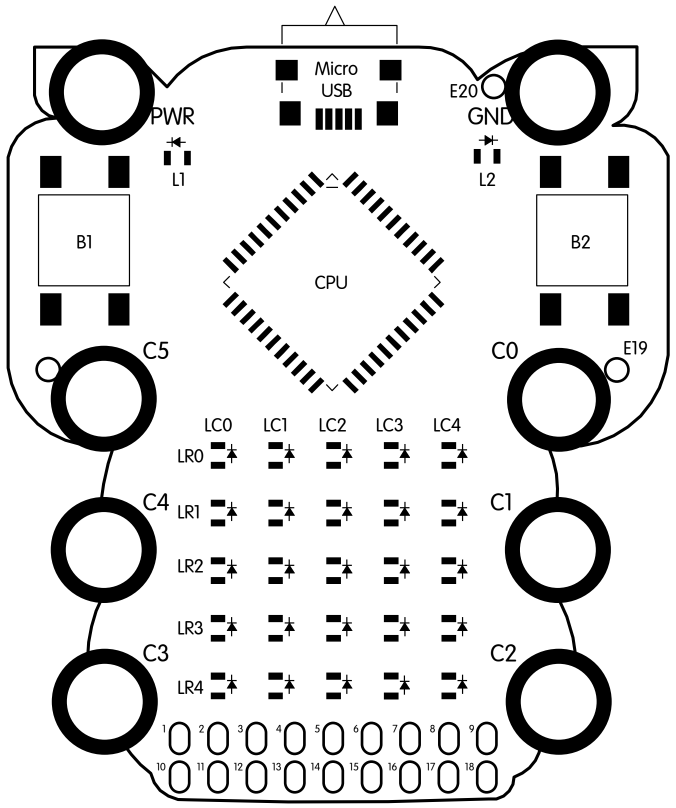
BBC Learning
Make It Digital

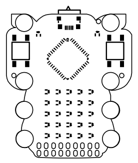
Project Microbug

Functional & Technical Specifications



Contents

1 Technical Specification



This document describes Project Microbug.

This is a working title for a BBC project targetted at assisting the nation's children with coding. This document using "Microbug" to describe the software and hardware platform as a matter of expedience. This is not the product name..

The intended audience of this document is primarily engineers who are interested in implementing, using or understanding one or more parts of the Microbug system.

This document consists of the following parts:

- A complete overview of what microbug is, and what it does
- A full functional specification of the system components. The functional specification is in no way prescriptive as to how the final system should be implemented.
- A full description of the complete reference implementation that has been created to sanity check the functional specification. The reference implementation is not intended to constrain the functional specification, but rather <u>illustrate</u> the concepts in the system.
- It is hope that potential partners will look at the reference implementation and be willing and able to offer better alternatives of various subsystems. This is actively welcomed. In order to assist with this chapter 3 discusses the impact of changing various parts of the system.

Aspects such as branding, form factor editorial direction and tone are out of scope for this document.

1.1 What is Microbug ?

Microbug is a platform supporting children who code, those who want to code, and those who need to code. The Microbug itself is a small, programmable, wearable device. It has an LED matrix capable of scrolling messages or simple graphics. It deliberately exposes the electronics of the device and can be powered by a coin cell battery

It has a number of crocodile clip connectors for attaching power and sensors. It also makes available a large number of IO pins for the owner to build more capable devices.

It can be programmed via a simple graphical web application and client side loader. It makes the underlying code visible to the user, and it compiles user programs online.

Furthermore, it can be used in a "tethered" mode allowing it to be controlled and act as a controller Tethered mode also enables users to explore Internet of Things related concepts and scenarios.

1.1.1 Aspirational

Microbug aspires to be simple to code for with a very low barrier to entry when getting started.. However it also aims to be a sufficiently capable platform such that it can grow with the user's experience. It does not aim to do everything, but does aim to enable users to take that first step in an engaging way to encourage them to take that second and third step too.

Key design constraints:

- Programmable – The device must be reprogrammable
- Simple to program – via a graphical browser based UI
- Small – maximum 5cm x5cm
- Cheap – manufacture cost must be approximately £2 or less when produced

in large scale – for example in batches of 100,000 units.

1.1.2 Technical

Technically, the device has the following high level characteristics:

- A small device with a microcontroller
- A 5x5 LED matrix
- 2 buttons
- 2 extra status LEDs
- 2032 coin cell battery power
- 2 crocodile clip connectors for external power (for example as a necklace)
- 6 analogue I/O crocodile clip connectors
- Access to a large number of digital IO pins, for attaching and controlling a wide variety of standard devices
- A reference implementation

The website has the following characteristics:

- Web based editng of programs graphically and textually.
- The ability to browse programs created by others
- Tutorials
- The ability for users and their mentors to have a user identity, and for mentors to be able to assist users on the system. Example mentors may be teachers, parents, youth group leaders, etc. The reference identity system utilises the fact that a device is distributed to users to enhance the user experience.
- It provides compilation services for taking web based programs and compiling them into a form suitable for running on the microcontroller. The primary service has at its heart a restricted-python to C++ compiler. There is a secondary C++ service.
- The python/C++ code targets a device abstraction layer rather than any given microcontroller's API.
- The website is specified such that there can be multiple editors, different identity systems, different tutorials, different storage systems and even potentially different hardware.
- The services the website uses are exposed as JSON oriented APIs
- There is also a reference implementation of the website

Client side, there is a need to get output of the compilation service onto the client. For reasons that will be detailed later this requires the use of a "loader" application. The features of a loader application:

- It will take a uHex file and flash the device with the code for the device.
- It can take additional data stored in the uHex file and use that to program the EEPROM data on the device.
- This client side application works on Windows, Mac OS X, Linux (including Raspberry Pi)

1.2 What pieces make Microbug up

1.2.1 Systemic elements

The functional elements of the Microbug platform are as follows:

- The device itself, and the bootloader on the device
- The client side loader application
- A browser
- The "static" elements of the system – eg editor website, tutorials.
- Web services – (eg compiler)
- Identity
- Tethered Operation (to extend utility of the device)

The reference implementation implements a version of many possible systems matching this description. We invite partners to bring their expertise, technology and services to the table to produce the eventual proposition. This means that the there could be changes to any part of the system, while the core editorial proposition remains the same.

For example the reference implementation includes an editor. There are many possible editors, and the specification allows for many different editors. Another partner may prefer a different chip at the core of the device. We invite partners to propose working with us in their area of expertise.

We are open to changes to the implementation. Implementation changes in one part of the system affect others. This document details these effects in section 3

1.2.2 Specific elements

The specific elements of the system break down as follows:

- Device
  - Core hardware elements
  - Simple & advanced expansion capabilities
  - 3 modes: Bootloader mode, run mode, tethered mode
- Client side loader
  - Bootloader functionality
  - Tethered mode functionality
- Browser
  - the users browser. A recent HTML5 based browser is generally assumed, but the system should work with old browsers
- Static elements of the website
  - Core web pages, templating, CMS
  - Program editor
  - Tutorials
  - Program browsing
- Web services
  - Program storage:
    - Immutable version storage, mutable programs relate to immutable

version(s).

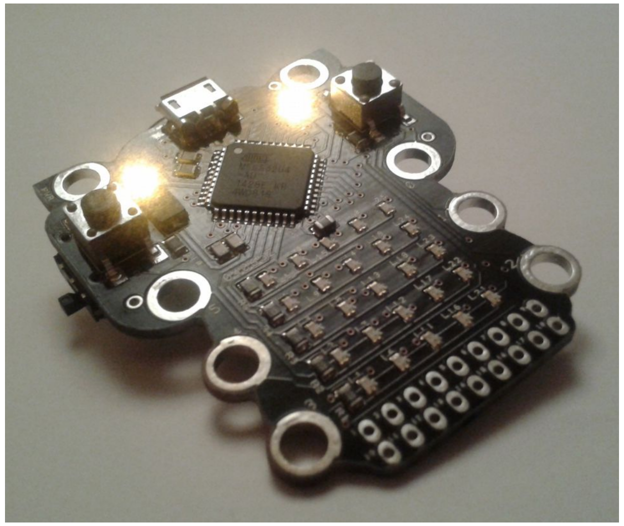
        - upload
        - download
        - browse
    - Compiler services:
        - Pro-active compilation
        - Python to .hex compilation
        - C++ to .hex compilation
        - Others, subject to agreement
    - API Services
        - Web services can all be used (and implemented) as JSON oriented APIs.
- Identity
    - Baked in concepts of identity:
        - Anonymous users – create programs
        - Basic users – create/own programs
        - Facilitator users – assist basic users (and above)
        - Independent edit permissions
    - Integration with thirst
- Tethered Operation
    - Tethered mode is provided to extend the utility of the device, and allow the device to act as a controller, a sensor, and to explore IOT concepts.
    - USB Serial based protocol to interact with the device - can be used without any library
    - Supports device introspection & control through a command line style API
    - Supports all aspects of the device
    - Potential support for user functionality
    - Python library provided to enable using the tethered device in a simple programmable fashion
    - Python library provided also allows IOT functionality:
        - Device can be automatically advertised onto the local network via mDNS
        - REST oriented API is auto generated via introspection
        - Clients can introspect the REST oriented API
        - Python code to automatically introspect a device, and create a local API for the device is also include.
        - The IOT library aims to provide an IP-like layer for devices to interact locally easily..
        - Does NOT publish any data beyond local network and must be explicitly activated. Requires a host machine.

1.3 Walk through of basic usage.

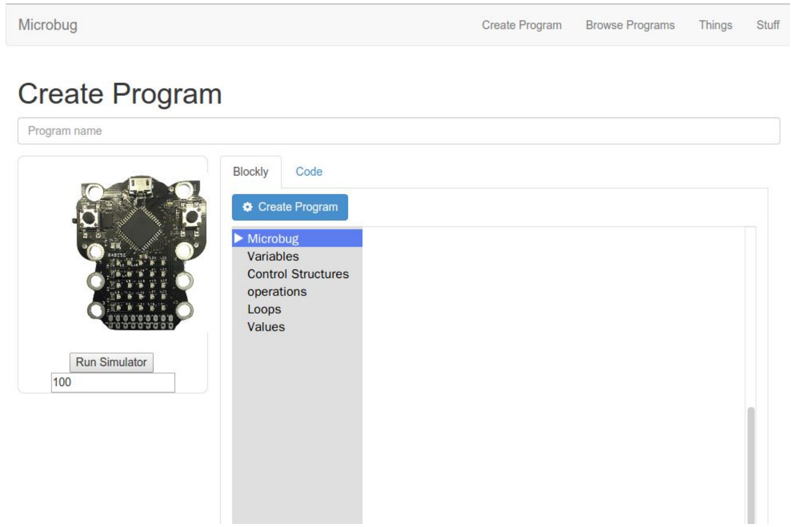
This is a walk through of basic usage of the reference implementation to illustrate the concepts used in Microbug.

NOTE: The reference implementation is in no way prescriptive. Use of a technology does NOT recommend that any particular technology or approach be used. Information relating to form factor, shape, colour, branding etc, are are provided separately. The implementation of software and electronics will be finalised in partnership.

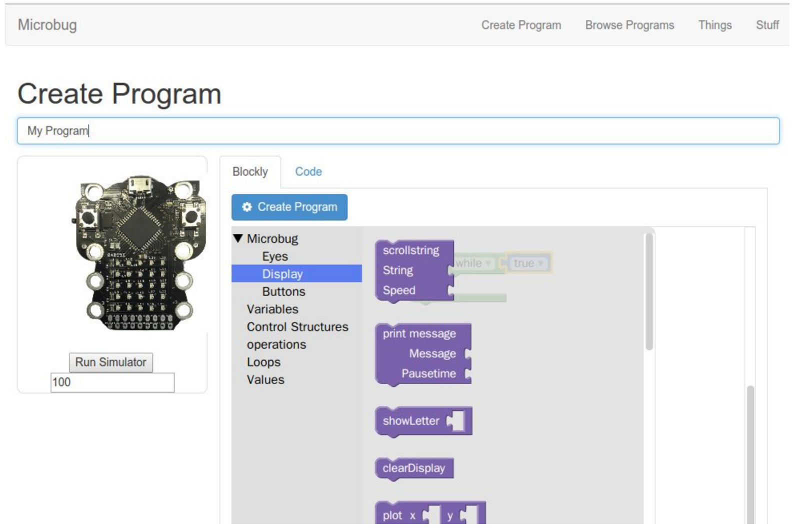
This is the current reference implementation device:



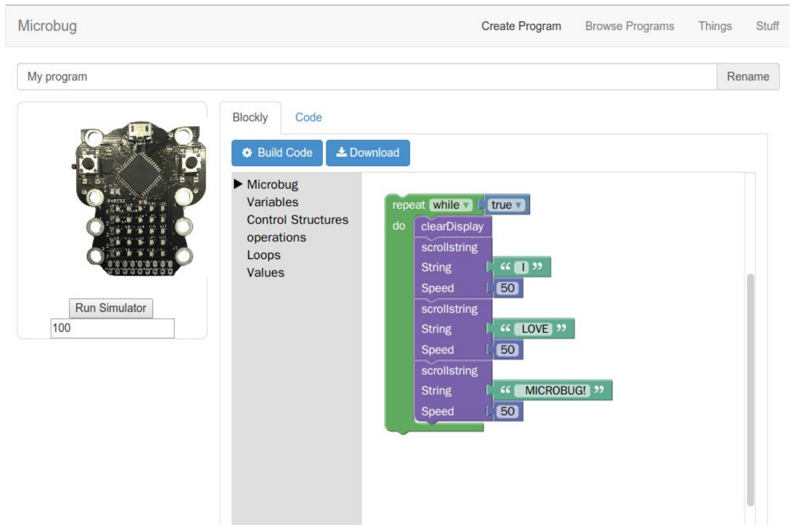
The user goes to the website, clicks on “create”, and is taken to an editor, where they can create a program of their choice. In the reference implementation it looks like this when the user goes to create:



They can start editing their program:

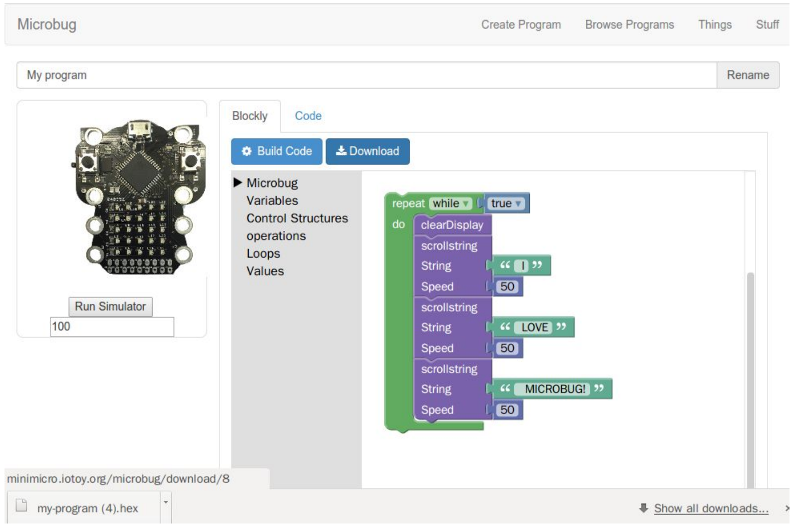


Once they're happy with their program they can save it:



```
repeat while true
do  clearDisplay
    scrollstring
      String " I "
      Speed 50
    scrollstring
      String " LOVE "
      Speed 50
    scrollstring
      String " MICROBUG! "
      Speed 50
```

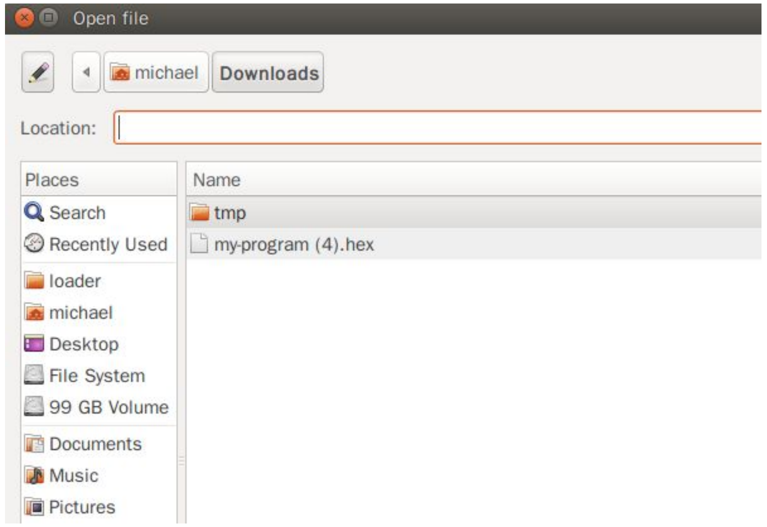
The code is compiled on the server at that point, which is available for download:



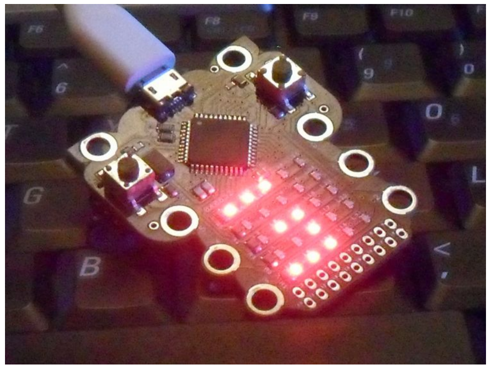
```
repeat while true
do  clearDisplay
    scrollstring
      String " I "
      Speed 50
    scrollstring
      String " LOVE "
      Speed 50
    scrollstring
      String " MICROBUG! "
      Speed 50
```

This results in them having a .uhex file on their machine.

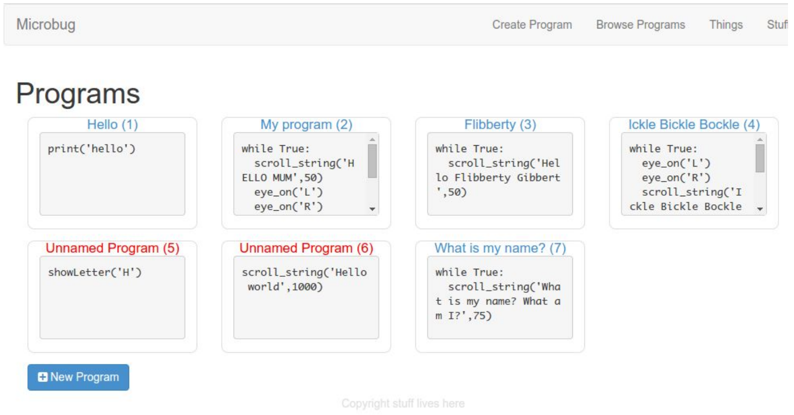
The user starts their client side loader to take the .uhex file for loading onto their device:



At which point the loader puts the code onto the device, which can then be reset to run the program:



Additionally, there is a browse area of the website. If we had gone to the website before they created their program, it would look like this:



Programs

Hello (1)
```
print('hello')
```

My program (2)
```
while True:
  scroll_string('H
ELLO MUM',50)
  eye_on('L')
  eye_on('R')
```

Flibberty (3)
```
while True:
  scroll_string('Hel
lo Flibberty Gibbert
',50)
```

Ickle Bickle Bockle (4)
```
while True:
  eye_on('L')
  eye_on('R')
  scroll_string('I
ckle Bickle Bockle
```

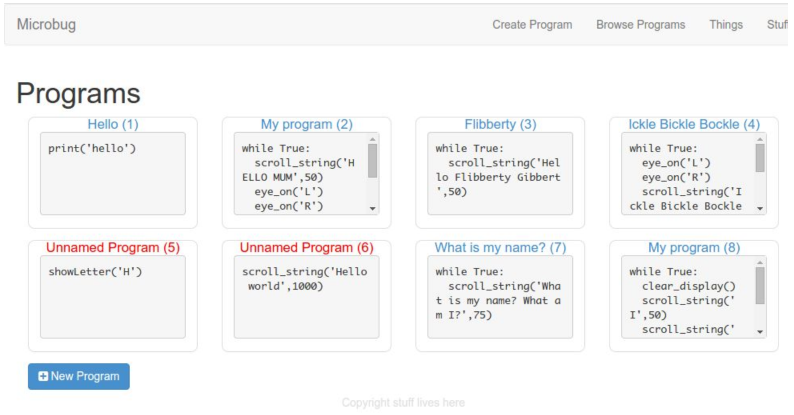
Unnamed Program (5)
```
showLetter('H')
```

Unnamed Program (6)
```
scroll_string('Hello
 world',1000)
```

What is my name? (7)
```
while True:
  scroll_string('Wha
t is my name? What a
m I?',75)
```

(The program names in red indicate the user didn't give that program a name - yet).

After the user creates the program we describe earlier, their program shows up here:



Programs

Hello (1)
```
print('hello')
```

My program (2)
```
while True:
  scroll_string('H
ELLO MUM',50)
  eye_on('L')
  eye_on('R')
```

Flibberty (3)
```
while True:
  scroll_string('Hel
lo Flibberty Gibbert
',50)
```

Ickle Bickle Bockle (4)
```
while True:
  eye_on('L')
  eye_on('R')
  scroll_string('I
ckle Bickle Bockle
```

Unnamed Program (5)
```
showLetter('H')
```

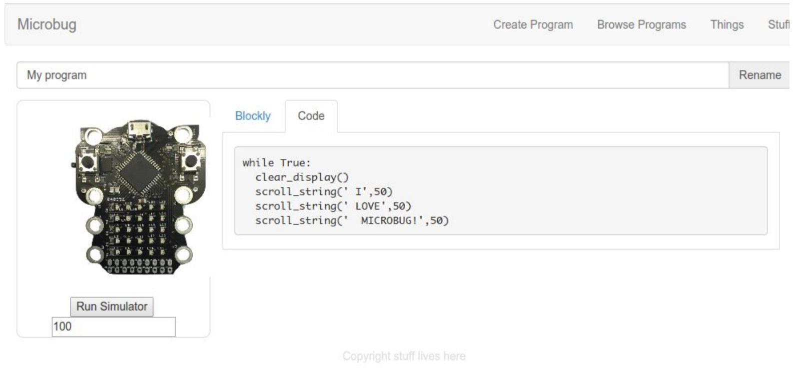
Unnamed Program (6)
```
scroll_string('Hello
 world',1000)
```

What is my name? (7)
```
while True:
  scroll_string('Wha
t is my name? What a
m I?',75)
```

My program (8)
```
while True:
  clear_display()
  scroll_string('
I',50)
  scroll_string('
```

If a user clicks on any program in that display they are taken to an editor preloaded with that program. They can then create their own version, download it to their device or look at the python code behind the blocks.

The python code you see here:



```
while True:
  clear_display()
  scroll_string(' I',50)
  scroll_string(' LOVE',50)
  scroll_string('  MICROBUG!',50)
```

… is auto-generated from the blocks, and is the code that the website compiler service actually compiled into code the microbug device can run.

Finally, it is worth noting that every editor page has access to a simulator – this simulates the effect of running the code. It is not however an emulator in that it doesn't emulate the effects of running the .hex file on a device

1.4 Operational overview

The following operational overview assumes the following standard setup:

- A client device – such as a laptop, or desktop computer, running a common operating system
- A server consisting of standard relatively static web components
- Web services including storage, program compilation etc.

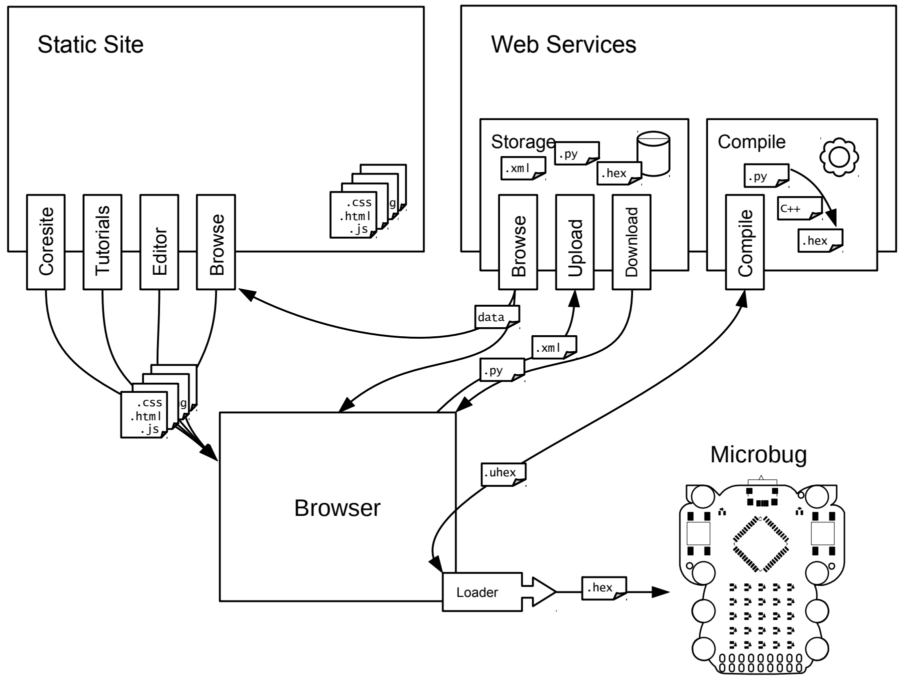
This split between subsystem functionality is illustrative, not prescriptive.

If the implementation was changed to entirely offline on a client device, all the server aspects and web services would either need to be running on the client, or running in an offline proxy mode.

If the implementation was changed to be app oriented – that is designed around running as an app on client devices such as iPads, Android devices, Windows Surfaces, and so on, then it is reasonable to expect parts of what is labelled on the following diagrams as static web or web services to run inside a client. Furthermore, the loader application on the diagram may also be integrated into the system.

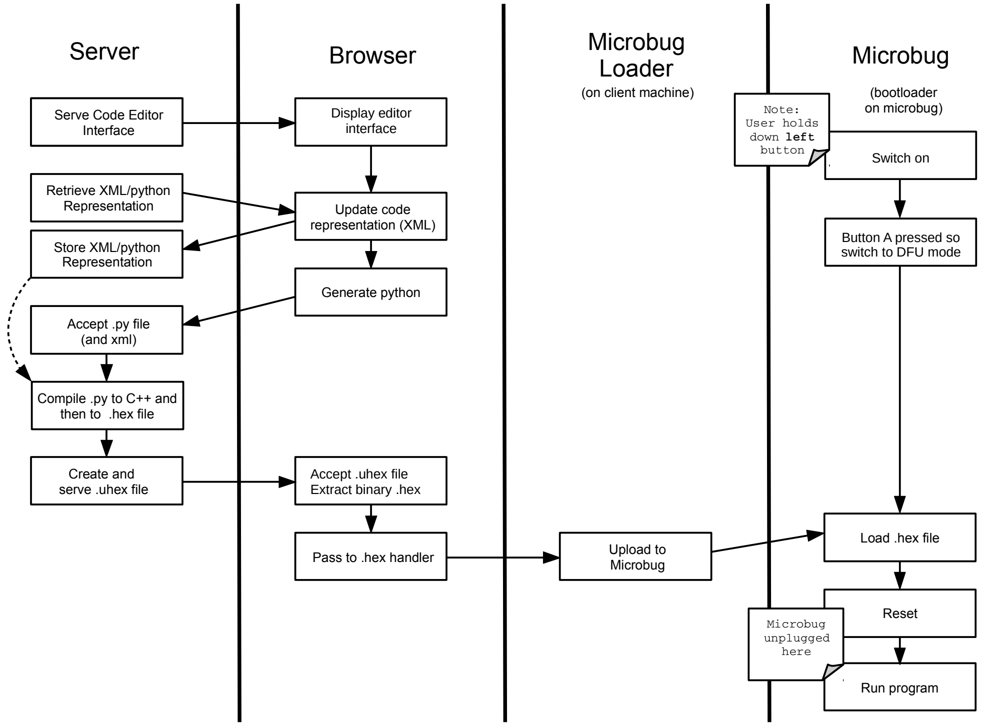
1.4.1 Operational overview – standard usage

Earlier sections described the functional subunits of the system. Having walked through how they are used, this diagram shows the relationship between the functional units.



Things this diagram does not show include: user identity and functional flow related to users. Please see the section on identity for this.

The following activity diagram shows the key interactions between functional units, as illustrated by the walkthrough in the previous section:


Server
Browser
Microbug Loader
(on client machine)
Microbug
(bootloader on microbug)
Serve Code Editor Interface
Display editor interface
Retrieve XML/python Representation
Update code representation (XML)
Store XML/python Representation
Generate python
Accept .py file (and xml)
Compile .py to C++ and then to .hex file
Create and serve .uhex file
Accept .uhex file Extract binary .hex
Pass to .hex handler
Upload to Microbug
Note: User holds down left button
Switch on
Button A pressed so switch to DFU mode
Load .hex file
Reset
Microbug unplugged here
Run program

This diagram does not include aspects relating to user identity, but provides an overview of how the system as a whole interacts to enable the user to program their device.

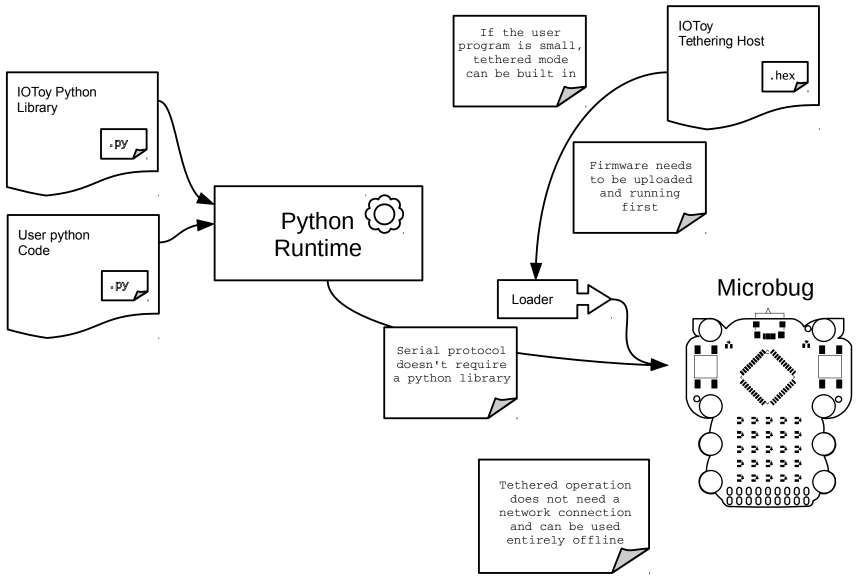
1.4.2 Operational overview – tethered usage

Tethered usage is where the device is connected to a computer and can be controlled and queried by a program running on the host computer. This includes scrolling messages, turning LEDs on and off, button presses, reading/setting pin values etc.

The reference implementation of tethering uses BBC R&D's IOToy project's IOT stack. [1]

The protocol it implements specified here as the protocol for tethering and network tethering. This protocol has been used on even very small microcontrollers

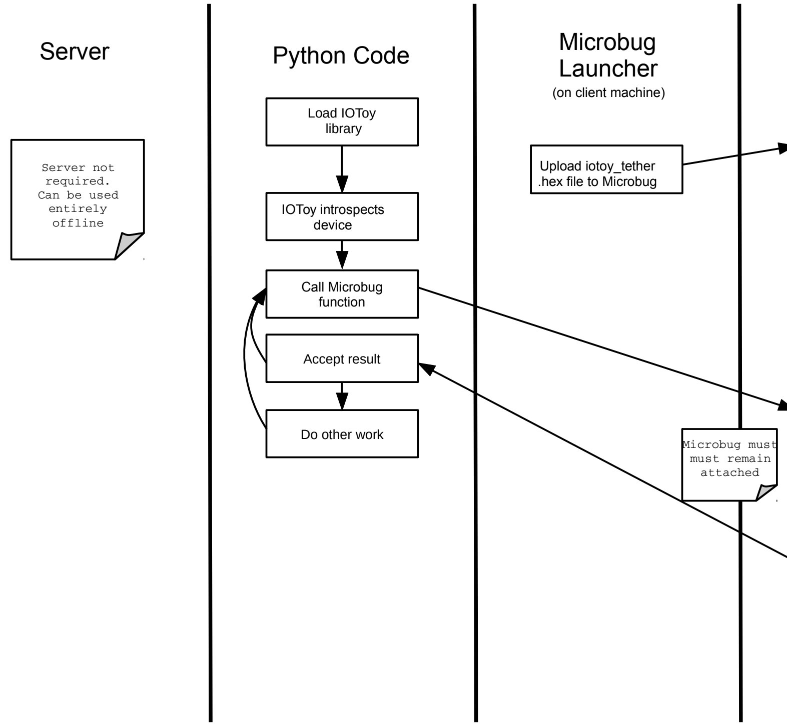
The following diagram summarises the context for the simplest interaction with tethered mode, using sample python bindings:


If the user program is small, tethered mode can be built in
IOToy Tethering Host
.hex
IOToy Python Library
.py
Firmware needs to be uploaded and running first
Python Runtime
User python Code
.py
Loader
Microbug
Serial protocol doesn't require a python library
Tethered operation does not need a network connection and can be used entirely offline

1 http://github.com/bbcrd/iotoy

Key points: Firmware for enabling tethering has to be on the device. That firmware provides a serial, line oriented protocol for interacting, controlling and introspecting the device. An example python binding exists to wrap that up for the user.

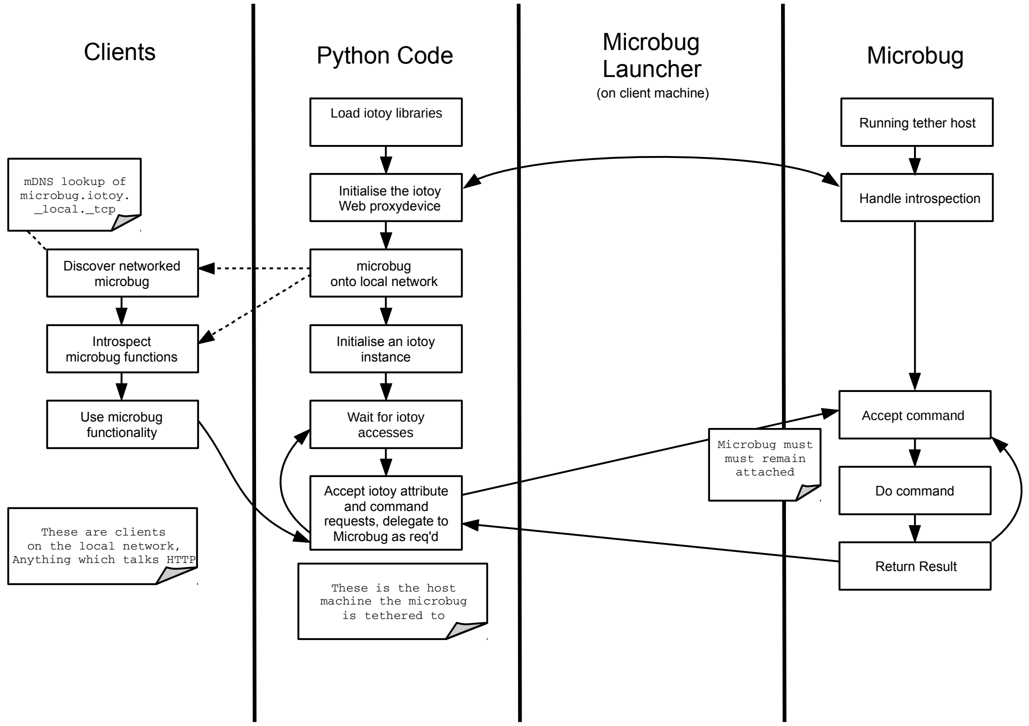
The following activity diagram describes the primary interactions with a device that only has the tethering code loaded:



Note: tethering mode does **not** require the use of a server, or use of a specific language. This allows the device to be used without installing any software specific to the device and discovery of what functionality it supports. Use of bindings and network connectivity do open up wider opportunities

1.4.3 Operational overview – Networked tether

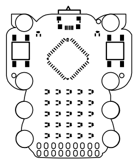
Networked tether mode introspects a device, and advertises the device as a RESTful service using mDNS. It also includes client side bindings for autodiscovery and introspection of microbug based IOT devices. The following activity diagram shows how this works from a host/device perspective:


Clients
Python Code
Microbug Launcher
(on client machine)
Microbug
mDNS lookup of microbug.iotoy._local._tcp
Discover networked microbug
Introspect microbug functions
Use microbug functionality
These are clients on the local network, Anything which talks HTTP
Load iotoy libraries
Initialise the iotoy Web proxydevice
microbug onto local network
Initialise an iotoy instance
Wait for iotoy accesses
Accept iotoy attribute and command requests, delegate to Microbug as req'd
These is the host machine the microbug is tethered to
Microbug must must remain attached
Running tether host
Handle introspection
Accept command
Do command
Return Result

Key points:

- The device is introspected as per the tethered mode
- That allows auto-creation of a REST/json oriented webservice, advertised over mDNS using a name derived from the device type
- Clients merely need to follow links In the json data mean in order to introspect the service and use it. What the data types mean is linked to from within the values.
- Sample python bindings are provided for the networked tether mode, but could be implemented in any language which can interact with a REST service that uses json for object structure.

2 Functional Specs



This chapter describes functional requirements of the system and also also non-functional perspectives. Generally speaking it describes what must exist in each layer in terms of functionality. It does not prescribe what that implementation should be.

Clarification of what the functional specification means can be found in two place:

- Firstly in the reference implementation
- The project team

The key focus of the functional spec is:

- To say what functionality is required
- To enable flexibility in the implementation
- To enable interoperation and co-ordination between a heterogeneous implementation.

2.1 Safety

The system must be safe:

- This means the device must be electrically sound and safe for an 11 year old to use
- The website must be designed and built with child protection issues and privacy in mind – it should no be possible for someone to be able to identify a child or their work without being their facilitator (eg parent or teacher)
- The web services should be implemented in such a way that data leakage cannot occur. They should also be designed such that they do not compromise the security of the system either directly or indirectly through denial of service attacks
- The client side loader should have a simple and clear implementation, to reduce the chance of the client side loader being a vector for problems for the user, facilitator

2.2 Users and Identity in the System

The system identifies the followng core active identities in the system:

- Anonymous users
- Anonymous users granted edit rights
- Users
- Facilitators – those who assist users
- Facilitators of Facilitators.

It should be possible to assert your identity either using a built in system or using an external identity provider. Users who forget their login credentials should be able to recover them, with the potential assistance of a facilitator and/or their device.

More details on this are described in the section on identity in the section on the website and its services.

2.3 Flexibility

In so far as possible, as much of the system must be able to be implemented by multiple partners to increase choice and flexibility, while retaining interoperability and compatibility and keeping costs to a minimum.

2.4 Supported Operating Systems

2.4.1 Desktop Operating Systems

The minimum supported operating systems for Microbug are:

- Windows (As many versions as practical)
- Mac OS X (As many recent versions as practical)
- Ubuntu (14.04LTS onwards)
- Raspian on Raspberry Pi

Support for other operating systems is very welcome.

2.4.2 Tablet / Phone Operating Systems

It is desirable for the Microbug System to have interfaces for running on devices running on leading smartphone and tablet systems. This includes Android (4+), iOS, Windows Surface and others. As with desktop operating systems, support for other operating systems is very welcome.

2.5 Device

The device MUST support (at minimum):

- Programming over USB
    - For example DFU, generic HID, FAT12 filesystem, etc.
- 25 LEDs – which may be controlled by as few as 10 I/O pins.
- 2 Buttons
- 2 status LEDs
- 4 or more analogue I/O pins – connected to via crocodile clips
- Power from a 2032 battery, USB or external battery
- Additional header connection points to allow more complex expansion capabilities
- The ability to run serial communications over pairs of I/O pins
- Being tethered, including control of the 25 LEDs, detection of button and I/O states, using the protocol described in the section on tethering

Minimum MCU specs (these specs can obviously be exceeded)

- Programs must be able to be created using a command line C (or C++) Compiler, to allow integration with the web platform.
- 8 bit, 16Mhz Clock
- 2K Ram, 16K Flash
- 18 I/O pins, which may be used: 10 LED matrix, 4 analogue I/O, 2 for buttons and 2 for status LEDs.
- A unique identifier
- EEPROM storage, minimum 256bytes, preferred minimum 1K

Desirable non-functional requirements:

- Tool chain that has widespread community support
- Device cost at scale should be £2 or less.
- Croc clips connectors should map to a 1cm grid spacing – to allow integration with other physical prototyping tools.

- Access to a number of IO pins on the device – perhaps via allowing soldering of one or more headers to the board, either via 0.1 inch grid pitch headers, or via ~5mm grid pitch screw in terminal blocks.
- Use of standard spacings and connectors to allow the creation of expansion “shields” to use with the device.
- Bricking of devices is a realistic possibility. While steps can be taken at the software level to try to prevent this, access to appropriate pins to enable last ditch rescue (eg via ISP) is highly desirable.
- Ability to group devices together in at minimum a daisy chain fashion.
- Allow exploration and teaching of IOT concepts

2.5.1 Device interface - Device abstraction layer

The purpose of the device abstraction layer (DAL) is to enable a hardware manufacturer to be able to change the implementation of the device right down to chip level, while presenting a consistent interface for the users of the device.

The device abstraction layer is a C++ artefact. It should not use templates, the STL or similar. By definition the device abstraction layer must relate to the hardware of the device and any basic operations.

By necessity therefore the specification of the device abstraction layer is far more specific than any other part of the system. Changes to the DAL are possible, but require co-ordination with other areas and implementors of the system. It is expected that the DAL may extend, but only via pre-agreed co-ordination.

The purpose of the device abstraction layer is to give clearly defined names for each part of the device in a way that leaves the implementation open to the device manufacturer. Essentially it abstracts away details of the hardware to provide core functionality, but without any policies on top of that hardware. It therefore performs a similar role to that of device drivers in a massively simplified single user operating system.

The device abstraction layer manages functionality that is exposed to the end user and that exposed to system implementers.

- User functionality:
    - Display management, access and update - including update frequency
    - Status LED management and update
    - Button management
    - Creation, management and display of images
    - Creation, management and display of displayable strings
    - Getting and setting of pin values for both analogue and digital pins
    - Standard names for all device I/Os, and common values
    - A font for text display
- Standard library functions, such as random, pause, sleep, memory allocation, tethering mode, EEPROM access, unique Ids, etc
- Startup handling to allow switching into bootloader mode, tethering mode or identity mode
- Provide the standard infrastructure for compiled programs to be inserted into when compiling programs.

As a result the device abstraction layer is a key subsystem in enabling flexibility in the system. Note, the device abstraction layer for a given device may well include other functionality used to implement the device abstraction layer. Use of that additional functionality in end user code however is not supported without agreed extension of the DAL.

This section outlines minimal behaviour in the device abstraction layer. Extensions to the device abstraction layer are possible, and would extend the functionality of the device. Extensions and changes to the device abstraction layer and subsequent interactions with other parts of the system are detailed in section 3.

2.5.1.a Display management

Display assumptions:

- The display assumes a 5x5 LED matrix.
- This 5x5 matrix is controlled by a 5 I/O lines tied to the rows and a further 5 tied to the columns. By marking a row I/O "high" and a column I/O "low", it allows current to flow between that row I/O and that column I/O, through any LED's connected to turn them on.
- By rapidly selectively marking some columns high, some low, you can iterate different (incompatible) patterns on different columns. This allows a persistence of vision style effect to be used to allow the display to show arbitrary patterns.
- The display updates at a frame-rate of 40Hz.
- As a result the display is driven at a rate of 200Hz to account for the 5 columns.

If a device implementation uses more than 10 I/Os, this ability must be emulated to allow code level compatibility with devices that use just 10 I/Os.

This display update routine is ideally run by a system timer. If this is the case, it should not interfere with hardware or software serial implementations. Ideally this should not interfere with common libraries.

Specific functionality and values:

The following constants are defined regarding the display:

`ON, OFF` – Constants for on/off levels for the LEDs
`DISPLAY_WIDTH, DISPLAY_HEIGHT` – Constants defining width/height of display

The following values are symbolic values for the IO pins for the row of LEDs

```
int row0, row1, row2 , row3 , row4
```

The following values are symbolic values for the IO pins for each column of LEDs

```
int col0, col1, col2, col3, col4
```

The following array is used by the display timer to render the display. Changing the contents of the display changes what is displayed.

```
int display[DISPLAY_WIDTH][DISPLAY_HEIGHT]
```

Internal setup functions

`void setup_display();` – Initialises the display. (Starts the timer, and the display hander)

Internal test functions.

These are useful for debugging the device, but not intended for end users

`void scroll_string(const char * str);` – Scrolls the given string across the display – with a delay of 50 milliseconds between scroll steps.

`void scroll_string(const char * str, int delay);` – Scrolls the given string across the display – with a delay of `delay` milliseconds between scroll steps.

Functions defined to operate on the display

`void clear_display();` – Clears the values in display[][]
`void plot(int x, int y);` – Sets display[x][y] to ON

`void unplot(int x, int y);` – Sets display[x][y] to OFF

`int point(int x, int y);` – Returns the ON/OFF value of display[x][y]

`void set_display(int image[5][5]);` – Copies the array image[][] to display[][]

The following two functions switch on/off the display timer/interrupt

```
void display_on()
void display_off()
```

2.5.1.b Status LED

The two status LEDs are provided to allow hinting about different behaviours. For example in DFU bootloader mode both status LEDs are required to light up to hint to the user the device is in "program me" mode. They are currently referred to as "eye" LEDs.

However, in the final board, the status LEDs may be in a different position, so anywhere the API says "eye", may later be replaced with "statusled". (For example eye_on may change to statusled_on)

Specific functionality and values:

The following constants are defined regarding the status LEDs:

`ON, OFF` – Constants for on/off levels for the LEDs

The following values are symbolic values for the IO pins for the two status LEDs

```
int lefteye, righteye
```

The following values are used to track the state of the status LEDs

```
int left_eye_state, right_eye_state
```

The status LEDs currently can be referred to as "L" and "R", or "A and "B"

The following functions may be used to manipulate the status LEDs

`void set_eye(char id, int state);` – Sets the status LED to the given state

`int get_eye(char id, int state);` – Sets the status LED's current state

`void eye_on(char id);` – Turn the given status LED on

`void eye_off(char id);` – Turn the given status LED off

`void eye_on(const char *id);` – Alternative API (first letter of string)

`void eye_off(const char *id);` – Alternative API (first letter of string)

`void toggle_eye(char id);` – Toggles the given status LED on/off

The alternative API is suggested based on the experience with the reference implementation.

2.5.1.c Buttons

Specific functionality and values:

The following constants are defined regarding the buttons

- `PRESSED, UNPRESSED`

The following values are symbolic values for the IO pins for the two buttons

- `int ButtonA, ButtonB`

The following function may be used to query the state of buttons. The buttons currently can be referred to as "A and "B". Note, the value returned refers to the button state **at that instant in time**.

- `boolean get_button(char id);`

2.5.1.d Images

The display can be used to display images. Images in Microbug may be larger or smaller than the display. If the display is smaller than the image, then the display operates as a viewport that can be panned over the image.

Specific functionality and values:

Images are defined simply as a byte array of ON/OFF values.

The following constants are defined regarding images

> ___ – This constant – three underscores – is provided to enable string literal representations of images to be clear in source. It is equivalent to “OFF”.

Images do not relate directly to IO pins, and therefore no values are defined relating IO pins to images.

Images are defined as stateful objects. They are defined using the following structure:

```
typedef struct Image {  – This implements...
   int width;
   int height;
   int *data ;
} Image;
```

The height/width define the interpretation of the data array. The data array is a collection on ON/OFF values. This may also be ON/___ values. An example Image literal is:

```
Image myImage;

int image_data[] = {
    ___, ___, ___, ___, ___, ___,  ON, ___, ___, ___, ___, ___,  ON, ___, ___,  ON,
    ___, ___,  ON, ___, ___,  ON, ___,  ON, ___, ___, ___,  ON, ___,  ON, ___, ___,
    ___, ___,  ON, ___, ___, ___, ___,  ON, ___, ___,  ON, ___, ___,  ON, ___,  ON,
     ON, ___,  ON, ___, ___, ___, ___, ___,  ON,  ON, ___, ___, ___, ___,  ON,  ON,
    ___,  ON,  ON, ___, ___, ___, ___, ___,  ON, ___, ___, ___, ___,  ON,  ON, ___,
    ___, ___, ___, ___,  ON, ___, ___,  ON, ___,  ON, ___, ___,  ON, ___, ___, ___,
    ___, ___, ___, ___,  ON, ___,  ON, ___, ___,  ON, ___,  ON, ___, ___, ___,  ON,
     ON, ___, ___,  ON, ___,  ON, ___, ___, ___, ___,  ON, ___, ___, ___, ___, ___,
};

myImage.width=16;

myImage.height=8;

myImage.data = image_data;
```

Functions for working with images:

- `void showViewport(Image& someImage, int x, int y);` – This displays a display sized viewport starting at location x,y within the image.
- `void ScrollImage(Image someImage, boolean loop);` – This scrolls an image across the display. If loop is set to true, when the display viewport scrolls off the image, it displays the wrapped version of the image.
- `int image_point(Image& someImage, int x, int y);` – Returns the ON/OFF value of the point at the given location in the image
- `void set_image_point(Image& someImage, int x, int y, int value);` – Sets the ON/OFF value of a given point at the given location in the image

Internal function for working with images:

- `inline int image_point_index(Image& someImage, int x, int y);` – Returns the index value into the data array for a given point x,y in the image.

2.5.1.e Displayable Strings

In order to display strings on the display, they need to be converted from a collection of character bytes to an image that can be displayed. How this is implemented internally is implementation dependent.

The API defined for use has a working assumption that rather than render a 45 character string as an image 225 pixels by 5 pixels, that instead an optimised version will be used. This is based on optimisations used in the reference implementation where only two characters at a time are rendered to an internal data buffer.

A displayable string is represented by a data structure called a StringImage. This uses a C++ struct. Operations on the struct itself are performed using member methods.

The “current” viewport window on the virtual image is is tracked via a “current pixel position" representing the top left corner of the visible viewport.

The fact that the display can only show a subset of the string which is used to optimise the display given the limited space for data and in RAM on a microcontroller. The current pixel position in a StringImage is one way of achieving using that.

Structures defined here:

- `struct StringImage ;` – This implements...

Functions defined that operate on StringImages:

- `void scroll_string)mage (StringImage theStrImage, int pausetime);` – This implements...

Member Methods in StringImages:

  - `StringImage(const char * str)` – String Image constructor. Initialises the constructor with the given string
  - `void setString(const char * str)` – Changes the string the Image refers to to the given string
  - `void render_string()` – This renders the currently visible portion of the string into the internal image buffer
  - `void update_display()` – Renders the internal image buffer to the display
  - `void pan_right();` – Increments the internal current pixel position by 1
  - `void pan_left();` – Decrements the internal current pixel position by 1
  - `void set_pixel_pos(int pos)` – Sets a specific "current" pixel position in the string. A 45 character string is 225 virtual pixels wide.
  - `int pixel_width();` – Returns the width of the current StringImage in pixels. Implementations may choose to actually store the pixels. In the reference implementation, these are virtual pixels

Possible extensions: to include panning up and down strings, to render strings from right to left, up to down, and down to up, and add addition "pan" and set_pixel_pos alternatives. Extensions are welcome, but this is the minimal core.

2.5.1.f IO Pin Control

Access to and control of IO pins beyond display are really what enables Microbug as a proposition to have greater longevity. This represents a minimal device abstraction layer.

However, this is sufficient for many sensor and control tasks.

The following constants are defined regarding the buttons

- `PRESSED, UNPRESSED`

The following values are symbolic values for the IO pins for the LED array:

- `int row0, row1, row2, row3, row4   - This implements...`
- `int col0, col1, col2, col3, col4   - This implements...`

The following values are symbolic values for the IO pins for the status LEDs and buttons:

- `int lefteye, righteye, ButtonA, ButtonB`

The following values are symbolic values for the IO pins for the croc clip connectors

- `int lefteye, righteye, ButtonA, ButtonB`
- `int croc0, croc1, croc2, croc3, croc4, croc5`

The headers along the bottom of the device are also usable as digital pins for IO. The specific values available mapping to pins follow. For each header, a second values is given in bold like this: `h1` **`3,`** `h2` **`4,`** etc The value in bold is the silkscreen id on the board on the headers along the bottom. The values (and mappings) are:

- `int h1 3, h2 4, h3 5, h4 6, h5 7, h6 8, h7 9`
  `int h8 10, h9 12, h10 13, h11 14, h12 15, h13 16, h14 17`

There are two extra additional hardware pin points on the device – ee19 and ee20. These are designed for emergency or expert use only.

They are accessible from the DAL via the two following symbolic values:

- `int ee19, ee20`

Functions for working with digital IO pins:

- `void set_pin(int pin, int value)` – Sets the given pin to HIGH or LOW
- `int get_pin(int pin)` – Gets the current pin values as HIGH or LOW

Functions for working with analogue IO pins - specifically the croc clip connectors:

- `void set_analogue_pin(int pin, int value)` – Sets the pin value to a value between 0 and 1023
- `int get_analogue _pin(int pin)` – Gets a pin value – retruns a value between 0 and 1023

2.5.1.g Font

The DAL defines the existence of a 4x5 font. It is designed to be used with a single strip of blank pixels down the right hand side. The specific font defined is included in the appendix to this document.

It is primarily used internally within the DAL, but there are some specific functions that are used regarding the display of letters as images:

- `void showLetter(char c)` – Renders the given letter into display[][]
- `void print_message(const char * message, int pausetime=100);` – Renders each letter of the message in sequence onto the display, with a delay between each letter being rendered of pausetime milliseconds
- `typedef struct StringImage` – as defined In section 2.5.1.e this is used for working with strings as images
- `void render_string()` – This specifically renders the current visible portion of the string image into it's internal buffer

2.5.1.h Library Functions

There are a collection of library oriented functions in the DAL, which are required for various subsystems.

Basic functions

- `long random(int min, int max)` – Returns a random number between min and max inclusively
- `void pause(int time)` – Causes the program to pause execution of time miliiseconds
- `void sleep(int time)` - Causes the microbug to sleep for the given time period in microseconds. When the device sleeps, all functionality on the device insofar as possible – including display update – comes to a halt

Tether related :

- CommandHostTiny – This class implements the core command host as used by the tethering code. The class implements a number of methods. Those intended to be overridden in a sub class are underlined:
  - `const char *hostid()` - Returns the host id for the device.
  - `const char * attrs()` – Returns a string containing a comma separated list of names representing device attributes.

- `const char * funcs()` – Returns a string containing a comma seperated list of names representing device functionality.
- `bool has_help(char * name)` – Returns true is the device will return help for the given name
- `void help(char * name)` – Emits help over the serial console for the given name
- `bool exists(char * attribute)` – Returns true if the device implements the given attribute
- `const char *get(char * attribute)` – Called when the serial API requests the given attribute, the response is a string
- `int set(char* attribute, char* raw_value)` – Called when the serial API requests the given attribute is set to the given raw value
- `int callfunc(char* funcname, char* raw_args)` – Called when the serial API requests the named function to be called with the given argument(s)
- `void setup(void)` – Mandatory internal setup function

Common library functions:

- `int strcmp(string1, string2)` – If the two strings are equal, the result is 0.
- `int  atoi(const char* raw_string);` - Converts raw_string to an integer.
- void itoa (int value, char * result_string, base); – Converts a given integer value to a string in the given base, and stores the result in the result_string – which must be large enough to hold the value.
- `char * strstr(source_string,search_string)` – Searches for search_string within the source_string. If not found the value returned is NULL, otherwise the value returned is a pointer to where the value is found.

EEPROM Access

NOTE: Addresses 0 to 127 are reserved for use by a DAL implementation

- `int get_eebyte(int location)` – Reads a value from the given location in EEPROM.
- `void set_eebyte(int location, int value)` – Sets a value in the given location in EEPROM
- `void set_last_userid(const char*)` – Sets the id of the last user to program the device in EEPROM. This is defined as starting at location 0 in DAL reserved space. The first byte is the length of the username
- `const char *get_last_userid()` –.Gets the user id of the last user to program the device.
- `const char *get_unique_id()` – Returns a unique id of the device. For devices with a microcontroller with an inbuilt chip ID, this id can be that chip ID. If it's been overridden, then this should return the overridden id, which is stored in EEPROM.
- `void set_unique_id(const char* new_id)` – Sets the new unique id for the device, which is stored in EEPROM.

USB

- USBDevice.attach(); - Attaches & initialises the USB connection
- Usb_detach() - Detaches the USB connection

Conversions

- `int int(some_float);` – Converssts some_float to an integer
- `double double(wholepart);` - Converts whole_num to a floating point value

Serial console access

- `Serial.begin(int baudrate);` - Initialises the serial console at the given baudrate
- `Serial.print(const char* name);` - Sends the given string to the serial console
- `Serial.println(const char* name);` – Sends the given string to the serial console followed by a new line
- `Serial.available()` - Returns the number of bytes available on the serial console
- `char Serial.read();` – Returns the next byte from the serial console

2.5.1.i Startup Modes

Functions relevant to system startup mode:

- void bootloader_start(void); - When called, the device resets itself back to its base/initial state and launches the devices' DFU bootloader – assuming a DFU based bootloader (or similar)
- void check_bootkey(); - At start up checks to see if the left hand button is pressed down and switches the device into bootloader mode – could also check the righthand button for an identity start or both buttons for a tethered start.
- void tethered_start() - switches the device into tethered mode
- void identity_start() - Switches the device to identity recovery mode – specifically displays the last programmed username on a loop

2.5.1.j Compiled Program Hosting Infrastructure

Programs that are compiled on the hosting infrastructure should assume:

- `#include "dal.h"`– This line must be included at the start of the user c++ code
- `USBDevice.attach();`– This should be called to initialise the USB connector at startup
- `void loop()`– This should implement the body of the any core functionality that is to be repeated
- `void setup()`– This should implement the device's required startup. In particular, custom user programs must call microbug_setup – in order to enable the startup modes.
- `microbug_setup();`– As noted above, this must be called at device startup from the setup() function.

See the reference implementation for more detailed examples.

2.5.2 uhex file specification (Universal .hex file format)

The microbug reference implementation at present use .hex files.

A .hex file is a file with many lines of this format:

```
:1000C00007080B0816081108200000000000210492
:1000D00004040004220A0A000000230A0F0A0F0A7F
:1000E00024060D060B062509020409002606004094C
```

These lines essentially specify which parts of the flash memory contain which values. Then, when the device is switched into run mode, the MCU starts execution at a given point in memory. This means that the .hex file format for one MCU differs from another radically. The origin of this file format comes from intel and is commonly usefd with lots of microcontrollers.

If there were to be only one possible microcontroller for the microbug device implementation, then we would have only ever one .hex file format to consider. In that scenario, using .hex files directly could make sense.

However, we need to leave the door open for partners who are willing to manufacture devices to be able to change implementation details such as the chip at the core of the system. Additionally, the uhex format allows us to include meta-data.

This necessitates a need to have a file format that allows multiple binary representations to co-exist in a single file. Furthermore, we have a need – due to identity in the system – to be able to state "this hex file was built by this user". Doing this outside the .hex file has utility, since we can then store that data inthe EEPROM data of the device.

As a result the uhex file format is born.

Specification:

A uhex file is a json format file.

It contains just one json object

That json object has the following fields.

- Build-user – the identity of the user that download this .uhex file
- hexfiles

The hexfiles field relates to a json object of hexfile representations. Each field is a type of hexfile. Each type relates to a given microbug device. There have been 3 iterations of the reference implementation, so valid types at time of writing are: megabug, microbug, microkit. The value of this field is the hexfile itself.

In particular processing of a uhex file can proceed as follows:

```
import json
data= open("my-program.uhex").read()
j = json.loads(data)
print "Built by", j["build-user"]
print "Hexfile for microkit", j["hexfiles"]["microkit"]
```

This allows the user's client side loader to load the right sort of code onto the device.

2.5.3 Boot Loader

The boot loader is the firmware running on the microcontroller that allows it to be programmed. On entry level devices, this is generally a different mode of running the device. The boot loader is often a feature of either a given chip and/or a given tool chain.

The minimal functional requirements on the bootloader are as follows:

- Must be able to be switched into from software running on the device
- The hardware must make it possible to the device into bootloader mode. If this is generally under software control, this can be of the level of.
- Must be able to accept updated firmware over the USB connection
- Preferably use a recognised protocol for USB update, such as:
    - Generic HID
    - DFU update
    - To present as a file system with firmware file
    - Other protocols may be acceptable, depending on cross platform tool support.
- There must be either
    - Cross platform command line tools on the supported client operating systems, for communicating with the boot-loader to perform the update.
    - A cross platform C-library is a requirement to allow a cross platform firmware update tool to be created.

Devices must be supplied a bootloader installed. If a device is supplied without a program to run, it MUST default to running in the bootloader mode.

For more details on device update, see the next question on the "loader" application.

2.5.4 Loader

The MicroBug loader is an application running on the supported operating systems has the following core functionality:

- It is a graphical application that takes a compiled program and uploads it to the device.

2.5.4.a Loader Core Characteristics

It has the following preferable characteristics:

- Minimal installation
- It must be a graphical tool – though it must also be possible to use the functionality of the loader as a command line tool.
- No drivers (if possible)
- Integration/launch via a custom mime-type – allowing integration with browsers.
- Provides a helping hand for the user
- Helps prevent “bricking” of devices
- Potentially validates code for upload onto a device
- Potentially validates devices to load onto

Possible characteristic:

- Assists in the creation of automated serial numbers of IOT purposes

2.5.4.b Loader Modes of Operation

There are three possible types of operation

1. The device presents as a file system that works identically on the supported operating systems. This represents the ideal world state.
2. The devices presents as a filesystem that has correct operation default dependent on the operating system in use, and incorrectly on other operating systems.
3. The device requires a protocol like DFU update or similar to perform an update and requires the use of a program or library.

Options 2 and 3 have essentially equal merits due to practicalities, however option 2 has slight preference due to giving at least one operating system a second option.

Option 1

To meet this type of operation

- The device presents as a file system containing one file such as firmware.bin
- This file may be deleted to erase the device
- A new file can be created called firmware.bin, which results in the device being flashed with that image
- The firmware.bin file can be copied off the device for backup purposes
- That all 4 operations MUST operate identically on all of the supported operating systems.

It is our experience of looking at low cost microcontrollers that this functionality does not interact identally on all 4 operating systems. In particular this functionality operates differently on Mac OS X and Windows, due to file handling differences.

If a device does operate this way, then no loader application is required.

However, it may still be be useful for reasons discussed later in tethered mode of operation and also to enable standalone/offline usage of the system.

Option 2

To meet this mode of operation, on at least one of the supported operating systems the following behaviour must be possible:

- The device presents as a filesystem containing one file such as firmware.bin
- This file may be deleted to erase the device
- A new file can be created called firmware.bin, which results in the device being flashed with that image
- The firmware.bin file can be copied off the device for backup purposes

Other operating systems may interact with this process differently. If that is the case, the following must be true:

- The device presents as a filesystem containing one file such as firmware.bin
- This file should not be deleted
- It must be possible to update the firmware.bin file in place – ala unix tools like "dd"
- The firmware.bin file can be copied off the device for backup purposes

In devices that support file systems, it is our experience of looking at low cost micro controllers that this mixed handling of file systems on different operating systems is the common case. Asking users on one operating system to use "dd", and others to simply copy files is liable to leave to confusion.

Given the differences in behaviour between the supported operating systems, a loader application is still required. The reason for this is to ensure that all users of the device have the ability to update their device in a way that matches the same instructions for any given device. This allows users of the device to share their knowledge about how the device works without worrying about what operating system they're using, since they may not even know.

On the operating system where the file-system operates as the device expects, the file system based approach is something that can be used. However doing so will be treated as an emergency alternative.

As a result, the preferred approach will be to use a loader application in option 2.

Option 3

While this approach should use a standard protocol for update, the protocols generally used are often only supported via a stand alone application.

In this scenario, the loader application should operate cross platform and ideally have as few requirements as possible. It must be a graphical tool, but that graphical tool may wrap a command line tool.

2.6 Offline Support

Offline support is highly desirable, but not a core functional requirement. Offline support could be implemented the following approach:

- Running a webserver inside the loader application. (for example, using the built in server to django or flask)
- Ensuring that webserver can only be accessed from the local server
- Allowing the web application that runs the website to run off that server

For example, shipping the reference implementation as a client side application running behind a Flask/Django webserver is something that could be done. This has not been done for the reference implementation, but is something that could be desirable, and would fulfill this functional requirement.

While 95% of homes have broadband facility, 5% of homes in raw numbers is still a very high number of homes, and often in under served demographics.

Speculative solutions for offline support are therefore an open area for discussion.

2.7 Browser Requirements

The preferred browser support requirements are detailed here:

http://www.bbc.co.uk/guidelines/futuremedia/technical/browser_support.shtml

In particular, those creating web related content should target Level 1 clients as per those requirements. The key requirements on the browsers are:

- To support modern drap and drop based editting of graphical program code
- To support modern text based editting of textual program code
- HTML5 Canvas support – for rendering graphical views and simulators

Ideally creators of various web components in the system should bear in mind that the first contact with the device is likely to be through schools. Schools often do not run the most recent versions of browsers, so it would be wise to consider supporting level 2 browsers.

In particular though, implementations should state which level they support.

2.8 Identity

Identity in Microbug has the following core working assumptions, principles and core requirements:

- All users have poor memories, and identities do get compromised.
- The **prime users** are children – that is vulnerable users.
  - Uniquely identifying information regarding prime users **MUST NOT** be stored in the microbug system. **The system should actively aim to dissuade/prevent the storage of uniquely identifying information for prime users.**
  - Every prime user of the system will have their own microbug device, programmed by that prime user. Microbug devices will change owners – either accidentally, or deliberately.
- Every prime user has at least one helper, or **facilitator:**
  - Examples: parents, teachers, youth group leaders, scout/guide leaders, etc. ie a person with whom the child already has a pre-existing mentor/mentee relationship.
  - A facilitator's systematic purpose is to assist prime users with account details – password resets, account recovery, etc. Functionality revolving around facilitators is there to support prime users usage of the system. Only minimal information will be stored to enable facilitation
  - Facilitators may use the system in the same way as a prime user
  - Every **facilitator** may themselves have a facilitator – a super-facilitator. (That is someone who can assist them with their account)
  - Facilitators can only "see" accounts they directly facilitate (& own)
  - Prime users will need to change/add/remove facilitators
  - A prime user **must not** be a facilitator
- Accounts:
  - A prime user needs a way to recover their account in the event of losing their account details, and having disconnected their facilitators

  - Users can self register as prime users or as facilitators, in the event of account loss.
  - Attempts should be made to prevent prime users registering as facilitators. For example, having facilitators confirmed as adults by other facilitators. eg via manually or via a confirmation code style approach etc.
- Distribution chain
  - It is currently expected that devices will be gifted to children through schools. There could be other mechanisms, but the key point is they will be distributed through adults who facilitate groups of children.
  - It is currently expected that devices will be distributed in batches of a given size. Each batch can include an equal number of pre-generated identities. This can include a confirmation code.
- Devices are programmed by a user with a given identity using a loader application. In addition to loading the program into flash, that identity could be loaded into EEPROM data.

This description of identity very deliberately avoids discussion of how this will be implemented. Any authentication mechanism is used is orthoganal to this set of role requirements. This model of identity is based on taking best practice forms of prime user and facilitators from a variety of groups and systems.

Specifically the following approaches have been used as starting points:

- Online children communities – such as Moshi Monters, Club Penguin, Bin Weevils etc. These all use a parental facilitator to act as a helper to the child. The difference here is that we can provide facilitators usernames/passwords upfront, and that the initial batch of facilitators are guaranteed to be adults.
- Electronic ID tokens. The system takes advantage of the fact that we have a physical device that the child owns. We can use it to assist in account reminders. While this isn't a traditional eID token, this approach of using a physical programmable device to store/retrieve a piece of unique information to assist with login is inspired by various eID tokens.
- Scout & Guiding organisations. Scout and guiding organisations has 100 years of experience in terms of managing adults in assisting volunteers in mentoring, guiding & protecting children. The idea of using facilitators to confirm facilitators, is based on the model used in scouting. In particular, adults must be vouched for by other adults in order to be added to scouting systems. If a weak link is discovered, then any weak links can be audited and dependent links broken.
- The idea of a super-facilitator – that a facilitator can have a facilitator, is based on many standard models of system administration. In youth organisations this may map to a group leader or district leader. In groups of parents this could map to a parent who happens to be particularly tech-savvy. In schools this could map to an IT manager/assistant, an ICT teacher. By making it such that the superfacilitator cannot see the view of the system the facilitator has, it doesn't risk or enable access to prime users.

2.8.1 Operation of Identity

The descriptions within the following scenarios are illustrative, not prescriptive.

The system MUST support prime users in the following optimal scenarios and failure scenarios for identity. We would expect that if taken on by a partner that identity would be fleshed out to meet best practice. Furthermore, any implementation of identity is subject to ratification according to BBC Editorial Policy, specifically including those revolving around child protection guidelines and information security.

Optimal Scenario:

- The facilitator has pre-installed the loader software on the machines the prime users will be using.
- A batch of devices are shipped to a facilitator along with a list of usernames & passwords on an accompanying sheet. This includes a facilitator username/password
- The facilitator gives devices, usernames and passwords to prime users
- The prime users can then use the system and program their devices. If they log in, their programs are saved to their identity
- Facilitator asks prime users to edit their preferences to set the name the facilitator knows them by. (For example Adam or Adam S)
- Facilitator asks prime users to edit their preferences to answer the short survey that's there as well. The answers to the survey are there as a last ditch approach to unlocking their account.
- The facilitator confirms their email with their username. Upon confirming sees the list of prime users they facilitate, including the names they know them by.

This scenario allows the facilitator to just give out the devices and run with it. It doesn't involve a lengthy registration period. The aim is to enable the prime users and facilitators first experience to be painless.

Failure Scenario: Fail to receive a sheet with usernames/passwords in a batch (A)

- They could re-request a new sheet – the sheet is in no way tied to any particular batch.

Failure Scenario: Fail to receive a sheet with usernames/passwords in a batch (B)

- Prime users can self register on the system
- The facilitator can self register on the system
- The prime users can add the facilitator's username
- The facilitator can add a username of another facilitator to confirm them (in a school this would be another teacher. The alternative is via the distribution chain)
- Once the facilitator is confirmed, the facilitator can then confirm they are facilitating those prime users.

Failure Scenario: Prime user forgets password

- Prime user goes to website, clicks forgotten password, enters username
- Facilitator is informed, goes to their profile, resets password
- Facilitator gives new password to prime user face to face

Failure Scenario: Prime user forgets username

- Prime user goes to facilitator, asks for username
- Facilitator gives prime user name face to face

Failure Scenario: Prime user forgets username, facilitator doesn't remember, prime user set up account

- Prime user forgets username, facilitator doesn't remember
- Facilitator logs into site
- Goes to the list of prime users they facilitate
- Finds the name they know the prime user by
- Gives prime user their username

Failure Scenario: Prime user forgets username, prime user programmed their device last, device based reminder

- User switches on device, holding down button B.
- Device switches to reminder mode
- Device scrolls the username stored in EEPROM

Failure Scenario: Prime user forgets password, no active facilitator

- Prime user goes to website, clicks on forgot password
- User selects the child friendly option that means “no active faciltator to reset password”
- Child is given survey based on the questions they answered when given the bug
- If they get “enough” correct, they are provided with a new passwords

Once again, the requirement here is to handle these failure scenarios gracefully, and the mechanisms described here are illustrative, not prescriptive.

2.8.2 Authentication

In this context, authentication means the assertion and verification of an identity within the system. Clearly there are many mechanisms that could be used here. It is preferable that users are able to use their preferred form of identity within the system, without losing the benefits described in the previous section.

In the previous section where one possible mode of operation exists for identity there is a working assumption of a particular form of authentication.

That particular sequence of illustrations assume:

- That usernames/passwords can be pre-generated
- That they can be associated with identity roles within the system
- That relationships between identity roles within the system can be dealt with.

The illustrative solutions to the scenarios presented there are highly desirable because they have the key benefits:

- No knowledge of prime user identity
- Takes advantage of the pre-existing face to face relationship between mentors and mentees
- Takes advantage of the fact there is a device to help the user authenticate themselves.

Clearly these identity aspects can be implemented in many ways. An ideal implementation for authentication would enable prime users and facilitators to asset other online identities match their identity in the system such they can be then authenticated using an external system.

For example schools have online systems already. While the device is clearly the prime users’ device, not the schools, being able to assert a school identity is also your identity would be useful since then the prime user could assert their identity using their school login.

Furthermore, if the prime user is older than 13, they (like facilitators) may have other services that they would wish to use to assert their identity. These potentially include service like Google+, Facebook,, and OAuth2 based services – such as twitter.

Use of such services should not be enabled for prime users younger than 13 for legal reasons, and also to not encourage prime users to use such services they should not have access to.

It is likely therefore that any implementation of identity may have a fall back implementation which accompanies the distribution of devices – to bootstrap a users identity within the system. As time progresses they become able to assert other preferred identities. These preferred identities could then be used to unlock their account – reducing the prime users dependency on the facilitator.

The authentication scheme chosen should be clearly described, including it's interface to other mechanisms to allow a preferred pre-existing identity to be asserted as matching an identity within the Microbug system.

Secondly since services the Microbug system uses may be implemented by multiple vendors on multiple systems, any implementation of authentication for microbug needs to support authentication for those services.

Specific non-exhaustive requirements:

- Ability to confirm the identity of a specific user within the system
- Ability to assist in the management of the relationship between user roles within the system
- Ability to asset an identity within the Microbug system matches another identity outside the Microbug system, and then to be able to identity as that Microbug identity through trust in that external system. External identity vendors include :
    - School authentication mechanisms (such as wayf etc.)
    - Youth organisation authentication schemes.
    - Common educational authentication schemes
    - Commonly used internet authentication schemes – such as Google+, Facebook ID, Microsoft Account (formerly Windows Live ID), OAuth2, etc. This is not an exhaustive set nor does it recommend one external ID system over another.
    - Pre-shared Public Key - Private Key systems etc.
- Able to assert to an sub-system implemented separately that the user using the sub system is a given Microbug identity, with a given role type. Microbug subsystems implemented independently may include:
    - Editing tools, apps and services
    - Storage systems
    - Tutorial systems
- Any implementation of authentication must match current best practice – it should not be practical for a user to steal another users account, nor should it be possible to derive user passwords from a compromised machine
- If pre-generated identities are used, it should be based on a mechanism that results in memorable ideas, and should not result in offensive identities. Any pre-generated passwords should be ephemeral and changed upon log on.

2.8.3 Authorisation

In this context, this means authorisation of tasks an authenticated identity is able to perform. Authorisation is as much about what is allowed as it is about what happens when authorisation is compromised.

A key aspects of authorisation in the system is that as far as possible compromising information is kept out of the system. This is due to the recognition that prime users are vulnerable, forgetful and that identity does get compromised in real world systems, despite best efforts.

In authorisation, we consider the following identities:

- Anonymous
- Prime User
- Facilitator
- Super Facilitator
- Authorised Edit Users (see later)

Specific tasks that may be performed:

- Register
- Login
- Logout
- Assert their identity as a facilitator to another facilitator
- Have their identity as a facilitator confirmed
- Confirm a facilitator is not a prime user
- Have their identity confirmed as a facilitator using a confirmation token or similar approach
- Edit new program
- Save program as public
- Save program privately
- Build/Download program
- Fork an existing program (create a new program based on another program)
- Edit existing program
- Browse all public programs
- Browse of my programs
- Ability to share a program by link
- Ability to shared a program by link that is editable by others
- Browse of programs I've contributed to
- Show user profile – which includes browse my programs / browse programs contributed to
- View as facilitator
- List users they facilitate include name they know users by
- View as prime user
- Reset user password
- View user settings
- View user's name set in settings
- View other data in settings
- Answer survey questions (to set other data)
- Set user's name
- Set email
- Confirm email
- Change password
- Create tutorials

- Edit tutorials
- View tutorials
- Follow tutorials
- Make tutorial visible for general use
- View an un-published tutorial
- Add facilitator
- Remove facilitator
- Assert an external identity is the same as the current identity

2.8.3.a Authorisation - Anonymous users

Anonymous users are simply those that have not logged into the system and not asserted an identity. Such users can perform the following tasks:

- Register
- Login
- Edit new program
- Ability to share a program by link
- Ability to shared a program by link that is editable by others
- Save program as public
- Build/Download program
- Fork an existing program (create a new program based on another program)
- Edit existing program
- Browse all public programs

2.8.3.b Authorisation – Prime Users

Prime users are discussed in previous sections. Such users can perform the following tasks:

- They can re-register as a new user if they're not logged in
- Login
- Logout
- Edit new program
- Save program as public
- Save program privately – visible to them and their facilitator(s)
- Ability to share a program by link
- Ability to shared a program by link that is editable by others
- Build/Download program
- Fork an existing program (create a new program based on another program)
- Edit existing program
- Browse all public programs
- Browse of my programs
- Browse programs I've contributed to
- Show user profile – which includes browse my programs / browse programs contributed to

    - Only their own profile
- View as prime user
- Reset user password – only their own
- View user settings – only their own
    - This means only the name their facilitator knows them by and age
    - Their gender is considered highly confidential and not displayed.
        - The reason for this is because gender can be male, female, prefer not to say, and other (e.g. intersex). The reason we collect gender is solely for statistical purposes. In particular, we wish to track whether the system is used more by one gender or another and if usage style differs. This is due to the ongoing issues around gender diversity in computing.
        - Once it's been set, we do not need to display it. It requires the user to opt in to set it. (It's not enforced) The prime users on the system are at an age where gender issues are becoming an issue for them, and (by definition) have immature attitudes. By not displaying it, if a user account is compromised, their gender identity is not.
- View user's name set in settings – only their own
- View other data in settings – not visible – these are essentially the answers to the survey questions used as security settings for unlocking the account. They can be changed, not viewed.
- Answer survey questions (to set other data) - only their own
- Set user's name - only their own
- Change password - only their own
- Create tutorials
- Edit tutorials
- View tutorials
- Follow tutorials
- View an un-published tutorial - only their own
- Add facilitator
- Remove facilitator
- Assert an external identity is the same as the current identity
    - Certain systems only, and only if they are over 13.

2.8.3.c Authorisation – Facilitators

Facilitators are discussed in previous sections. If the facilitator has not confirmed their user identity – by confirmation of their email they only have the authorisation level of a prime user – with the addition of functions necessary to confirm their email contact identity.

Such users can perform the following tasks:

- Re-Register on the system if necessary
- Login
- Logout
- Assert their identity as a facilitator to another facilitator
- Have their identity as a facilitator confirmed

- Confirm a facilitator is not a prime user
- Have their identity confirmed as a facilitator using a confirmation token or similar approach
- Edit new program
- Save program as public
- Save program privately
- Ability to share a program by link
- Ability to shared a program by link that is editable by others
- Build/Download program
- Fork an existing program (create a new program based on another program)
- Edit existing program
- Browse all public programs
- Browse of my programs
- Browse of programs I've contributed to
- Show user profile – includes browse my programs / browse programs contributed to
  - their own profile
  - profiles of those they facilitate
- View as facilitator
- List users they facilitate include name they know users by (from prime user profile)
- Reset user password
  - Their own
  - Those of accounts they facilitate
- View user settings
  - Only their own
- View user's name set in settings
  - Only their own
- Answer survey questions (to set other data)
  - Only their own
- Set user's name
  - Only their own
- Set email
  - Only their own
- Confirm email
  - Only their own
- Change password
  - Only their own (for other users they must reset it)
- Create tutorials
- Edit tutorials
- View tutorials

- Follow tutorials
- Make tutorial visible for general use
- View an un-published tutorial
    - Their own
    - Those created by those they facilitate (to enable them to check for identifying data)
- Add facilitator
    - This adds their super facilitator
- Remove facilitator
- Assert an external identity is the same as the current identity

2.8.3.d Authorisation – Super Facilitators

Super facilitators are a special case of facilitators who facilitate facilitators. They may also facilitate prime users. For clarity:

- A super facilitator cannot see or interact with the prime users that they do not directly facilitate
- A super facilitator's task is their primarily to reset the facilitators account if the facilitator gets stuck.
- Two facilitators may be super-facilitators for each other.

2.8.3.e Authorisation – Authorised Edit Users

Authorised users is a special case of permission. In the general case, even authenticated users cannot edit the same program. If one user creates a program, another user may fork it – that is create their own version based on the original users. However generally speaking only the original author can edit the same actual program.

In order to support group working, a second mode of behaviour can be enable – which created authorised edit users.

An authorised edit user specifically asserts the following identity:

- I am an individual who has been granted edit access to this code

The way this is asserted is as follows:

- When a program is being created, an edit password is generated and associated with the program
- When the program is shared by link, if the edit password is also shared, then the recipient of the link becomes an authorised edit user
- If the authorised edit user tries to edit the program, despite it not being "theirs", they are able to edit the program.

This identity is complementary to the other identities in the system. It allows anonymous users, prime users, and facilitators to work together editing the same program as a collective.

2.8.4 Identity Data Collected

The following identity data must be collected by the system:

All Users – demographic information:

- Their gender
- Their age.
- Neither of these are displayed, but they must be changeable

Prime Users

- The name their facilitator knows them by.

Facilitators (including super facilitators)

- The name they wish to be known by in the system
- Contact details for resetting the account:
    - Email
    - Mobile (SMS) - for 2 factor authentication is required by the authentication platform

Additional limited identity data may be collected by the system for the purposes of unlocking the user account in the event the user forgets their password, and has no active facilitator. It should not be possible to uniquely identify an individual from such data.

Such data should **not** be traditional identity security information. For example, favourite colour is preferable to "mother's maiden name". This could mean collecting more than one response in order to secure an account from malicious takeovers.

2.9 Front End Types

The following sections on Website Overview , Website Core, Compiler Services, Storage Services, Upload/Download services, combined with the sections relating to browsers and loaders are all systems that interact together to form the front end of the system.

For simplicity's sake the specification here assumes the following breakdown of location of services:

- Website Core – Web server
- Website Services – Web server
- Compiler Services – Web server
- Storage Services – Web server
- Upload/Download Services – Client side to/from web server
- Editing – Client side
- Loader application – Client side

That model of operation assumes a traditional connected model setup consisting of:

- Client side device with common operating system and standard browser
- Loader application running client side
- Server or servers running appropriate web services

A modern system may be like that, but disconnected models of operation are also required.

Specifically:

- It should be possible to use the Microbug system without any form of network connection, or without a continuous/reliable connection
- Ideally the user should be able to use an app on a tablet device, or similar, to use the Microbug system - either in a connected mode or in a disconnected mode. (or both)

In such scenarios, the location of each of these services and provision could well be in different locations.

The core proposition does assume the traditional connected model setup described above, but it is desirable for the system to also support the disconnected front end types described above.

These will typically take on aspects of various parts of the user experience, and either shift them client side, and/or into an app.

Apps and disconnected mode systems should state which parts of the system their app or disconnected mode system will support.

2.10 Website Core

The Microbug website is the key entry point to the Microbug system for the user.

Specifically, it consists of the following core components:

- Core site pages, server infrastructure, templates, styling, assets, css, etc.
- Identity – including authentication, roles, etc. as described in section 2.8
- Program editing & simulation
- Browsing of programs
- Tutorial support – including stepped/sequential tutorials
- Code generation by client side javascript
- Website services – including storage/compilation etc. See section 2.11

The 2 key functional requirements for the website are:

- Each item above should be possible to be implemented by any partner, and in many cases, many bullet points should be possible to be be implemented by many partners.
- To an end user this should appear to be a single conceptual proposition. The Core Website is the central core around which all the other elements are added.

The Core website should have minimal integral functionality, and should focus on aggregating the other aspects of the system – both functionally, and from a UX viewpoint. Specifically:

- It should provide core assets for use inside the proposition as a whole, including branded assets.
- It should provide the core infrastructure for navigating between various aspects of the site.
- It should provide core infrastructure for selecting appropriate editors on the system

2.10.1 Browsing

Browsing programs created by users of the system is a primary mechanism of discovering content on the system. Each front end system that provides an editor will need to provide a means of integrating with the browse implementation. This can be two core mechanisms – either separate browse implementations launched by the core implementation, or as pluggable alternatives for browsing.

The obvious mechanism to achieve that is for each browse repository to provide a browse API, and for the browse front end to query each of the browse repositories for content to display. If this is the case, in keeping with all other APIs in the system, the API **must** be a json oriented RESTful service.

It is a requirement on the implementer of core browsing to provide this mechanism for content from a variety of editors to be integrated into a coherent whole.

When a user selects a program from browse mode, they are taken to the editor that created that program. The user should not need to re-register or re-log in, etc. They can then use that editor as normal.

Browsing content can also mean browsing tutorials. Each tutorial form will relate to an editor of some kind. Picking a tutorial implicitly sets a preference for that editor.

2.10.2 Editing

The system must support the following:

- At least one graphical editor – at least one such editor must contain a visual language which resembles that which is being used by schools – to allow transferable skills. Each graphical editor needs to be able to convert any internal data format it uses to python.
- At least one text based editor, capable of creating python
- The user must be able to select an editor and use that
- The system should not favour or preclude any one editor over another – though it should operate within the preferences of its users.
- As noted in browsing, following through a tutorial which uses a given tutorial implicitly selects a preference for that editor type.
- The user must be able to switch between editors as they see fit.

Any editor that is used must be able to generate python, from its an internal format. This is particularly the case with graphical editing systems.

The editor may assume that the web server will store a single blob of data for it.

There must be at least one graphical editor capable of driving the Microbug simulator.

The editor should use the Microbug storage services for storing & retrieving programs the user edits.

2.10.3 Simulator

The Microbug site must have a simulator capable of being driven other parts of the system – in particular by editors.

This simulator must be a a visual representation of the Microbug device. It should have it's own appropriate javascript implementation of the device abstraction layer. When functions in the javascript device abstraction layer are called, the functionality those DAL elements represent must be activated.

As a result, this allows the editor to step through the code, visually highlight the code, and call appropriate functions within the simulator, allowing the user to preview the device's behaviour with given code.

The simulator is **not** an emulator. That is it is **not** required to be able to take a .hex file and execute that. What it is required to do is to be able to be driven by a graphical editor.

2.10.4 Tutorial Resources

The Microbug site MUST have tutorials, that assist the user in learning how to use their device. Given that visual programming tools differ in visual language, and differ from textual based languages, each editor MUST provide at least 1 tutorial that users that editor – ideally more.

Given the requirement for multiple editors there is clearly a requirement for to support many alternative sets of tutorials. As with browsing the obvious solution here is for their to be a single mechanism for browsing and displaying tutorials, and for this to be populated via RESTful json queries to a tutorials API for each type of editor.

It is preferable for tutorials to be active – that is for them to be linked to an editor. The tutorial system should be able to launch a preconfigured editor. Once that is done, it should be able to determine when code inside the editor matches the expected outcome from the tutorial.

The tutorial infrastructure should support:

- Paginated tutorials

- Tutorials from a variety of tutorial APIs – one per editor/browse system
- These may be active tutorials that can query a given tutorial API for progress. While this is expected to relate to checking programs created against expected solutions this could equally be used by tutorial systems for writing simple quizzes to confirm knowledge.
- Support for creating tutorials within the Microbug system from using a browser is desirable, as is the ability to create tutorials offline for upload onto the site- this includes the upload of assets and resources.
- Tutorials should be possible to be created by any user. However, tutorials written by prime users should be checked by facilitators before being made public.

Tutorials form the core resource for learning on the system, and are a crucial requirement to be supplied with any given code editor/browser.

2.10.5 Code Generation from a visual editor

Visual/graphical coding systems have not as yet solidified on a single visual language, or internal representation. As a result, each graphical system essentially implements its own visual language. Some are more similar to each other, some are radically different. Most do not provide a means of compiling their code for an embedded system.

The route for compilation onto the device is as follows:

- User edits a program using a graphical editor
- They save their code onto a server in python
- The python code is compiled to C++
- The C++ code is compiled for the target embedded device
- The uhex file is downloaded and loaded onto the device.

The key step in this setup is to generate python code in the visual editor.

There is also an alternative route:

- User edits a program using a graphical editor
- They save their code onto a server in C++
- The C++ code is compiled for the target embedded device
- The uhex file is downloaded and loaded onto the device.

This however requires the creator of the visual editor to also be able to create C++ cleanly. Doing this correctly is harder than python, and may also result in the editor seeking to use language subsets not supported. The python subset supported is the preferred interface because the language supported is well defined.

As a result, any visual editor must be able to convert its internal data structure to python.

2.11 Website Services

The website provides and uses a number of services. These are:

- Identity Services
- Compiler Services
- Python Interface
- C++ Interface
- Storage Services

2.11.1 Identity Services

The system must implement identity services. This is to implement the identity model describe in section 2.8 Identity. In particular, it should do this:

- Assert the users identity to the browse/edit and related elements of the Microbug system implemented on third party systems
- Enable the user the associate and asset their identity to the system with a third party authentication mechanism.

2.11.2 Compiler Service

The system must implement a compiler service in two ways:

- When programs are saved on the server, they should be speculatively compiled. This is to speed up the UX flow.
- It should be possible to send a program to the compiler service for compilation and to download the resulting .hex file.

The compiler service should operate the following way:

- When a program is saved or explicitly sent for compilation, the program should enter into a pending compilation queue
- When the compilation is complete, the resulting .hex file and .py files should be moved to a “competed” queue/location.
- While the program is in the pending queue, it should be possible for a client to query the compilation service as to whether the program is compiled
- Once the compilation is complete the compilation service should allow the user to download their compiled code.

The compiler service should a json oriented RESTful service.

2.11.3 Python interface

The python interface is the primary compilation target in the system. Supporting general python is not practical on the small scale microcontroller. As a result, it is necessary to define the python subset that is available. The python subset that has been chosen is based upon experimentation, and observation of how beginner programmers use python. As a result it allows many of the basic types of programs a user will implement.

This section specifies as unambiguously as possible which parts of python are and are not supported.

2.11.3.a Language Features to be supported.

Overview of python to be supported by implementations:

- Value literals: Integers, String, Boolean
- Identifiers
- basic assignment statements – i.e. identifier equals expression
- print statements
- expression statements
- expressions – specifically:
    - Those involving value literals, identifiers and function calls
    - Infix expressions involving * + / -
    - Parentheses ( ) for nested expressions.
    - bitwise operators, logical operators, boolean operators
- comments

- Loops - while / for
    - forever_loop (while True)
    - while takes an expression for the condition
    - for takes an identifier for the iterator, and expression to be iterated over. The expression is treated as indexable thing with a length. A range() function call may be detected and treated as a special case.
- break / continue statements
- If statements including elif and else clauses
- function calls with expressions as arguments
- function definitions with an optional argument list
- return statement
- Lists, list literals
- Library functions to be treated as builtins consist of the functions described in the device abstraction layer
- Dictionaries, dictionary literals

The following are desirable:

- yield statements / generators (This can be compiled to C++ using Duff's device – based on prior tests)
- doc strings
- import statements, from...import.. statements
    - What this would actually mean in the context of a device would need further discussion with an implementer, since what is available depends upon the device.

2.11.3.b Language Features not supported.

As important as the list of language features that are to be supported is, clarity on language features not supported is just as important.

This is NOT an exhaustive list. If an implementer wishes to support these, this list could shrink. If there are pedagogical reasons for implementing these we welcome that feedback.

- General assignment statements -
    - No unpacking of identifier lists – i.e. no x,y,z = <rhs>, nor x,(y,z) = <rhs>
    - No augmented assignment – e.g. no += -= *= and so on
- exceptions – exception values; raise statements; try, except, finally, else blocks. (maybe later)
- classes (and associated)
- operators: ** (power), ~ (bitwise negation), modulo, //, string templates via modulo operator, shift operators, conditional expressions
- Operator precedence does not as yet match Cpython spec
- Line continuation
- parentheses for line continuation
- String literal concatenation
- String concatenation over line continuations.
- function calls : named arguments, calling with *argv, **argd

- function definitions : optional arguments, named arguments, *argv, **argd
- redirected print statements
- generator expressions
- list/dictionary/set comprehensions, slices
- tuples, sets
- with statements
- decorators
- Octal, binary, hex literals
- long integers, float numbers, imaginary numbers
- back quoted string conversions
- variant string literal types. (include r'' strings)
- Escaped strings (for now)
- Objects / object attribute access
- else clauses for while / for loops
- for does not support unpacking of iterated objects – i.e. for x,y,z in <thing> is not supported
- for does not support single line nesting (for x in y for y in z)
- yield expression parsing
- implementation of yield
- generators, asserts, del statement
- modules – including:
    - importing modules, importing names from modules, relative imports
- future statements
- exec statement, eval expressions
- global statements, nonlocal statements
- The python standard library is not available

2.11.4 C++ Interface

The C++ interface is not a primary network service in the system. However given the python code is compiled to C++ before compilation for hardware, it is reasonable to allow some editors to generate C++ if that is more convenient. If an editor does use the C++ interface, it needs to ensure that it follows the requirements in the section 2.5.1.j on compiled programs in the dal.h.

2.11.5 Storage Services

The storage service has the following requirements:

- When a program is stored on the service:
    - What is actually stored is a version.
    - If this is the first version, a program metafile is also created and updated to point at the version.
    - If this is not the first version, the program metafile is updated to point at the new version. The new version contains a link to the previous version.
    - This applies even in the case of forked software

  - The version contains the userid of the user who edited the file.
- This creates a situation whereby versions are immutable, but programs appear mutable.
- Immutable versions also have the property that they are final. If they're final they can be compiled immediately. The compilation service may or may not be configured to do so.
- The storage service should accept a json blob that contains the following core data:
  - The editor representation
  - A python (or C++) representation of the code
  - Possibly more data.
- Storage on the server should trigger any other services reliant on changed data files.

2.12 API

All services in the Microbug system should be usable as web services.

The key requirement for these APIs is that they must be accessible using a RESTful json oriented API.

It is a requirement of all implementations of services within the Microbug system that they also implement an API. This is to allow a multitude of alternative interfaces to the system. Implementation of these APIs is left to the implementers of these websites.

2.13 Tethered Mode

The purpose of tethered mode is to allow direct control of devices attached to I/O pins and the LED display. It also allows integration of devices into a wider device environment, and in particular should allow the user of tethered mode to explore and create IOT type systems.

To this end, the IOToy library (previously created by BBC R&D), is suggested as the path of least resistance for implementing this. Key features:

- Apache 2 licensed (open source, minimal restrictions, no impact on user's IPR)
- Cookie cutter implementation designed to work on microcontrollers. Tested on Atmel 8A micro-controller, Texas Instruments MSP430, and more capable devices.
- Device library mocking to enable off-device testing – allowing round-trip testing off device.
- Device introspection over a serial API in a standardised fashion, allowing automated proxy creation – essentially providing tethered mode
- This API allows automated creation of RESTful proxies
- The RESTful proxies may also be auto discovered and then used transparently by clients

Thus once a device is tethered using the IOToy library, it can be used by any other device in the locality, rather than just the machine it's tethered to.

The license for IOToy is included in the appendix.

Use of the IOToy library itself is not a requirement, however the specification for tethering follows the IOToy protocol specification. In part the rationale for this is that even if you ignore the IOT related aspects, the serial introspection protocol is usable using a simple terminal application.

This makes a Microbug a usable device to a user even without access to the web service – i.e. completely standalone.

2.13.1 Tethered mode – Serial Protocol

The serial protocol for tethered mode follows:

All messages from the device to the client have the form:

```
<status code>:<Human readable status>:<data>
```

Below --> means "sent to device", <-- means "received from device".

Status code - a numerical value, based on HTTP status codes

Human readable status - can be passed through to people

data - intended to be parsed by machines

Switch on:

```
        --> (nothing sent)
        <-- 200:DEV READY:<device id>
```

Device id may be one of the two following options:

- plainid
- plainid ":" specifierid

The plainid must be something that can be used in DNS. A good rule of thumb for this is that it should match the following regex: [a-zA-Z][a-zA-Z0-9_]*

The specifier id, if used, must also be something that can be used in DNS, and it should conform to the regex [a-zA-Z][a-zA-Z0-9_]*

The idea is that the plainid is generally expected to refer to a class of devices, and the specifier id would refer to a specific device.

Base function:

```
--> help
<-- found:help -> str - try 'help help', 'funcs' and 'attrs
```

This enables discovery of functions:

```
--> funcs
<-- 200:FUNCS OK:ping,funcs,attrs,set,get,help,devinfo
```

For the an example device this this looks like this:

```
--> funcs
<-- 200:FUNCS  OK:ping,funcs,attrs,set,get,help,forward,backward,left,right,on,off
```

Data format is a list of tokens.

General form: --> funcs <-- 200:FUNCS OK:

Where funclist is a comma separated list of names of the form [a-z][a-z0-9_]* with no spaces. NB: No capitals.

Discovery of attributes:

```
        --> attrs
        <-- 200:ATTRS OK:drive_forward_time_ms:int,turn_time_ms:int
```

General form: --> attrs <-- 200:FUNCS OK:

Where attrlist is a comma separated list of . Each attrspec has the following form:

```
        attrname ":" attrtype
```

Note:

- attrname is of the form [a-z][a-z0-9_]* with no spaces. NB: No capitals.
- attrtype must be one of "str" "int" "float" or "bool"
    - No support for structure at present. Some variant of binary json will be specified however at some point after an evaluation of various options has taken place.

Note the data format here is a list of tokens and types. Valid types - in terms of current implementations are str and int initially - may expand to basic json types.

This allows the extraction of attributes and checks that the values passed down to the C API are of the right type.

In order to allow devices to be self documenting, help can be provided for attributes:

```
--> help drive_forward_time_ms

<-- 200:Help found:int - How long to move forward

...

--> help turn_time_ms

<-- 200:Help found:int - How long to turn
```

Note the data format here is:

```
<type> " - " <description>
```

Similarly, help for functions is there to aid introspection and self documentation. The general form is this:

```
--> help <funcname>

 <-- 200:Help found:<funcname> <funcspec> - Description
```

Where:

- funcname is of the form [a-z][a-z0-9_]*
- funcspec has the form: " -> "
- argspec may be empty
- result spec may be empty
- If argspec is not empty it has the form: ":"
- If resultspec is not empty, it has the form ":"
- name - must be [a-z][a-z0-9_]*
- type describes the type of the argument/result and may be the following:
    - str - string - no quotes should be used
        - This seems awkward, but remember this would set a string to empty:
            - "set name \n" As opposed to non-empty:
            - "set name .\n" You can't have an empty attribute or function name.
    - int - may be up to +-2**31. If you need more, use more attributes
    - float - IEEE 754 double
    - bool - valid representations of true: 1, true, True, t, T
        - valid representations of false: 0, false, False, f, F
    - T -- This is a special case meaning the type will vary. This sounds mad, until you realise that this is helps define "get" and "set" where the return or argument type depends on the attribute.

The upshot of this is that the following are valid responses for attributes and functions:

```
"200:Help found:int - How long to move forward"
```

```
"200:Help found:int - How long to turn"

"200:Help found:forward dist:int -> - Move forward for a distance"

"200:Help found:backward dist:int -> - Move backward for a distance"

"200:Help found:on -> - Turn on"

"200:Help found:off -> - Turn off"

"200:Help found:set name:str value:T -> - set an attribute to a value"

"200:Help found:get name:str -> value:T - return an attribute's value"
```

It should be relatively clear here that:

- on/off map to functions without any arguments and no result
- forward/backward map to functions with 1 argument and no result
- get maps to a function with 1 argument and a result
- set maps to a function with 2 arguments and no result - this is currently a special case and not generally supported
- The lack of "->" in the first two indicate they are response strings from attributes not functions - which means they only mention the type and human readable text.

2.13.2 Tethered mode – Manual usage

This essentially usage of the serial mode API of the IOToy library

2.13.3 Tethered Mode - local API usage

Local usage can use the IOToy library to introspect the device and then uses the device using the IOToy library's built in proxy. This then allows local usage relatively trivially.

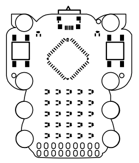
2.13.4 Tethered Mode - local API advertising

This is not directly part of the Microbug system but bt using the IOToy library, the device can be advertised onto the local network.

2.13.5 Tethered Mode - remote API access

Again, this is not directly part of the Microbug system but bt using the IOToy library, the device can be advertised onto the local network, enabling proxies for access and control of Microbugs to be auto created.

4 Appendix Reference Implementation



This appendix contains 4 main section types, all relating to the reference implementation.

These sections relate to:

- 4.1 : Hardware description
- 4.2 : Hardware font
- 4.3 - 4.8 : Software stack
- 4.9 : Licenses used within the subsystems of the software stack

Author: Michael Sparks, Senior Research Engineer, BBC R&D

4.1 Reference Implementation Hardware Description

4.1.1 Electronics

The reference implementation hardware is based around an Atmel ATMega 32u4 microcontroller. This has the following specification:

- 8 bit, 16Mhz Clock
- 2.5K Ram, 32K Flash
- 26 IO Pins
- Programmable using the Arduino tool chain compatible

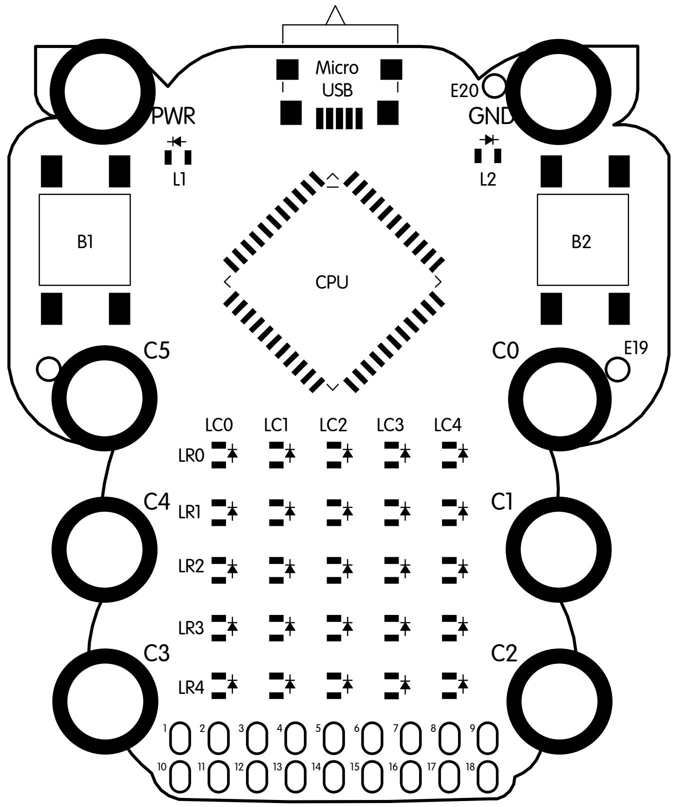
This decision was based primarily on device familiarity & pragmatics. Additionally, this represents an open tool chain, and availability of maker friendly resources, and community support. While not implemented in the reference implementation, this allows access to control of servos, many well known sensors, and also the creation of software serial circuits on arbitrary I/O pins allowing device to device communication.

The hardware schematic has been created using DesignSpark PCB. This was primarily chosen due to the fact that DesignSpark PCB puts no constraints on how designed created with the tool may be used or shared.

The reference implementation includes the necessary Gerber, Excellon, and pick/place x, y, rotation files in order to test the full stack, as a starting point for implementation.

As with the rest of the reference implementation, the hardware implementation is illustrative, not prescriptive. It does **not** confer any approval over any particular technology. If the final hardware implementation changes – for example to an alternate microcontroller – then an alternative implementation for the device abstraction layer would need to be written. (Indeed part of the purpose of the tight specification of the DAL is to allow this)

4.1.2 Device Physical Connections



The device has a 5 x 5 LED matrix, a couple of buttons, a couple of extra LEDs, a row of header holes, and 8 large crocodile clip rings. There is also support for an expert mode, which allows for emergency recovery of the device – through 2 extra data pins.

Between these, 22 I/O pins are accessible. 6 of which are easily accessible, and 14 are available on the header strip. No headers are in place in the header strip, but it is suitable for 0.1" pitch header pins to be added.

Input/Output Overview

- B1,B2 are buttons
- 2 Amber LEDs - L2, L2
- 8 crocodile clip connectors, 6 numbered C0 .. C5 & 2 for PWR and GND
- There is a matrix of Red LEDs – controlled by LR0..4/LC0..4
- There are two rows for additional headers, labelled 1 to 18
- 2 additional single point headers E19/E20, for emergency or expert level extension
- Micro USB connector for programming, power, control

I/O Attributes

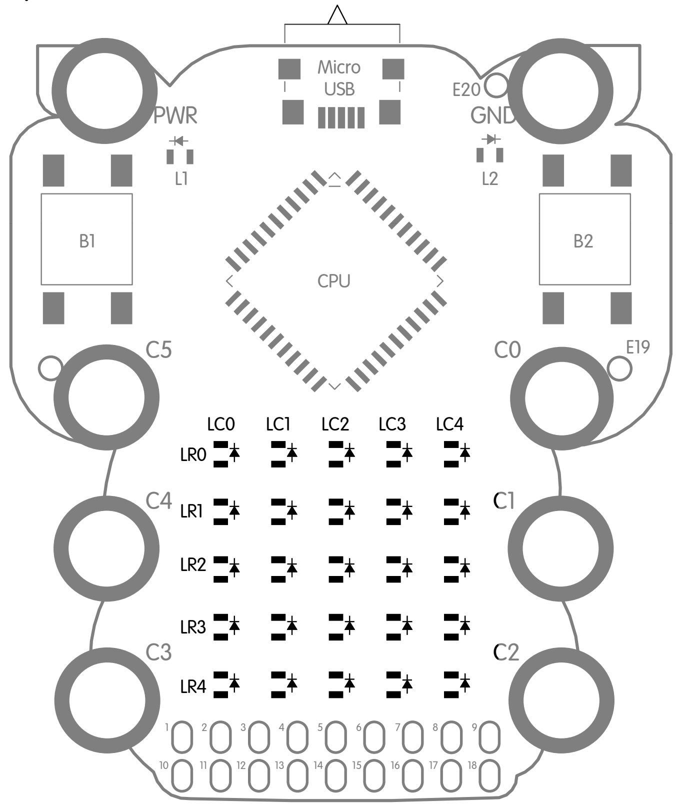
On the following pages each pin is described relative to it's attributes.

- **I/O ID** – This is the label on the diagram to the left
- **CPU Pin** – This is the pin the I/O is physically connected to
- **Dev ID** – This is the way the device is referred to internally to the implementation of the device abstraction layer
- **DAL ID** – This is the way the I/O is referred to within the DAL – how the user may use this.

Note:

- **I/O ID** and **DAL ID** are part of the technical specification
- **CPU Pin** and **Dev ID** pin are implementation artefacts of the reference implementation

4.1.2.a Physical Connections: LED Matrix



Input/Output Detail

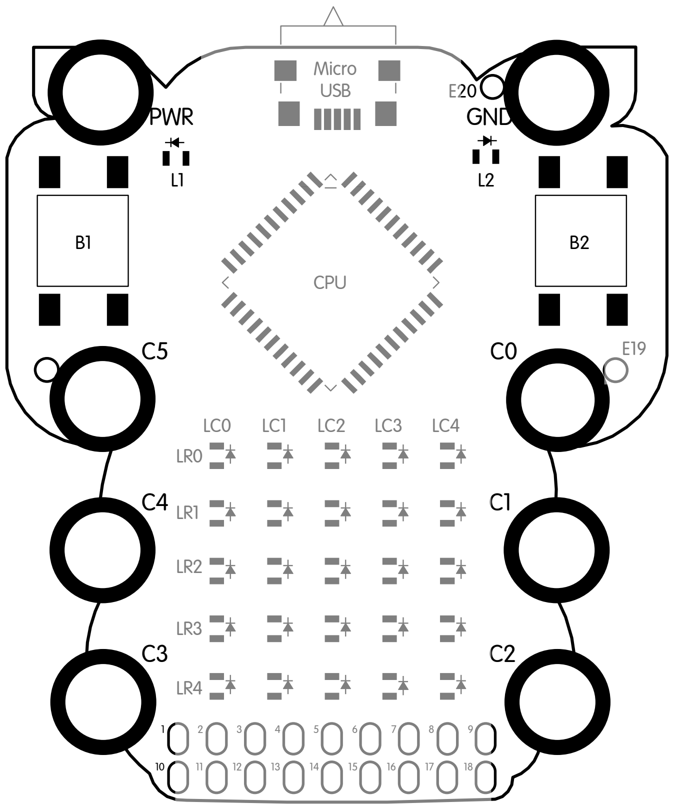
LED Matrix

I/O ID	DAL	Dev ID	CPU Pin	Description
LR0	**row0**	D1	21	Provides power LEDs on row LR0
LR1	**row1**	D0	20	Provides power LEDs on row LR1
LR2	**row2**	D2	19	Provides power LEDs on row LR2
LR3	**row3**	D3	18	Provides power LEDs on row LR3
LR4	**row4**	D11	12	Provides power LEDs on row LR4
LC0	**col0**	D4	25	Selects power for LEDs in column LC0
LC1	**col1**	D12	26	Selects power for LEDs in column LC1
LC2	**col2**	D6	27	Selects power for LEDs in column LC2
LC3	**col3**	D9	29	Selects power for LEDs in column LC3
LC4	**col4**	D13	32	Selects power for LEDs in column LC4

Notes:

- Generally speaking, users should NOT be accessing row/column IO's directly and should use the DAL. See the Device Abstraction Layer for more details.
- Switching on specific LEDs requires setting LC0..4 to LOW, and one or more LR0..4 to HIGH.
- Power on each row is limited by a 220 Ohm resistor

4.1.2.b Physical Connections: Buttons, Eyes, Croc Clips



Input/Output Detail

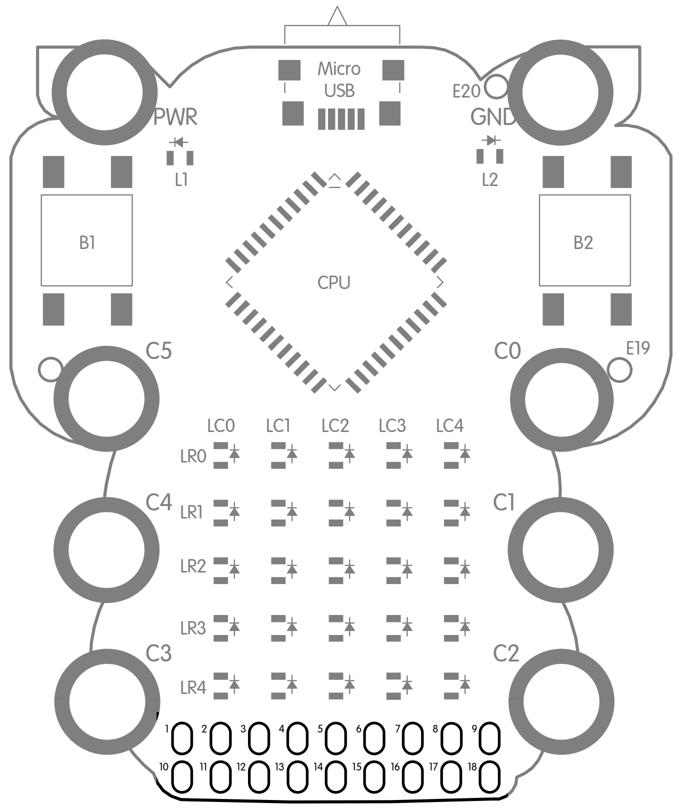
Buttons & Amber LEDs

I/O ID	DAL	Dev ID	CPU Pin	Description
L1	lefteye	D7	1	Provides power LEDs on row LR0
L2	righteye	D14	11	Provides power LEDs on row LR1
B1	ButtonA	D17	8	Provides power LEDs on row LR2
B2	ButtonB	D16	10	Provides power LEDs on row LR3

Crocodile Clip Connectors

I/O ID	DAL	Dev ID	CPU Pin	Description
PWR	-	-	VCC	Positive power connector. Maximum 5V.
GND	-	-	GND	Ground connection. (0V rail)
C0	croc0	A0	36	Analogue or digital input/output.
C1	croc1	A1	37	Analogue or digital input/output.
C2	croc2	A2	38	Analogue or digital input/output.
C3	croc3	A3	39	Analogue or digital input/output.
C4	croc4	A4	40	Analogue or digital input/output.
C5	croc5	A5	41	Analogue or digital input/output.

4.1.2.c Physical Connections: Headers



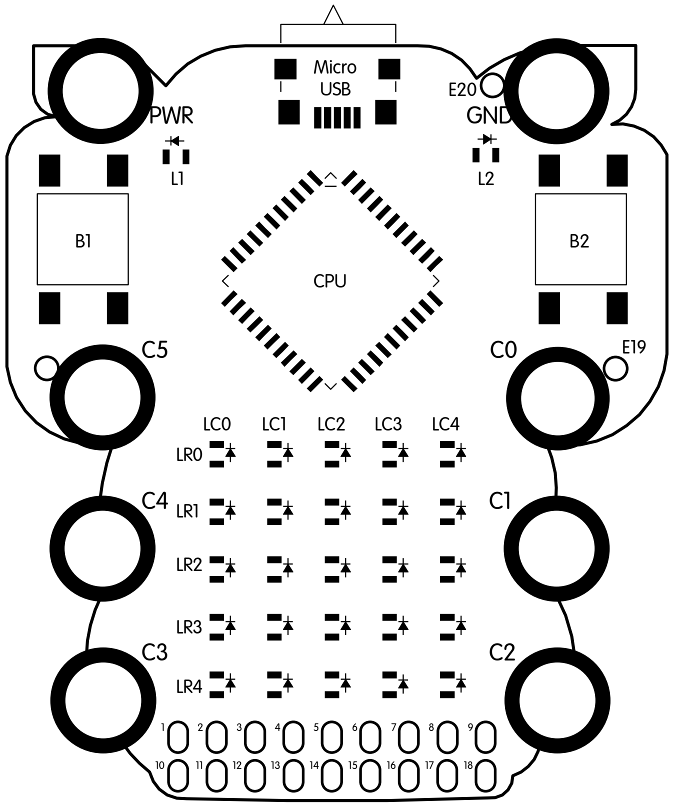
Input/Output Detail

Header Pins

I/O ID	DAL	Dev ID	CPU Pin	Description
1	-	-	13	RESET
2	-	-	-	GND
3	h1/row2	D3	18	PWM/SCL – connects to LR3
4	h2/col1	D12/A10	26	Also LC1/col1
5	h3/col2	D6/A7	27	PWM - connects to LC2
6	h4	D8	28	
7	h5/col3	D9/A8	29	PWM - connects to LC3
8	H6/col4	D13	32	PWM - connects to LC4
9	h7	D15	9	SCK
10	h8/row4	D11	12	PWM - connects to LR4
11	-	-	-	VCC
12	h9/col0	D4/A6	25	connects to LC0
13	h10/row1	D0	20	RX - connects to LR1
14	h11/row2	D2	19	SDA - connects to LR2
15	h12/row0	D1	21	connects to LR0
16	h13	D10	30	PWM
17	h14	D5	31	
18	-	-	33	HWB

Note:

4.1.2.d Physical Connections: Expert/Emergency IO Pins



Input/Output Detail

Expert/Emergency Device Pins

I/O ID	DAL	Dev ID	CPU Pin	Description
1	**-**	-	13	RESET (header pin 1)
2	**-**	-	-	GND (header pin 2 or croc clip GND)
9	**h7**	D15	9	SCK (header pin 9)
11	**-**	-	-	VCC (header pin 11 or croc clip PWR)
E19	**ee19**	D16	10	MOSI/PDI **(also, Button B)**
E20	**ee20**	D14	11	MISO/PDO **(also, right eye)**
18	**-**	-	33	HWB (header pin 18)

Note:

These pins are likely to be only useful in the following scenarios:

- Someone has uploaded a program to their device which they made themselves without using the Microbug development environment, and that does NOT check for an external factor on start-up, preventing reprogramming the device. (Requires RESET/HWB)
- They wishes to replace their device's bootloader via ISP
- They wish to program their bug using another bug via ISP

The latter two require essentially an ISP, which can connect to the first 6 pins

4.2 Font

The font used by the device abstraction layer is based very loosely on the font called “Tiny” by Matthew Welch. This is a free font with a permissive, BSD/MIT style license. The “source” for tiny however is a TrueType font, so while there are similarities, there Tiny was more used for inspiration than a directly used.

Reference: http://www.squaregear.net/fonts/tiny.shtml

License: Standard BSD License (A copy of Tiny’s license is included in 4.9)

Extensive changes have now taken place to the font. The following subsection defines the font to be used:

4.2.1 DAL Reference font

The definition of the DAL reference font follows. In the following _ means no pixel, and @ means a filled pixel. This is the autogenerated from the reference implementation. This font is a 4x5 font to allow a single strip of pixels between characters.

32 ____ ____ ____ ____ ____	33 _@__ _@__ _@__ ____ _@__	34 @_@_ @_@_ ____ ____ ____	35 @_@_ @@@@ @_@_ @@@@ @_@_	36 _@@_ @@_@ _@@_ @_@@ _@@_	37 @__@ __@_ _@__ @__@ ____	38 _@@_ _@__ @__@ _@@_ ____	39 @___ @___ ____ ____ ____
40 ___@ __@_ __@_ __@_ ___@	41 @___ _@__ _@__ _@__ @___	42 ____ @_@_ _@__ @_@_ ____	43 ____ _@__ @@@_ _@__ ____	44 ____ ____ ____ _@__ _@__	45 ____ ____ @@@_ ____ ____	46 ____ ____ ____ _@__ ____	47 __@_ __@_ _@__ @___ @___
48 _@__ @_@_ @_@_ @_@_ _@__	49 _@__ @@__ _@__ _@__ _@__	50 @@@_ ___@ _@@_ @___ @@@@	51 @@@@ ___@ __@_ @__@ _@@_	52 @___ @_@_ @_@_ @@@@ __@_	53 @@@@ @___ _@@_ ___@ @@@_	54 _@@@ @___ @@@_ @__@ @@@_	55 @@@@ ___@ __@_ _@__ _@__
56 _@@_ @__@ _@@_ @__@ _@@_	57 _@@@ @__@ _@@@ ___@ ___@	58 ____ _@__ ____ _@__ ____	59 ____ _@__ ____ _@__ _@__	60 ___@ __@_ _@__ __@_ ___@	61 ____ @@@_ ____ @@@_ ____	62 _@__ __@_ ___@ __@_ _@__	63 _@@_ @__@ __@_ ____ _@@_
64 _@@@ @_@@ @_@@ @___ _@@_	65 _@@_ @__@ @__@ @@@@ @__@	66 @@@_ @__@ @@@_ @__@ @@@_	67 _@@@ @___ @___ @___ _@@@	68 @@@_ @__@ @__@ @__@ @@@_	69 @@@@ @___ @@@_ @___ @@@@	70 @@@@ @___ @@@_ @___ @___	71 _@@@ @___ @_@@ @__@ _@@@
72 @__@ @__@ @@@@ @__@ @__@	73 @@@_ _@__ _@__ _@__ @@@_	74 @@@@ ___@ ___@ @__@ _@@_	75 @__@ @_@_ @@__ @_@_ @__@	76 @___ @___ @___ @___ @@@@	77 @__@ @@@@ @@@@ @@@@ @__@	78 @__@ @@_@ @_@@ @_@@ @__@	79 _@@_ @__@ @__@ @__@ _@@_
80 @@@_ @__@ @@@_ @___ @___	81 _@@_ @__@ @__@ _@@_ __@@	82 @@@_ @__@ @@@_ @__@ @__@	83 _@@@ @___ _@@_ ___@ @@@_	84 _@@@ __@_ __@_ __@_ __@_	85 @__@ @__@ @__@ @__@ _@@_	86 @__@ @__@ @__@ @_@_ _@__	87 @__@ @_@@ @@@@ @@@@ @__@
88 @__@ @__@ _@@_ @__@ @__@	89 @__@ _@_@ __@_ _@__ @___	90 @@@@ __@_ _@__ @___ @@@@	91 __@@ __@_ __@_ __@_ __@@	92 _@__ _@__ __@_ ___@ ___@	93 @@__ _@__ _@__ _@__ @@__	94 _@__ @_@_ ____ ____ ____	95 ____ ____ ____ ____ @@@@

96	97	98	99	100	101	102	103
_@__	____	@___	____	___@	_@@_	__@@	____
__@_	_@@@	@@@_	_@@@	_@@@	@__@	_@__	@@@@
____	@__@	@__@	@___	@__@	@@@_	@@@_	@__@
____	_@@@	@@@_	_@@@	_@@@	_@@@	_@__	@@@@
____	____	____	____	____	____	____	@@@@
104	105	106	107	108	109	110	111
@___	_@@_	__@_	@___	_@__	____	____	____
@@@_	____	____	@_@@	_@__	@@@_	@@__	_@@_
@__@	_@@_	_@__	@@@_	_@__	@_@@	@_@_	@__@
@__@	_@@_	_@__	@__@	_@@_	@_@@	@_@_	_@@_
____	____	@___	____	____	____	____	____
112	113	114	115	116	117	118	119
____	____	____	____	_@__	____	____	____
@@@_	_@@@	__@@	__@@	_@@_	@__@	@__@	@__@
@__@	@__@	_@__	_@@_	_@__	@__@	@_@_	@_@@
@@@_	_@@@	_@__	@@@_	__@_	_@@@	_@__	_@@_
@___	___@	____	____	____	____	____	____
120	121	122	123	124	125	126	
____	____	____	__@@	_@__	@@__	_@_@	
@__@	@__@	@@@@	__@_	_@__	_@__	@_@_	
_@@_	@@_@	_@__	_@@_	_@__	_@@_	____	
@__@	__@_	@@@@	__@_	_@__	_@__	____	
____	@@__	____	__@@	_@__	@@__	____	