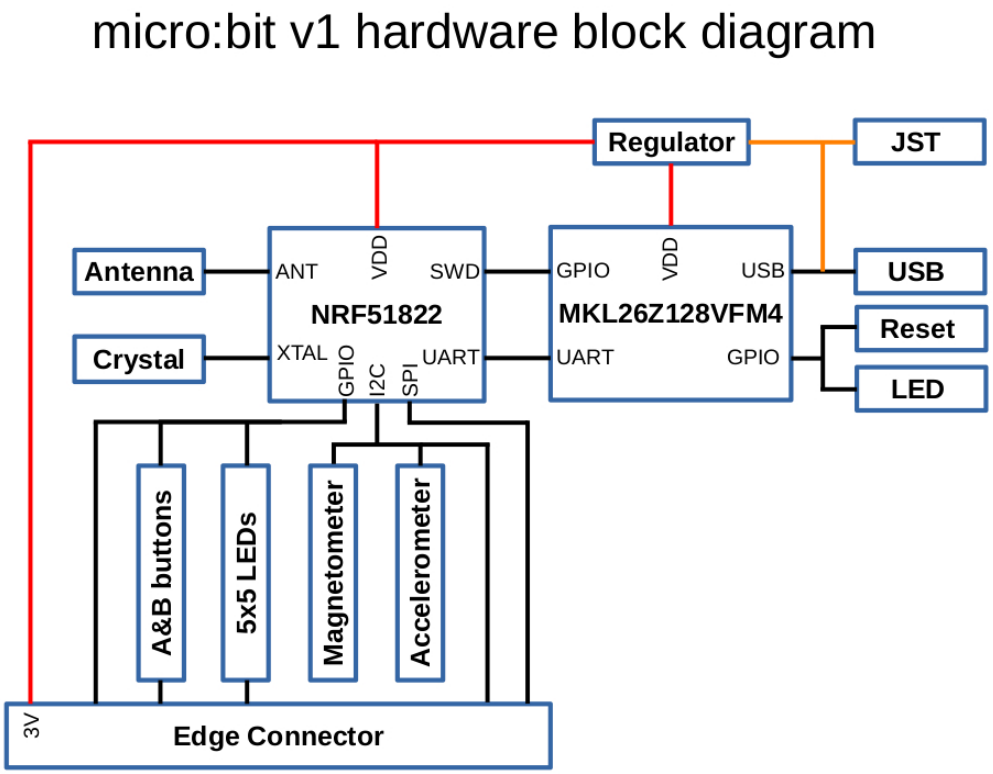

Figure 3: Block diagram of the micro:bit v1

Board size	4 cm x 5 cm, AAA battery pack 6.5 cm x 2 cm
Weight	Board 8g, batteries 36g
Power source	2 x AAA batteries or USB
Connectors	1 x micro USB, 1 x JST battery connector, 23 pin edge connector (6 can be used as analog, 19 can be used as digital I/O)
Digital/analog interfaces	1xSPI, 1xI2C, 3xPWM, 1xTX, 1xRX, 3 x touch sensors, 2 x button detection
Processor	Nordic nRF51822-QFAA-R rev 3 ARM Cortex-M0 32 bit processor, 256 KB flash ROM, 16 KB RAM, 16 MHz
Accelerometer	3-axis Freescale MMA8652, 800 Hz, 10 bit, 2/4/8g ranges
Magnetometer	3-axis Freescale MAG3110, 80 Hz, 0.10uT sensitivity
Temperature	Using the on-core nRF51, 0.25° C sensitivity

User interfaces	2 x user push buttons, 1 x system button, 5 x 5 LED grid, capacitive sense pins via edge connector
Wireless	2.4 GHz Bluetooth 4.1 with Bluetooth low energy with 40 channels or proprietary Nordic Gazell protocol offering 101 channels and 255 groups at up to 2 Mbps.

Table 1: BBC micro:bit v1 specifications

The board contains an accelerometer and a magnetometer. External temperature can be measured using the temperature sensor on the nRF41. The nRF41 runs so cool, that this temperature sensor can be used to measure the environmental temperature instead. The LEDs can be used to measure external light levels. Light induces a current in the LEDs when they are not active. This current can be measured to infer the external light level.

For my AT projects the micro:bit has several features that make this board a suitable choice:

Designed to be safe

As these boards are designed to be safe to give to 11-12 year old students, they are safe to distribute in the ‘real-world’, compared with handing out home-made devices.

Somebody else makes them

If a suitable product already exists, then the fastest way to deploy a system for real-world use is often to repurpose the existing product. I like hunching, Golem like, over a home-made circuit board, enveloped by toxic solder fumes. However, the goal of these projects is to get something that is fit for purpose and can be easily distributed and implemented.

Robust board to board radio

In use, I found that the custom radio stack available on the micro:bit reminded me of using the XBee nodes that I wrote about in my last Circuit Cellar article a few years ago in issue 319. The custom radio stack works without handshaking and in use proved to be reliable. The alternative wireless protocol available is to use the radio module in Bluetooth mode. I find Bluetooth a fickle beast. Sometimes the target node needs resetting to become visible to the master node, which is not a practical thing to ask the target user group of these projects to do.

The micro:bit’s custom radio stack can be configured to send messages with both an address and a group, reducing the potential for interference from other micro:bits. The data transmission rate can be set up to 2 Mb a second.

Good programming tools

Several programming platforms are freely available for the micro:bit. As the board is aimed at school use, the programming tools are designed to be easy to use.

Software tools

I used the micro:bit implementation of MicroPython [MicroPython] for the two projects presented here. There are several other programming languages and platforms available.

I had a play with the Code Blocks block-based drag and drop programming interface. Code Blocks is used at schools to start students with programming. The programming environment shows the JavaScript code that the Blocks produce. JavaScript can also be used to directly program the boards.

Mbed provides an online C compiler for the micro:bit v1, but not for v2 of the board. Lancaster University also developed a C compiler for the micro:bit which the micro:bit's MicroPython implementation and Code Blocks are built on top of. The Lancaster University C compiler can be integrated into e.g. the Eclipse interactive development environment, but I prefer using it from the command line with a Makefile. The Lancaster University compiler is compatible with both versions of the micro:bit.

Connecting a micro:bit to your PC creates A folder called 'MICROBIT'. If you use one of the editors aimed at the micro:bit, such as the mu editor, then your MicroPython code can be deployed with a button click. I tend to use a text editor for coding. To deploy my MicroPython code to the micro:bit I use the the uflash command line utility [uflash]. I created a bash script to automate running this tool each time I save the file that I am working on. I put details of this script on my web site [uflash].

In common with many people who program embedded hardware, I often use C for my projects. Using MicroPython generally enables faster development and makes the code easily accessible for others to build on. However, there are limitations with using MicroPython compared with C.

The micro:bit implementation of MicroPython lacks access to event handling which means my MicroPython code relies on manually checking for changes of state during each iteration of a while-loop. This is not a limitation with the C-compiler, which implements access to events.

There is no implementation of Bluetooth in micro:bit's MicroPython due to memory constraints. For this project, not having access to the Bluetooth functionality was not an issue. The board to board custom radio stack proved to be robust and reliable.

Software Limitations

I ran out of memory using MicroPython for another project when I structured the code using classes. I got the project to run within the memory limitations by discarding the classes and just using methods. This memory constraint, lack of event handling and the loss of Bluetooth all point me to using the Lancaster University C compiler for more advanced projects. As v2 of the micro:bit has 128 Kb of RAM compared with the 16 Kb of RAM in the v1 boards, the memory constraints for writing in classes should be removed. However, I prefer writing event driven code. I would like to see the assert statement included in the micro:bit's MicroPython implementation to help with writing tests.

Project 1: Give Me a Minute - visualising speech preparation

For many high-tech AAC users, it is difficult to see when they are composing a message. For instance, it may not be clear when people who use eye tracking technology are actively preparing speech.

We implemented a system that visually indicates when communication software is in use. This enables a more natural interaction and encourages good communication practice; giving adequate time for composition and respecting personal space. The visual feedback reassures others in the conversation that the AAC user is actively involved.

The system continually monitors the AAC software to show when new messages are being entered, without the user having to manually trigger a "hang on" type message and interrupt their composition. This monitoring is done with a Python script called activity_indicator.py. I used pyinstaller to generate an executable called activity_indicator.exe which can be run without having to install Python.

The activity_indicator script finds and monitors the window that is running the communication software. When the script detects that new text has been entered, or existing text deleted, a signal is sent to a micro:bit through the USB port. The micro:bit then displays an animated pattern over about one second to indicate that the communication software is in active use. The activation threshold is adjustable, to cater for different software and screens sizes.

Monitoring communication software

The win32gui library is used to find the window that runs the communication software. In testing, I found that one communications package I wanted to monitor spawned more than a single window, although only one was visible. Luckily the invisible window has a slightly different title from the window that needs monitoring. I added a list of window titles to ignore to filter out the invisible window. The Python method for finding the window that has the communication software running inside of it is shown in Listing 1. This method uses the win32gui library to interact with the Windows operating system and get a list of windows and their titles.

```
import win32gui

COM_SOFTWARE = ['grid', 'communicator']

IGNORE = ['grid 3.exe', 'users']


def find_window_handle(com_software=COM_SOFTWARE, ignore=IGNORE):
    ''' Find the window for communication software. '''
    toplist, winlist = [], []


    def _enum_cb(window_handle, results):
        winlist.append((window_handle, win32gui.GetWindowText(window_handle)))
```

```
    win32gui.EnumWindows(_enum_cb, toplist)
    for sware in com_software:
        # winlist is a list of tuples (window_id, window title)
        for window_handle, title in winlist:
            if sware in title.lower() and not any (x in title.lower() for x in
                  ignore):
                return window_handle
    logging.info('no communications software found for {}'.format(com_software))
    time.sleep(0.5)
```

Listing 1: Method for locating the window containing the communication software using win32gui

Typically, only the top 20% of the window is used to display the text being prepared for speech. By monitoring only this part of the window, the amount of resources used by the software is reduced. The Pillow image processing library is used to count how many black pixels are in the top 20% of the communication software window, twice a second.

Making the micro:bit hot swappable

Once I tested a prototype with the target user group, I realised that I had missed important use cases. What happens if the software is started without the micro:bit attached? Or what happens if the micro:bit is removed during use and then re-connected?

I needed to make the micro:bit “hot swappable”.

I wrote a custom class for setting up and tearing down the serial connection with the micro:bit. I instantiate this class in a context manager. Using a context manager means that however the serial connection is terminated, the __exit__ method of the class runs. This ensures that if the micro:bit is unplugged during use, the serial connection is cleanly closed down by the __exit__ method.

To wrap the serial port in a context manager, the serial connection is instantiated using the keyword 'with', in the line:

```
with Serial_Con(mbit_port) as mbit_serial:
```

The main method from the software that runs on the communication software, called activity_indicator.py, is presented as Listing 2.

I use the Singleton pattern to ensure that only one instantiation of the serial interface exists. This may be overly cautious, but it prevents any possibility of multiple connections being created if e.g. a second micro:bit is plugged in. I learned how to implement the singleton pattern the usual way, by reading answers in Stackoverflow. I also contribute to Stackoverflow, so I try to pay back some of the help I receive. I use the method that implements the Singleton pattern as a decorator for the serial connection class.

If the micro:bit is removed during use, then an exception is generated when the program tries to send a message to the micro:bit over the now non-existent serial connection. This exception is caught and the inner loop in the main method exits. The outer loop then re-establishes the connection to the micro:bit when the micro:bit is re-connected.

```
def main(limit, fraction):
    logging.info('*** starting find_microbit ***\n')
    check_fraction(fraction)
    logging.info('limit={} fraction={}\n'.format(limit, fraction))
    old_black = 0
    while True:
        logging.info('*** looking for a microbit')
        mbit_port = get_comport(PID_MICROBIT, VID_MICROBIT, 115200)
        logging.info('microbit found at comport: {}'.format(mbit_port))
        with Serial_Con(mbit_port) as mbit_serial:
            # occasionally mbit_serial is not created, so is None
            if not mbit_serial:
                logging.info('failed to create mbit_serial')
                time.sleep(0.5)
                continue
            logging.info('microbit serial port created at:
{}'.format(mbit_port))
            while True:
                time.sleep(0.5)
                # look for the top fraction of a window running the target
software
                window_top = get_window_top(fraction)
                if window_top is None:
                    continue
                # count black pixels in top fraction of target window
                new_black = num_new_black_pixels(window_top)
                logging.debug('new_black: {}'.format(new_black))
                if new_black is None:
                    continue
                is_limit_exceeded = check_limit(new_black, old_black, limit)
                if is_limit_exceeded:
                    try:
                        mbit_serial.write(b'flash')
```

```
                    except serial.SerialException as e:
                        logging.info('connection to microbit failed
{}'.format(e))
                        break
                old_black = new_black
```

Listing 2: Main method for monitoring communication software and communicating with the micro:bit

Figure 4 shows Give Me a Minute being tested at Beaumont College, Lancaster, UK. The tester was so pleased with the system that she took it away with her - which I had not planned for!

Figure 4: Give Me a Minute in use at Beaumont College, Lancaster, UK

Project 2: Handshake - using gesture to control software

Handshake is designed to enable people who cannot use physical controllers such as buttons or joysticks, but who can make an intentional hand or arm movement, to interact with switchable AAC software.

Regular readers might remember an article I wrote in issue 319 where I presented a prototype of this system [Oppenheim Full-Stack Python]. For that project I used several boards to achieve what I do here with a single micro:bit. The micro:bit replaces having to interface a microcontroller board with an accelerometer board, battery management board and an XBee wireless communication board.

Handshake consists of two micro:bit boards programmed using MicroPython. One micro:bit is configured as a transmitter, which is worn on the wrist – see Figure 5. The second is the receiver,

which is attached to the communication device running the AAC software with a micro USB cable. Software written in Python is installed on the AAC device.

The acceleration that the student’s hand is moving with is constantly monitored using the micro:bit on the wrist. When the acceleration exceeds an adjustable limit, a key press command is sent to the student’s AAC software to control it.

The threshold of acceleration at which the system triggers can be adjusted using the buttons on either of the micro:bit boards. This allows the system to be adjusted to cater for both gentle and energetic motion characteristics. As either of the micro:bits can be used to adjust the threshold, the person wearing the transmitter does not have to be disturbed.

The micro:bits have a simple operating system on them which allow persistent files to be saved. I leveraged this to save the shake detection threshold to a file. This value is loaded by the software running on the transmitter when it is turned on.

A simple orientation invariant acceleration threshold recognition algorithm is implemented on the transmitter. As the algorithm is orientation invariant, the system is not restricted in use to when the student is upright. For example, the system can be used when laying down in bed, perhaps to turn on a light or to signal for attention.

Initially, the transmitter micro:bit was worn on the wrist using a converted smart phone holder arm band. Now I use an off-the-shelf iPod Classic armband, so I don't have to do any sewing. The only addition to the armband is to add some tabs of back to back Velcro 'gender-bender' to enable the strap to stick down when worn on the wrist, as the straps are long enough to go around an arm.

Initial testing at Beaumont College used the system to turn on an LED using two volunteers from the target user group. A photo of the testing is shown in Figure 6.

The student that the system was developed for managed to improve his co-ordination so that he is able to use contact devices, such as switches. This means that he does not need to use HandShake. Beaumont College is looking to see if any of their other students could benefit from the system. There are some other organisations interested in testing the system out. This is all made a bit difficult with the ongoing Covid pandemic at the time of writing this article.

Having a system that gives a visible signal in response to an intentional arm movement is of use in itself. The system can be used to help assess if students are aware that their arm motion causes an external change.

I tested the battery life by simulating one shake detection per second. The battery life exceeds 24 hours using rechargeable batteries.

The Python software on the AAC implements the same 'hot-swappable' algorithm as the Give Me a Minute software. This means that the receiver micro:bit can be unplugged and replaced without crashing the system.

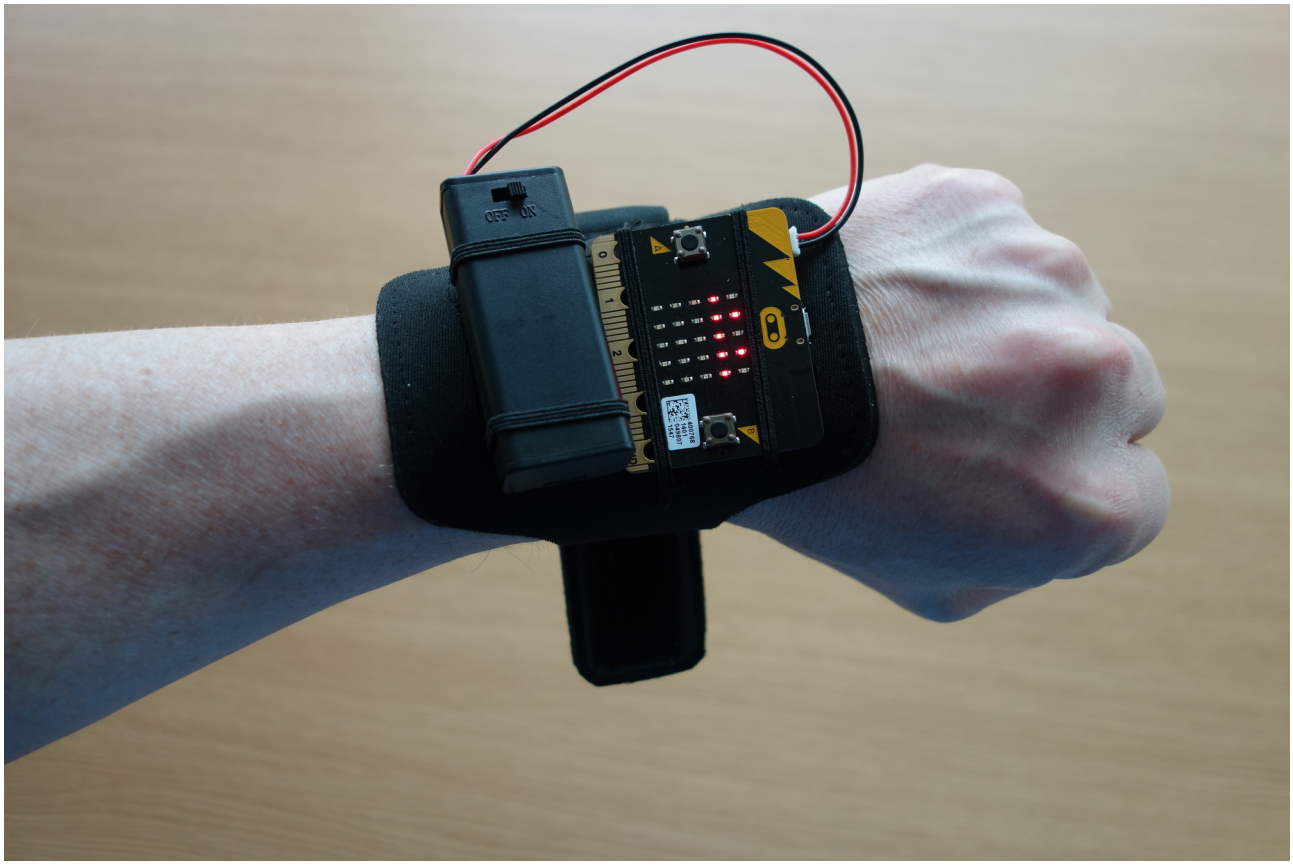

Figure 5: BBC micro:bit worn on the wrist as part of the Handshake system

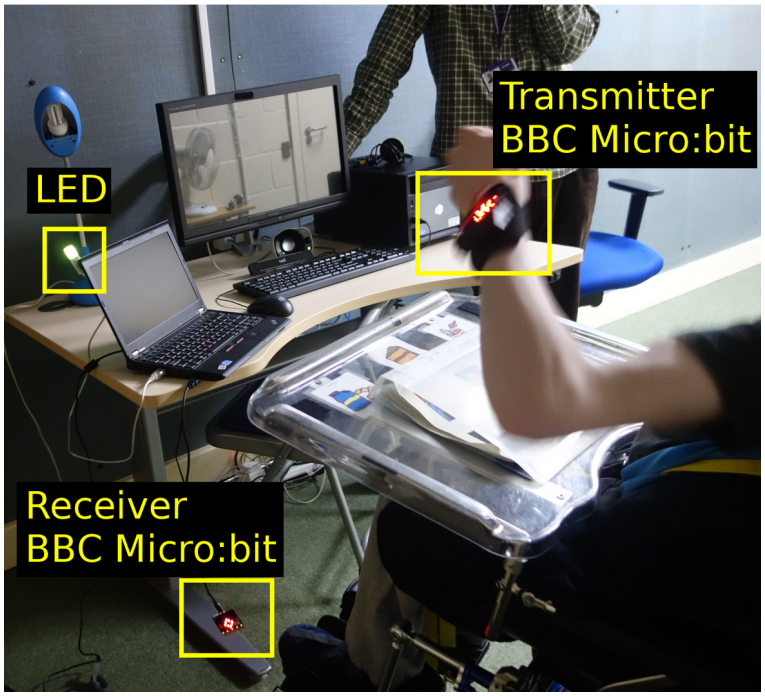

Figure 6: Initial handshake testing, using the system to turn on an LED

Discussion

The micro:bit is a versatile platform that lends itself to real-world applications such at the ones presented in this article. V2 of the board has a few new features that could be of use in the AT field, such as a speaker and a microphone.

I set up Git repositories for my software development and recommend the system. If I want to explore a new idea, I create a new branch for the code. If the idea works out, I merge this branch into the main branch, which I then push to my GitHub site. Using Git helps prevent losing ideas I had in earlier versions of code, then deleted. I can roll back any part of the code base to an earlier saved version.

I put full documentation onto my website and produced 'how-to' videos for both projects, as well as a short video explaining the projects for the TechAbility assistive technology conference. I am working on putting all of the documentation onto GitHub using the mkdocs tool. This makes the documentation open source as well as the code.

If anybody would like to implement either of the projects presented here or has ideas on how to improve them, please get in touch.

Resources

Videos and links to the project software and documentation can be found on the project website [Oppenheim website].

[BBC micro:bit]. Retrieved March 5, 2018, from https://www.microbit.co.uk/home.

[Oppenheim website] https://www.seismicmatt.com/handshake/

[Oppenheim Full-Stack Python] Circuit Cellar issue 319. February 2017.

[MicroPython] https://tech.microbit.org/software/micropython/

[uflash] https://www.seismicmatt.com/2020/08/23/automating-loading-micropython-code-to-the-bbc-microbit-in-linux/