Power Consumption Analysis of the BBC micro:bit

Matthew Frost

January 17, 2018

Abstract

This report analyses the power consumption of three APIs for the micro:bit by developing and executing a series of tasks that test the main components of the micro:bit.

The aim of this report is to discover areas of improvement for particular software components, and to benchmark the performance of the hardware when used by each language in terms of power consumption, for future product and language development.

This report concludes that the high level nature of some languages that support the micro:bit come at a cost of high power consumption. Moreover, suggesting some improvements and considerations that could be made for the languages under test and specific power 'power-hungry' hardware.

Introduction

The micro:bit is an embedded system primarily used for computer education. Since its launch in March 2016, it has seen global success and has recently launched in India. A key aspect of an educational tools success is it's ease of use, and as the micro:bit was originally targeted at school children in the UK, this aspect is more significant than ever.

An element that indirectly impacts ease of use is power consumption. Although the micro:bit can be mains powered via USB or pins, it can often be used in projects where electricity is more finite, such as a radio controlled car making use of the stock AAA battery pack. Alternatively, the micro:bit can be powered by solar, which although not finite, desires a more cautions power consumption.

Often, high power consumption can generally be attributed to inefficient programming by developers, who can hide behind the blessings of Moore's law. However, for devices with with a finite power source this is often an area of focus and cannot be overlooked.

This report abstracts away from the 'lazy developer', as this is an inevitability given the micro:bits average use case (education), and can only be compensated by low powered; hardware, drivers (microbit-dal) and runtime environments including interactive prompts (MicroPythons REPL). Therefore, those aspects become the primary focus of this report, comparing PXT (makecode), C/C++(micro:bit runtime) and MicroPython, by conducting a variety of tasks interacting with the hardware.

PXT provides the highest level of interaction for programming the micro:bit using a drag and drop block interface. Blocks are then converted into C++ and use drivers provided by the micro:bit runtime to create a HEX file to be flashed. Therefore, the power consumption should be similar to that of 'pure C++' for the tasks undertaken.

The micro:bit runtime provides the lowest level of interaction for programming the micro:bit (beyond Assembler) in C/C++. The micro:bit runtime also supports most of the higher level micro:bit languages such as Microsoft Touch Develop, JavaScript and as previously mentioned, PXT.

MicroPython, mostly has it's own implementations of drivers but for some component interaction it uses the micro:bit DAL.

1 Methodology and Setup

In order to test each software component effectively, a series of common tasks must be developed that can both isolate and capture the power consumption for a given software component. The outcome of each task should be a single reading showing the power consumption in milliamps (mA). Multiple tasks may have to be produced in order to ensure data validity.

Figure 1 illustrates the setup of the power consumption test apparatus, the state of which remains constant throughout.

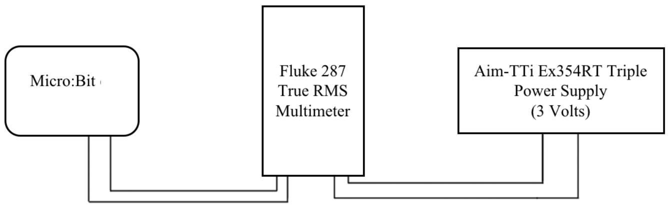

Micro:Bit
Fluke 287 True RMS Multimeter
Aim-TTi Ex354RT Triple Power Supply (3 Volts)

Figure 1: Test Setup

The specific tasks created can be found in their corresponding sections along with details of why they are chosen, how they are implemented and any revised setup aspects. Each component is also accompanied by results of the tasks and analysis of any interesting results found.

All code is open source, commented and structured such that each task is an individual program, assisting in the isolation aspect of the testing and reusability. In the interest of isolation, an additional 'flavour' of the micro:bit runtime can be tested containing the minimal device drivers needed to perform a test. This 'lite' version of the runtime will be documented for each test by noting the software components removed. This is done by removing the initialisation of the objects in *Microbit.cpp* and declaration in *Microbit.h*. An example would be a program performing just arithmetic operations, which would not have components such as the radio and display initialised. For ease of documentation this will be described as its own language named 'micro:bit Lite'.

Algorithms chosen to test a given component will evidently vary in syntax for each language, however, it will functionally and structurally attempt to be identical.

Due to the CPU and other hardware components running slightly more power efficient after the initial boot, the multimeter current reading takes approximately 5 minutes to converge for most tasks. Therefore, the duration of each test is 5 minutes with readings taken in 30 second intervals from which an average in calculated (mean). This mean value is then used as a comparison against other languages. The measuring process may have to be adapted for some task that provide a 'noisy reading' by using the inbuilt *MIN/MAX* function on the multimeter, which provides an average along with a minimum and maximum value.

2 Testing Specifications

See below a list of the specifications of the equipment used for each test. Detailed specifications of each individual test can be found in their corresponding sections.

- MacBook Pro (Retina, 15-inch, Mid 2014) 2.8 GHz Intel Core i7
- VM running Ubuntu 16.04 64-bit (virtualBox)
- Fluke 287 True RMS Multimeter
- Aim-TTi Ex354RT Triple Power Supply (3 Volts)
- Stock micro:bit USB cable (5V 100)
- Test Leads CAT I, $1mm^2$ conductor size
- micro:bit Firmware 0234
- uFlash 1.0.8

The build environment used can be downloaded from:

www.github.com/carlosperate/microbit-dev-env

3 CPU

The CPUs power consumption can be defined as:

$P_{CPU} = P_{static} + P_{short-circuit} + P_{transition}$

P_{static} is sub-threshold conduction and tunnelling current, both of which are the main causes of leakage within a CPU. Tunnelling power dissipation is an ongoing issue in silicon chip development with the desire to make them as small as possible. However, this report will not investigate this as a factor as it is just a function of the supply voltage.

$P_{short-circuit}$ and $P_{transition}$ are forms of dynamic dissipation which is a function of the *activity* being undertaken. Therefore, the *activity* becomes an area of interest for the testing of this component and the independent variable which is the case for most of the tests in this report.

Three tasks can be undertaken for the testing of this component; idle mode (sleep), infinite while loop and generating Fibonacci numbers. The idle mode consists of putting the micro:bit into sleep in its respective language. Using the DAL this is simply the *uBit.sleep()* function which deschedules the current fiber and perform a power efficient concurrent sleep operation. The same function is also used by PXT via the *pause()* function, whereas MicroPython uses it's own *sleep()* function.

The infinite while loop is simply a while loop containing a small amount of arithmetic operations that it does so continuously, for the purpose of wasting CPU cycles.

A recursive Fibonacci algorithm will allow the testing of the CPUs power consumption when having to perform memory operations. Due to MicroPython occupying a large amount of RAM it leaves a very small stack space. As a result, the Fibonacci algorithm is only able to generate up to the 6th sequence of the series. Therefore, all other environments also have to be tested to the same value

Finding the Maximim Fibonacci Sequence

Other languages, however, are able to generate a greater number in the series because of the increase in the available RAM size. An interesting comparison can be found by removing the Bluetooth stack, increasing the amount of available SRAM. The removal of the stack is done by setting the *MICROBIT_BLE_ENABLED* constant to *0.*

The DAL with and without the bluetooth stack have to be stoped after generating the 43rd number in the sequence due to the algorithms exponential time complexity, making the test an unfeasible duration. To estimate what number would cause a crash the following pseudocode can be used:

```
void fakeFibonacci(int x){
        serial.send(++x, SPINWAIT);
        fakeFibonacci(x);
}
```

Regardless of the value of x this function recursively calls itself, incrementing and passing x on each call, acting as a stack frame counter. The serial sending should also guarantee that the value has been sent before proceeding with the recursive call, hence *SPINWAIT.*

In order to find the maximum amount of stack frames the detection of a crash needs to be established. A stack overflow is likely to occur when trying to find this value which should be indicated by the display showing "[$sadface$]030" meaning a corruption has been detected in the heap. However, due to the micro:bit having no dedicated MMU this may go undetected, so the results should only be classed as valid when "[$sadface$]030" is displayed. A series of tests demonstrated that providing only a raw integer with no casing or appending of a character to the serial send function produces a valid result most of the time, as shown in the pseudocode below.

Passing in the value 0 to the function when the stack is removed returns 262 as the last valid value received. When the stack is included this value drops to 255 as expected. These values do not translate directly to the recursive Fibonacci algorithm as it is likely to require larger stack frames because of its two parameters and more operations to store.

Given that C++ and most languages are evaluated left to right, the first recursive call of fibonacci will be fully evaluated before the right most, as shown below:

$\texttt{return}\ \underline{\text{fibonacci(x-1)}} + \text{fibonacci(x-2);}$

As a result, given a number x the function will have a maximum recursion depth count of x-1, as the first entry is non-recursive. Therefore, the approximate maximum Fibonacci sequence that can be found with and without the bluetooth stack enabled is 256 and 263 respectively. Practically, these sequences are not achievable as the value returned would far exceed any standard number type, meaning a custom type would have to be defined, requiring more RAM and then be included in the function parameters further increasing the size of a stack frame.

Results

Figure 2 shows the results of these tasks, with the DAL Lite initialising just the scheduler and message bus.

Excluding the DAL Lite, all languages perform almost identically from one another. PXT and DAL are expected to perform the same because of PXTs use of the DAL, however, MicroPython seems to use an almost identical process to perform it's sleep. DAL Lite outperforms all other environments in all tasks, with on average using 170% less power when in sleep mode.

The DAL with all components initialised has the highest power consumption but not as significantly so, when compared to MicroPython and PXT.

It can also be said that C++ based languages perform better at Fibonacci, this is most likely due to it 'running closer to the hardware' with less layers

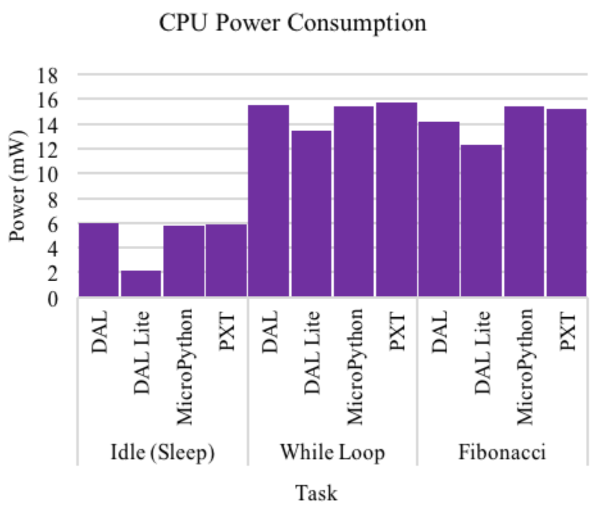


Figure 2: CPU Power Consumption

of abstraction, potentially allowing for more efficient operations. One consideration could be to use tail recursion optimisation which can implemented by the GCC compiler using a flag.

4 Accelerometer

The micro:bit makes use of the MMA8653FC Accelerometer. This component is separate from the CPU and connected via the internal BUS of the micro:bit. Communication is performed by I2C, which is isolated and tested in Section 7. The common feature used in an Accelerometer API is to retrieve the cartesian coordinates, the process of doing so is illustrated in Figure 3.

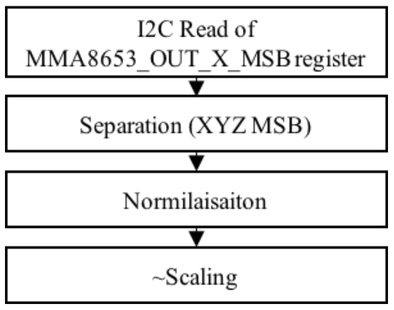


Figure 3: Accelerometer Reading Process

Fortunately, this process is repeated every time a request is made from the API to get the X, Y or Z position, creating a convenient layer of abstraction. For the testing of this component three tasks can be undertaken; the reading of all three positions in a while loop each of which are three separate API calls, an individual read of the X position in a while loop and a single call to retrieve the x position (X Isolated (Max)).

One purpose of conducting two while loop tasks is to discover if it is worth including a *readXYZ()* function as a *readX()* (or X or Z) already captures this data in the process shown in Figure 3, potentially having power consumption benefits.

The reasoning for testing a single call to retrieve the x position is to activate the accelerometer, which will cause continuous updates at the sample rate in the background using an idle thread. Because of the sampling period being so small, it becomes hard to record this data in 30 second intervals (noise), as a result the MIN/MAX feature can be used, with the average, minimum and maximum value recorded. The MIN/MAX feature should be used after 5 minutes of the micro:bit being active to allow the thermal effects to pass.

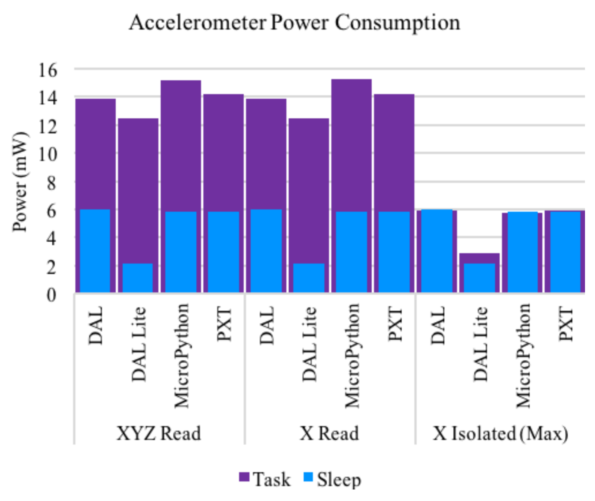


Figure 4: Accelerometer Power Consumption

Results

Figure 4 shows the results of the Accelerometer tests overlaid with the Idle (sleep) data shown in Figure 2. The the sleep overlay provides an insight into the additional power the micro:bit is using due to the task. Despite the retrieving of the X,Y and Z individually is has almost the equivalent power consumption usage as just getting one position (X Read), meaning there is no real need for a *readXYZ()* function.

To analyse the tasks power consumption further, let the purple area of the bar chart be denoted as $P_{component} = P_{task} - P_{idle}$. The environment which uses the least $P_{component}$ is PXT for both tasks, potentially meaning that it is more efficient at reading the accelerometer. Alternatively, it's idle state is poor and may be running part of the processes already in place to run the accelerometer.

Task	Environment	$P_{component}$
XYZ Read	DAL	7.90
	DAL Lite	10.29
	MicroPython	9.39
	PXT	8.37
X Read	DAL	7.92
	DAL Lite	10.27
	MicroPython	9.40
	PXT	8.34
X Isolated (Max)	DAL	-0.072
	DAL Lite	0.725
	MicroPython	-0.022
	PXT	0.054

Table 1: $P_{component}$ Accelerometer

Table 1 breaks down $P_{component}$ reaffirming that there is no need for an *readXYZ()* function. Despite MicroPython having a comparatively poor idle state, when looking at the DAL Lite, it has one of the highest $P_{component}$ values. This is primarily due to MicroPython creating it's own runtime environment as well as it's high level abstractions, which come at a cost of power consumption.

It is also worth noting that the DAL Lite had the scheduler, I2C, message bus and accelerometer enabled.

X isolated demonstrates that the activation of the accelerometer uses very little power consumption on average. The maximum value should be used (peak) as the power consumption measurement, as this is most likely when the accelerometer is being updated. The $P_{component}$ values for X under isolation are marginal and should not be treated as significant, except for DAL Lite. Because of the 'bare-bones' nature of DAL Lite it reduces the noise in the data compared to other languages, giving a true reflection of the components power consumption. Part of the issue with testing the *X Read* in a while loop is the drowning of the CPU with such commands, increasing the power consumption, however, this is a common use case for such a function and should therefore also be tested.

5 Display

The display of the micro:bit is a 5x5 surface mount LED matrix, making it the second most 'power-hungry' component(s) on the board. The LED matrix is also capable of becoming a light sensor, as LEDs can also be used as a photodiode by changing the direction of the current flow. To reduce the amount power consumed the display strobes each LED at a flicker fusion rate such that it goes undetected to the human eye.

Three tasks can be undertaken to test this component; all LEDs on (All HIGH), a single LED on (Single HIGH) and calling the light sense function in a while loop (Sensing).

Results

Light sensing is not available in MicroPython and is therefore not included in the results.

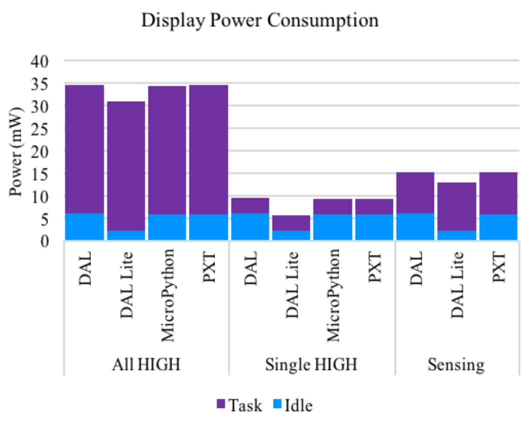


Figure 5: Display Power Consumption

Figure 6 shows the results with the idle (sleep) power consumption overlaid. As expected DAL Lite outperforms all other environments in each test, as it only has the display, message bus and scheduler initialised.

The performance of the other three environments demonstrates that there is no advantageous aspect to their display usage.

The standard deviation of all four environments $P_{component}$'s when ALL HIGH is 0.08, likewise, when a single LED is high, the standard deviation is 0.09. Because of the small amount of deviation the cost of a single LED could potentially be calculated.

The average $P_{component}$ for *All HIGH* is:

$28.635mW$
$28.635/25 = 1.1454mW$ per LED

The average $P_{component}$ for *Single HIGH* is $3.416mW$ which demonstrates that the workings of the display are as not as elementary as they appear.

The standard deviation between the remaining three languages for the light sensing is 0.88, again demonstrating that there is little discrepancy between their resource usage.

6 Magnetometer

The magnetometer, like the accelerometer is linked to the I2C Bus. The magnetometer is accessible through all the languages via a Compass API. The compass makes use of the accelerometer to reduce inaccuracies and will therefore be included in the DAL Lite along with the storage, I2C, message bus, accelerometer, compass and the compass calibrator.

The retrieving of the magnetic field strength both with and without a while loop can be tested, as the component works in a similar data push nature to the accelerometer. The remaining software components used by the compass are tested in previous or subsequent sections.

Due to the sensitivity of the magnetometer and physical environment that the test is conducted in, the methodology described in Section 1 where data is recorded in 30 second intervals can no longer be applied. This is due to electromagnetic noise in the environment causing the multimeters reading to become sporadic, as a result the *MIN/MAX* feature should be used.

Results

Figure 6 illustrates this reading. The field strength in isolation seems to show that MicroPython may not perform data push as it is $0.001mA$ different from its Idle (sleep) state, however, it may also be that MicroPython is more efficient at this task. Although, it could be expected that the magnetometer would operate at the same power consumption level as the accelerometer due to its similar underlining operations, the compass also requires the accelerometer and the compass calibrator.

The order of power consumption efficiency is the same as the accelerometer, which is understandable given that both are preforming similar operations to ascertain a value via I2C. However, the magnetometer as a hardware component does seem to require more power consumption.

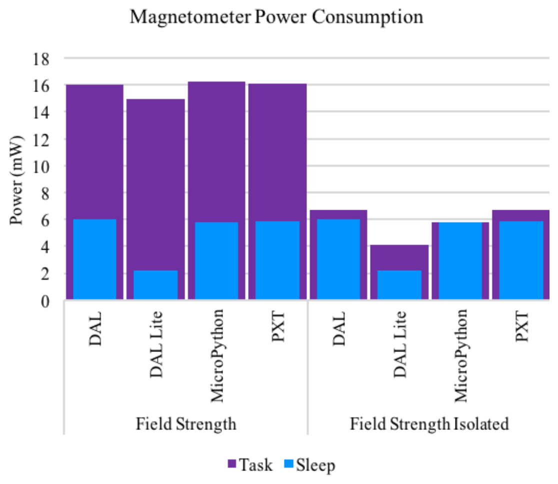


Figure 6: Magnetometer Power Consumption

7 I2C

I2C is a short distance master-slave(s) communication protocol often used for intra-board communication between the CPU and low speed peripherals, such as EEPROM (persistent storage). I2C is lightweight in the sense that it need not concern itself with security nor error checking.

I2C across all languages consists of a read and write function which can be tested, with some allowing more parameters by function overloading. The APIs allow configuration of the SDA and the SCL which by most users of the micro:bit will be set to pins, however, this testing uses the accelerometer, meaning that no pins are specified.

Addressing Schemes

It is worth noting the difference in addressing schemes across the languages. The DAL* uses 8-bit addressing (int) with the LSB as a flag forcing it to a 0 for *write* and a 1 for *read*. Whereas, MicroPython and PXT use 7-bit addressing, where the API will concatenate a new LSB to the passed address. For example the accelerometer is located at the following address:

$DAL = 0x3A = 00111010$

$MicroPython/PXT = 0x1D = 00011101$

The MicroPython/PXT address is the DAL address but shifted one binary position to the right for the concatenation of the read/write flag, whereas the DAL has left the LSB blank and ready to be forced to the appropriate read/write flag (1 or 0) by the API.

Writing

Fortunately, the addressing of a devices register when writing is less complex. For MicroPython and the DAL the process is very similar. The address of the accelerometer is first specified using the corresponding addresses above, followed by passing a byte/char array in the form of *[register address, value to be written]*. Writing cannot be performed using the PXT as the write block does not provide a way of addressing the internal I2C BUS devices, a devices register and a value in one block. The list of available registers and other accelerometer information can be found in the MMA8653FC Data Sheet.

The writing task consists of writing to the Y-axis offset adjust register (OFF_Y) a value, followed by reverting it back to its original value repeated continuously (while loop). However, to write to this register the control registers must first be altered (CTRL_REG1).

Reading

The reading of a devices register first consists of a *write* specifying the register that the data is going to be read from followed by a read. The repeat boolean must be set to *true* clearing the message end flag, and ensuring that the subsequent read command will receive the correct data and not a series of 1's (SDA default is high). As previously implied, the *read* command is then called passing a buffer or assigning a variable in PXT for the data to be read into. This process is applicable across all languages.

The reading task involves the retrieval of the data stored in the Back/Front, Z-Lock Trip threshold register (PL_BF_ZCOMP) continuously, containing the value 0x44. The DAL Lite for this test initialised the; scheduler, message bus and I2C.

Results

Figure 7 shows the results of these tasks. As expect the reading of data had a near identical performance to that of the magnetometer while loop task (Field Strength), but not the accelerometer, which

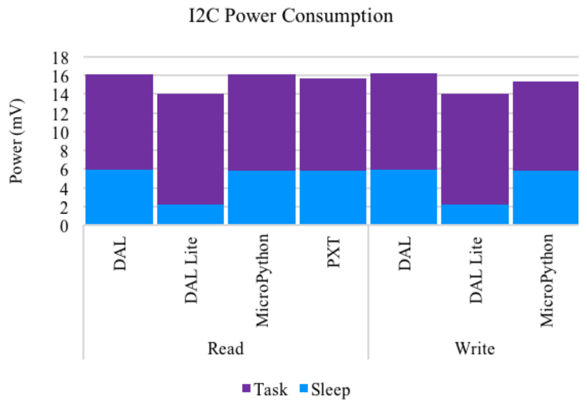


Figure 7: I2C Power Consumption

may be due better caching through the accelerometer APIs.

The reading and writing operations are almost identical in terms of their power consumption, except for MicroPython which uses $0.296mW$ less when writing. The DAL Lite for this section initialised the scheduler, message bus and I2C.

8 Radio

The radio on the micro:bit is integrated into the CPU as a 2.4GHz radio module. It is primarily designed for Bluetooth Low Energy (BLE) but can also perform micro:bit to micro:bit communication. This test contains three tasks; sending, receiving and enabled.

The DAL Lite contained the following initialisations; message bus, scheduler and radio. Also the *MICROBIT_BLE_ENABLED* has to be set to 0, for the API to work.

Results

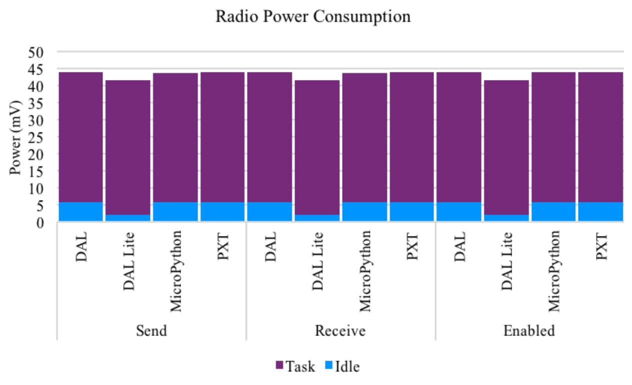


Figure 8: Radio Power Consumption

Figure 8 demonstrates the outcome of this test. What is evident is the similarity of power usage of each language for each task and overall. The radio being enabled almost completely 'drowns out' the cost of sending and receiving. Moreover, despite both the enabled and receive tasks going into continuous sleep after an initial setup it continues to use the same amount of power.

Cleary these results demonstrate that the most 'power-hungry' component on the micro:bit is the radio, and that a more power efficient protocol using the radio module would very beneficial.

9 Bluetooth

Bluetooth Low Energy (BLE) is a protocol included in the runtime firmware and is therefore available on PXT. However, due to MicroPython using its own runtime environment on the micro:bit, the BLE stack uses up 12 Kilobytes of RAM making it too large for it to be included as an API.

The DAL Lite for this component includes the storage, message bus, scheduler, display, bleManager and ble.

Results

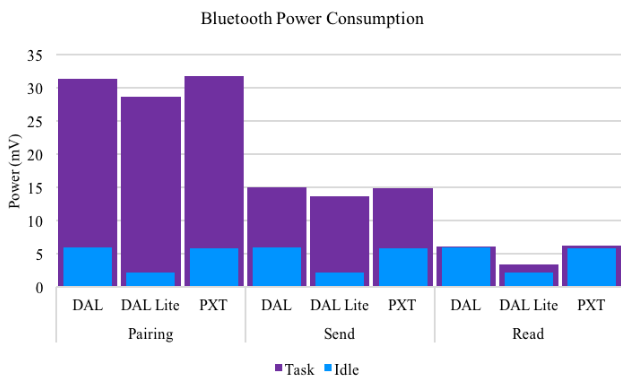


Figure 9: Bluetooth Power Consumption

Although, BLE does use a considerable amount of RAM, Figure 9 demonstrates that it is well worth the cost if the user wants to make a power efficient application. For sending, on average there is a 198% decrease is power consumption, and reading 792.51%.

The pairing task consists of the micro:bit in its broadcast mode, awaiting for a device to attempt to pair with it. This value is high due to fact that 17 LEDs are on as a visualisation that pairing mode is active, but still less than any task undertaken on the radio. The test begins when the pairing visualisation is shown.

The sending and receiving measurements only start to be recorded once a device is paired.

The pairing process used an iOS device running *nRF Connect*. The sending and receiving was done through the UART service.

10 Buttons

The micro:bit come with three push buttons; reset, A and B. The reset button is not tested as the power consumption is too hard to measure and not particularly useful data. Button A and B are buttons that all the environments can access and are often used in micro:bit projects for user interaction. All environments provide a high level API allowing the users to forget about complexities such as switch debouncing.

For this component, two tasks involving the detection of a button being pressed; loop and event are undertaken. The loop involves polling the status of the button to see if it has been pressed, which is a common although not recommend use case. The event registers an interest in a button being pressed and until it has been detected enters a power efficient sleep.

The DAL Lite for these tasks includes the message bus, scheduler, buttonA and buttonB.

Results

Figure 10 highlights these results with the event method clearly outperforming the loop and almost matching idle performance.

With the micro:bit PXT being the first exposure to programming for some, perhaps a consideration could be made to prompt the user that they could use an event, as it could have up to a 162% decrease in power consumption.

11 Serial

Serial communication is commonly used on the micro:bit through USB as a method of retrieving the current state of the micro:bit. It can also be used

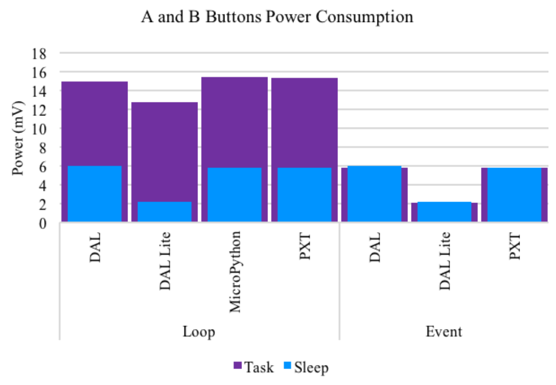


Figure 10: Buttons Power Consumption

through the PINs via a redirect command in the DAL and PXT, and by changing the UART init parameter in MicroPython. This is useful as the power measurement would become an issue if connected by a powered USB.

The DAL Lite includes the initialisation of the serial, message bus and scheduler. Pins 0 and 1 are set as the transmitter and receiver respectively.

Results

Figure 11 illustrates the results of this test with the DAL, DAL Lite and MicroPython almost identical in both tasks.

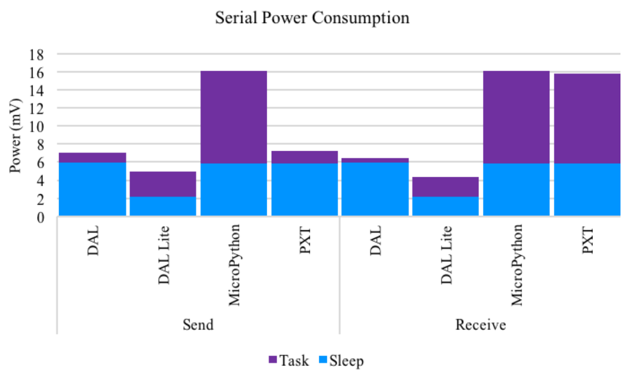


Figure 11: Serial Power Consumption

One interesting observation is the PXT when reading. It uses significantly more power despite is using the DAL API. When the DAL is reading in a while loop and no data is in the rx buffer the fiber goes into sleep (SYNC_SLEEP), which is why it almost matches the idle power consumption (sleep). Although PXT should do the same, this does not seem to be the case, and behaves as the SYNC_SPINWAIT describes by locking up the processor. The same can also be said for MicroPython, which has similar behaviour for both send and receive.

12 Summary

The data that has been collected should highlight areas of improvement in terms of power consumption and what software components excel compared to others for a particular task. All data collected and tasks (code) have been structured and documented for reusability, such that future hardware iterations can be easily recorded and compared, providing the API naming convention remains the same.

The five worst performing API components when compared to other languages performing the same task are:

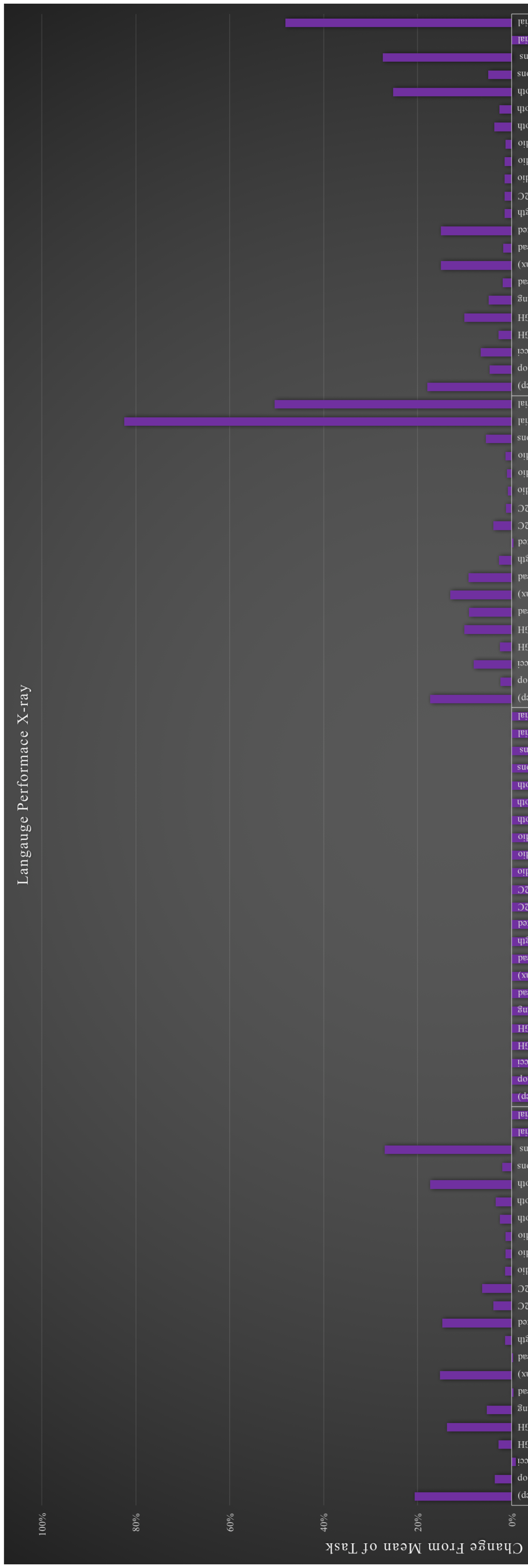
- MicroPython Serial Reading (82%)
- MicroPython Serial Receive (50%)
- PXT Serial Receive (48%)
- PXT Button Event (27.40%)
- PXT Button Event (27.40%)

Figure 12 shows a breakdown of each task for each language and how it compares to the average for a given task. This graph assists in finding the best and worst performers for a given task.

Also, attached to this report is a table of the tasks for each language with how much battery life they use (hours) when powered by two AAA batteries, calculated by the data sets captured in this report. However, some of the adverse results are due to simple design choices such as MicroPythons serial read not sleeping when the buffer is full.

One aspect that proves the validity of the data is the similarity of power consumption between the DAL and PXT throughout the whole testing process. A highly beneficial feature that could be considered by PXT and DAL is the initialisation of modules in the hex file only if they are used (i.e. DAL Lite).

Overall, the method of measuring the power consumption described in Figure 1 proved to be an effective. However, more isolation could take place using sense resistors at the surrounding power sources to given components, to help further target areas of improvement for future hardware iterations and APIs.


Langauge Performace X-ray
Change From Mean of Task