Using Flowcode Web Developer to Communicate with Embedded Projects

Introduction

This worked example will show how to use Flowcode to create an embedded project that is controlled via a web app. The web app will also be developed in Flowcode using Web Developer, a new feature introduced in Flowcode v10.1.

This example will gradually be extended over 3 pairs of programs which gradually introduce the techniques used to communicate between the embedded and Web Developer apps. The initial example will create a web app that communicates to an embedded device to turn a light on and off. This will then be extended to optionally flash that light at two different rates and to read back data from the embedded device. And finally, the data returned from the embedded device will be converted to the commonly used JSON string format.

- Example 1
    - Sets-up the WebServer component
    - Shows connection to Wi-Fi
    - Uses basic “on” and “off” commands via HTTP URLs
- Example 2
    - Adds “flash” command using URL parameters
    - Adds “getvalue” command to retrieve data
- Example 3
    - All commands send using URL parameters
    - Data is returned as a JSON-encoded string

The hardware will comprise of an ESP32 module connected to a single LED. In the extended example, the module will also require a potentiometer to provide a user-controlled value to be read back.

In this example, you will see that the Web Developer project is generally much simpler than the embedded project. This is because the embedded project required additional code so it can communicate via the web, whereas a Web Developer project has most of this functionality in-built.

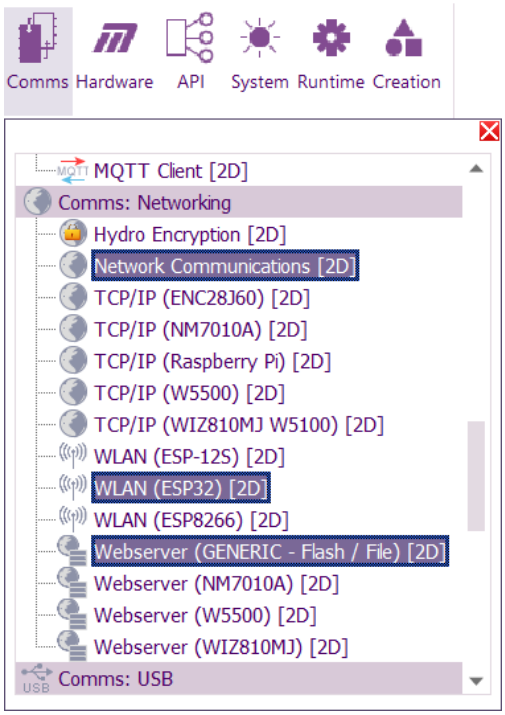
However, Web Developer projects are usually constructed in a different way to embedded projects due to the nature of web communication and program flow will rely on “event-driven” methods. Anyone who is used to programming embedded devices using linear procedural programming methods may find this confusing at first, even though it generally makes the programs easier to write.

Basic Embedded Example (EmbeddedProject-01)

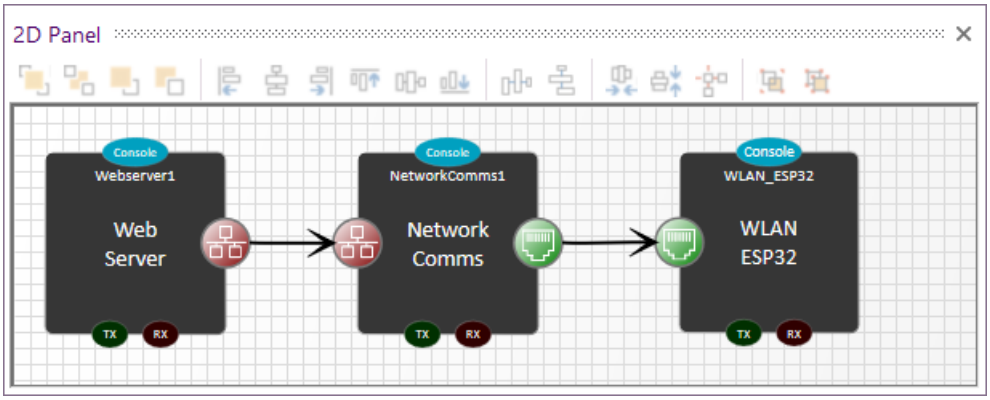
The goal of this initial basic example is to create a light that can be controlled via a web app. To do this, the embedded project must provide a web-based communication method to allow our web app to send commands to it. The easiest way to do this is for the embedded app to become a web server.

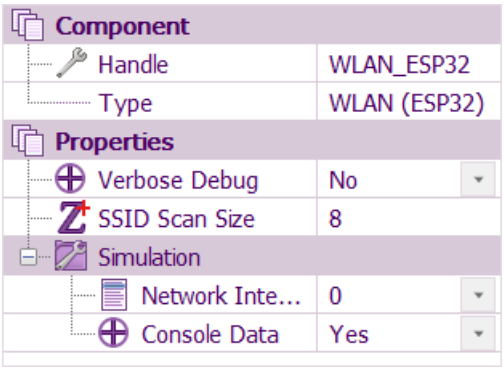
The embedded app will accept 2 commands – "on" and "off" – to control the light, which for simplicity will be an LED connected to one of the pins on the ESP32 module. These commands will be separate web pages that are accessed by our web app.

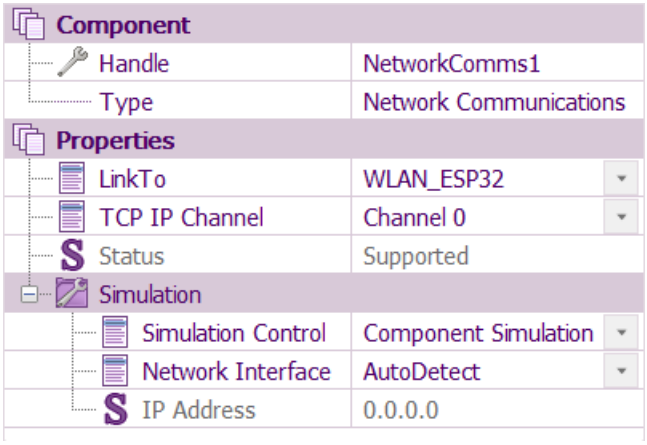
The first task is to create the web server, which is comprised of 3 components that are found under the "Comms: Networking" component library section as shown in the image below.

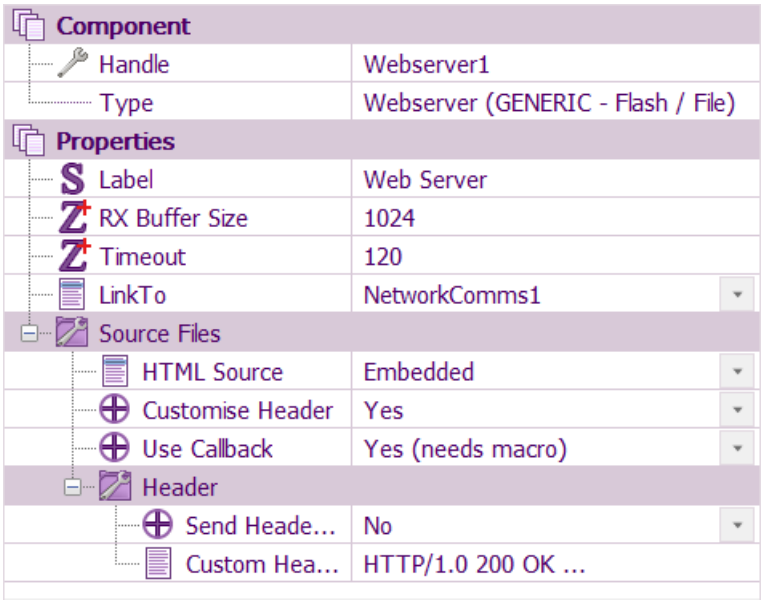


Drag each of these components onto a 2D panel and then set their properties so they are connected to each other. The component properties required are shown below.





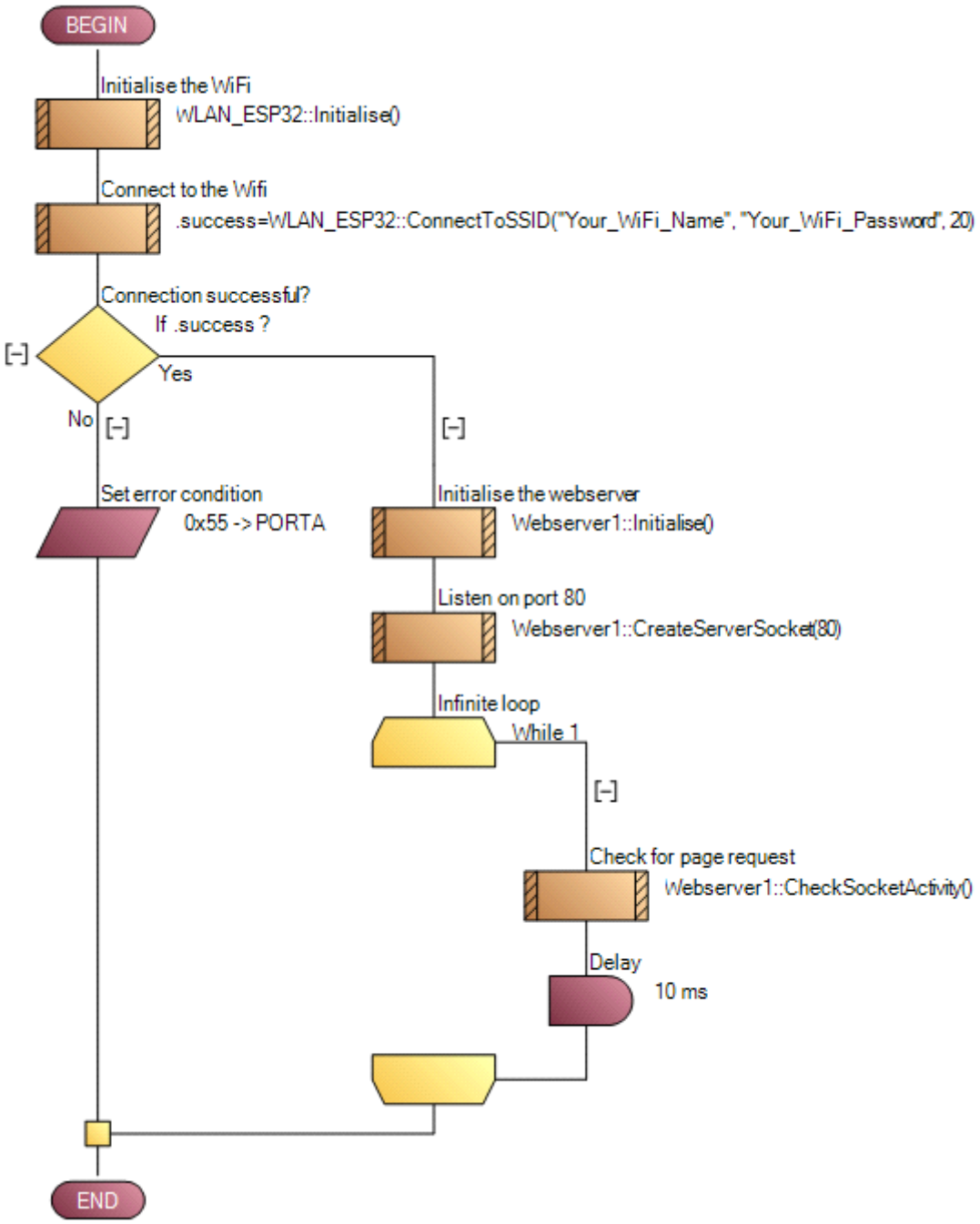




The “Custom Header” entry should have the following text:

```
HTTP/1.0 200 OK
Access-Control-Allow-Origin: *
```

The WebServer component is configured to use a “callback” function. This allows us to react to incoming HTTP requests and respond appropriately. More on this later, but first here is the code for the “Main” macro:



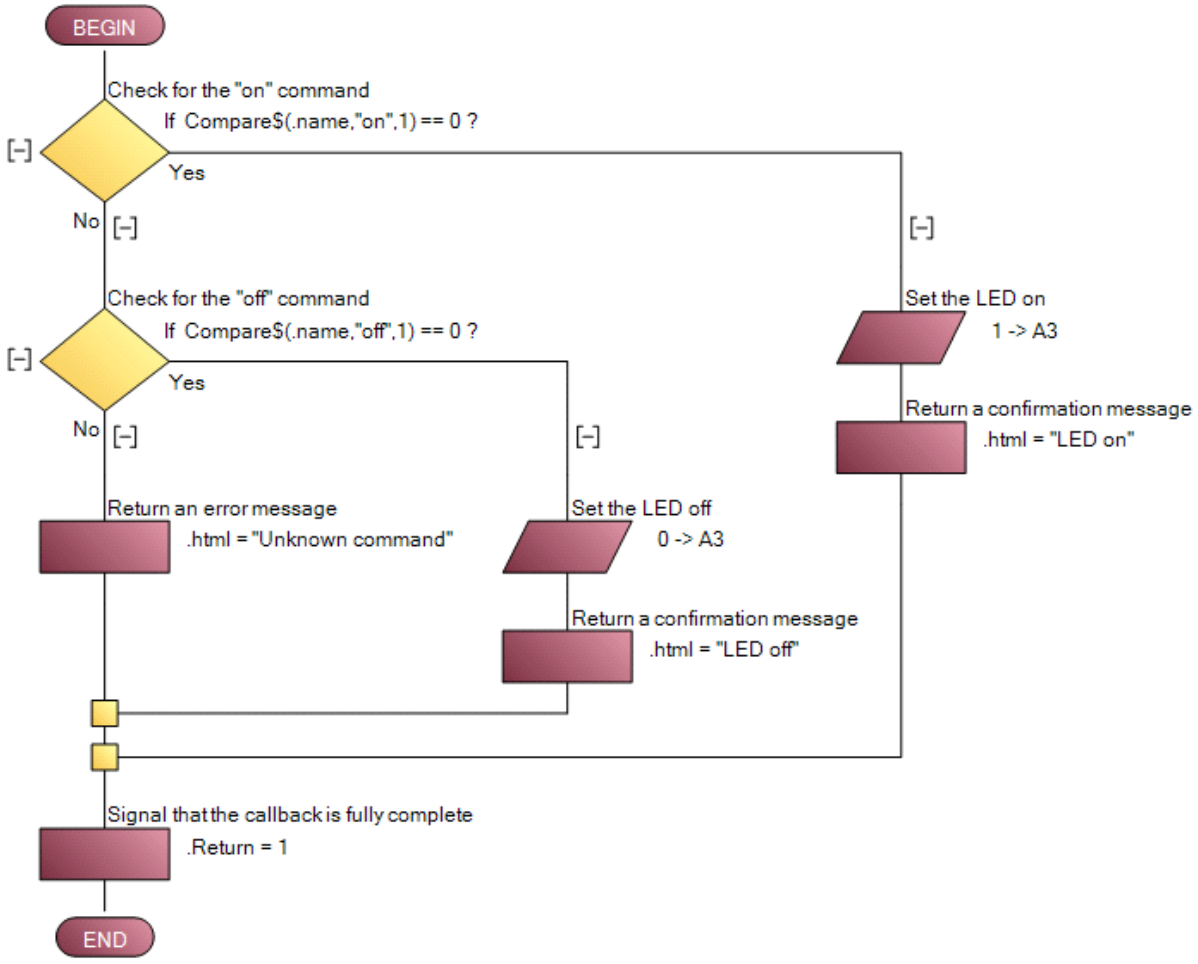
The first few icons initialise the ESP so it connects to the local Wi-Fi network. The name and the password for the network are parameters to the “ConnectToSSID” macro and these will need to be customised for the local network being connected to.

If connection is successful, the Webserver component is initialised and then an infinite loop is entered where the socket activity is checked on a regular basis to see if there are any external page requests. If there is a page request, a “callback” macro will be called.

The only other code in this initial example is the “callback” macro. This must be called “HtmlCallback”, return a BOOL type and have the following parameters:

name - STRING[32] - the name of the web page
params - STRING[128] - a string of parameters sent to the webpage
idx - UINT - an increasing number
html - byref STRING[256] - the returned html to send

This macro will be called whenever a page request is detected by the WebServer component.



The code checks the name of the page, which is essentially the command being sent from the web app. The two supported commands are “on” and “off”, which set the state of the LED appropriately.

This completes the basic embedded project. Click “Compile to Target” in the “Build” menu of Flowcode to create the embedded app and download it to the ESP32 target board. Once complete, reset the board and it should connect to the specified WiFi network.

You will need to determine the address of the embedded device on the network. This will take to form of an IP address that has been automatically allocated by your WiFi network. Once found, you should be able to switch the LED on and off by using a browser that is on the same network to visit the “on” and “off” pages using the address for your device. For example, if the device has an IP address of “192.168.0.87”, the pages to visit would be:

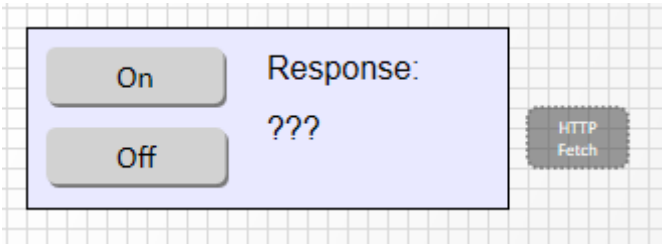
- http://192.168.0.87/on
- http://192.168.0.87/off

Basic Web Developer Example (WebProject-01)

Our goal now is to create a web app that will communicate with our embedded app to control the state of the light.

The app will consist of 2 buttons to turn the light on and off respectively, with a text field to display any returned text. A “Fetch” component is also required, which allows data to be sent to and received from a website. Since the embedded app has been designed to serve web pages, it is in effect its own simple website.

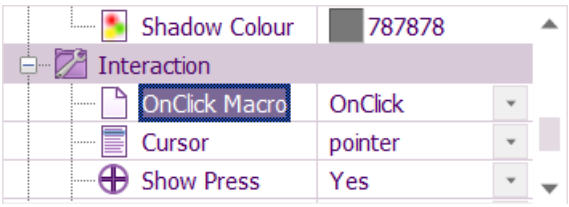
Here is the design of the basic web app:



There is very little code in this example, but the techniques used to develop this web app are different to a typical embedded app and so it is worth exploring some of these differences first.

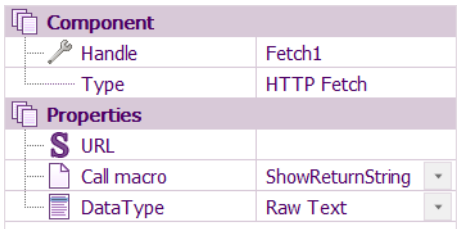
One major difference is there is no code in the “main” macro of this web app. Any code in the “main” macro is executed when the app is first loaded into a browser and it is important that this code is quick to execute because this “main” macro needs to finish executing before the app becomes responsive to the user. In fact, buttons and other UI objects will be inactive while code is executing in any macro, so code in every macro should be quick to execute. This is in stark contrast to an embedded project, where the code is always executing (even if it is waiting in a delay or loop).

The consequence of this is that user interaction generally initiates a macro. For example, each button in our project will execute a macro when pressed by the user. This is done via the “OnClick Macro” property of the button. The “On” button will point to a macro called “OnClick” and the “Off” button will point to a button called “OffClick”.

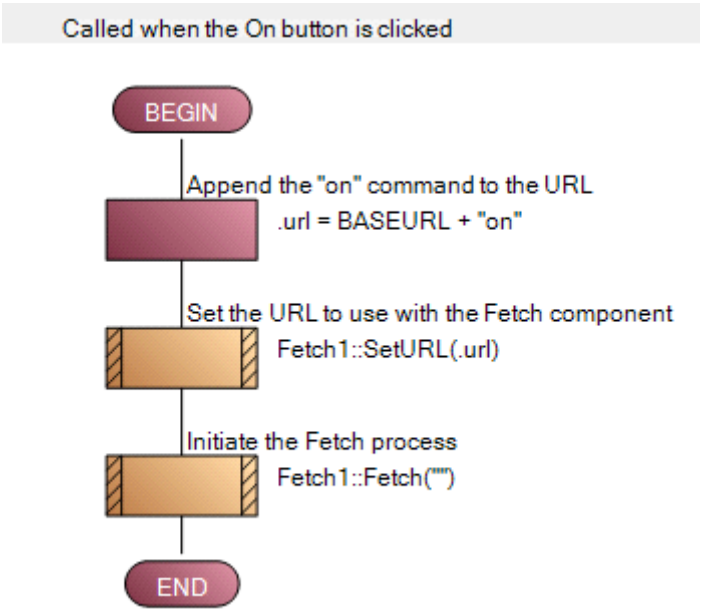


Any long operation, for example retrieving data from an external website, is often initiated within code and rather than waiting for this operation to complete, the macro will end and a different macro will be triggered when the long operation has completed.

This is illustrated by the “Fetch” component, which usually calls a macro when data is received from a website. Our basic web app will call a macro called “ShowReturnString” to display any returned message:



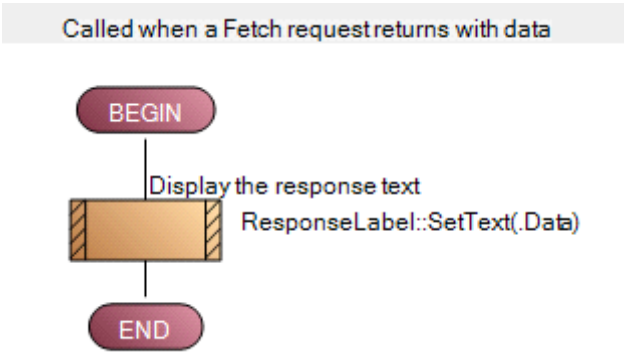
Here is the code within the “OnClick” macro, which is executed when the user clicks the “On” button:



The webpage (i.e. URL) to call by the “Fetch” component is created from a constant BASEURL and the command “on”, after which the actual Fetch operation is initiated. The macro then ends and when our app receives data back from this webpage, the “ShowReturnString” macro will be called. The “OffClick” macro is nearly identical but instead appends the command “off” to the BASEURL.

This BASEURL constant will be the IP address of our embedded device on our network. In the example above, this will be “http://192.168.0.87/”.

The “ShowReturnString” macro is shown below. This is a simple macro which takes a single string parameter named “Data” and sets the text of a label in our app to this string value.

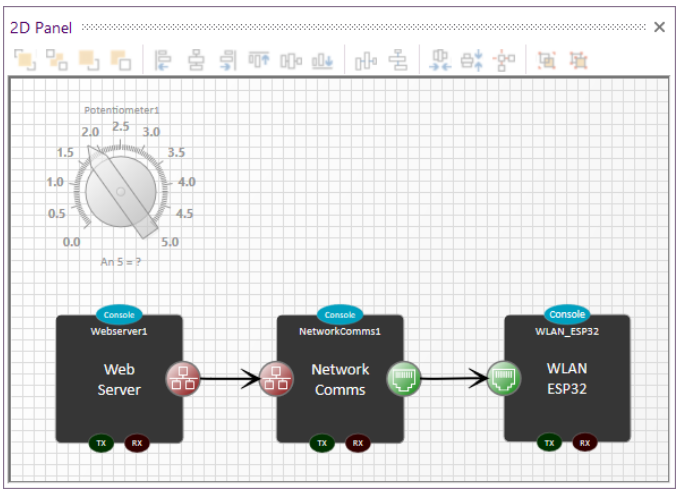


Clicking “Create web page” in the “Build” menu of Flowcode will create this web app. It is a .htm file which can be loaded into a browser on a PC, phone or any device which is connected to the same network as the embedded app. Once loaded in the browser, clicking the buttons in the app should switch the embedded app’s LED on and off.

Intermediate Embedded Example (EmbeddedProject-02)

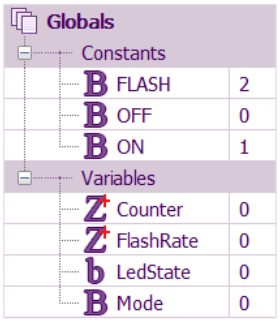
This will extend the basic project by adding functionality to flash the LED at 2 different rates and to read back a sensor value from the embedded device.

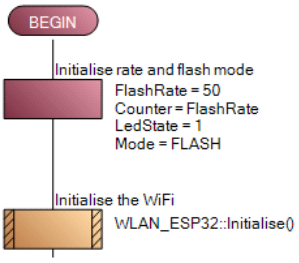
Starting with the Basic Embedded example, add a Potentiometer component to the 2D panel and set the channel to An5 (which is pin A1 of the device). This will be used during the HtmlCallback macro to return a variable analog value back to the Web app.



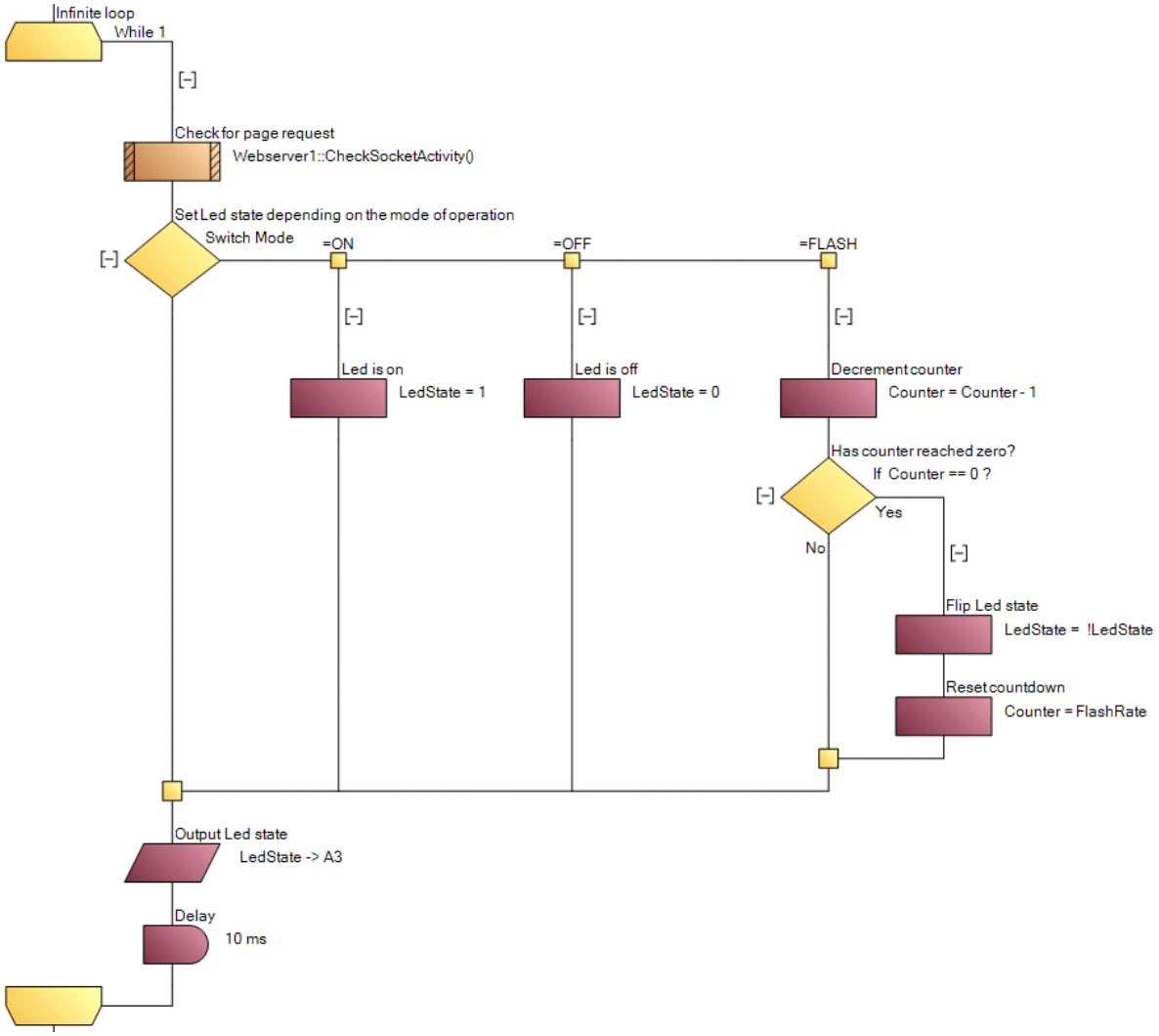
In the Basic Embedded example, the change to the LED state was made directly in the HtmlCallback macro as a response to the incoming command, but a different approach is taken here. The LED will instead be changed within the infinite loop of the Main macro depending on the value of a global variable which is altered when a command is received.

2 global variables – Mode and FlashRate – are used, and these are altered whenever an appropriate command is received by the app. Mode has 3 possible states identified by the constant values ON, OFF and FLASH, and the FlashRate is used to reset a counter to determine when to flip the state of the LED when in a 'flash' mode. 2 further global variables are used to regulate the LED flashing – Counter and LedState.



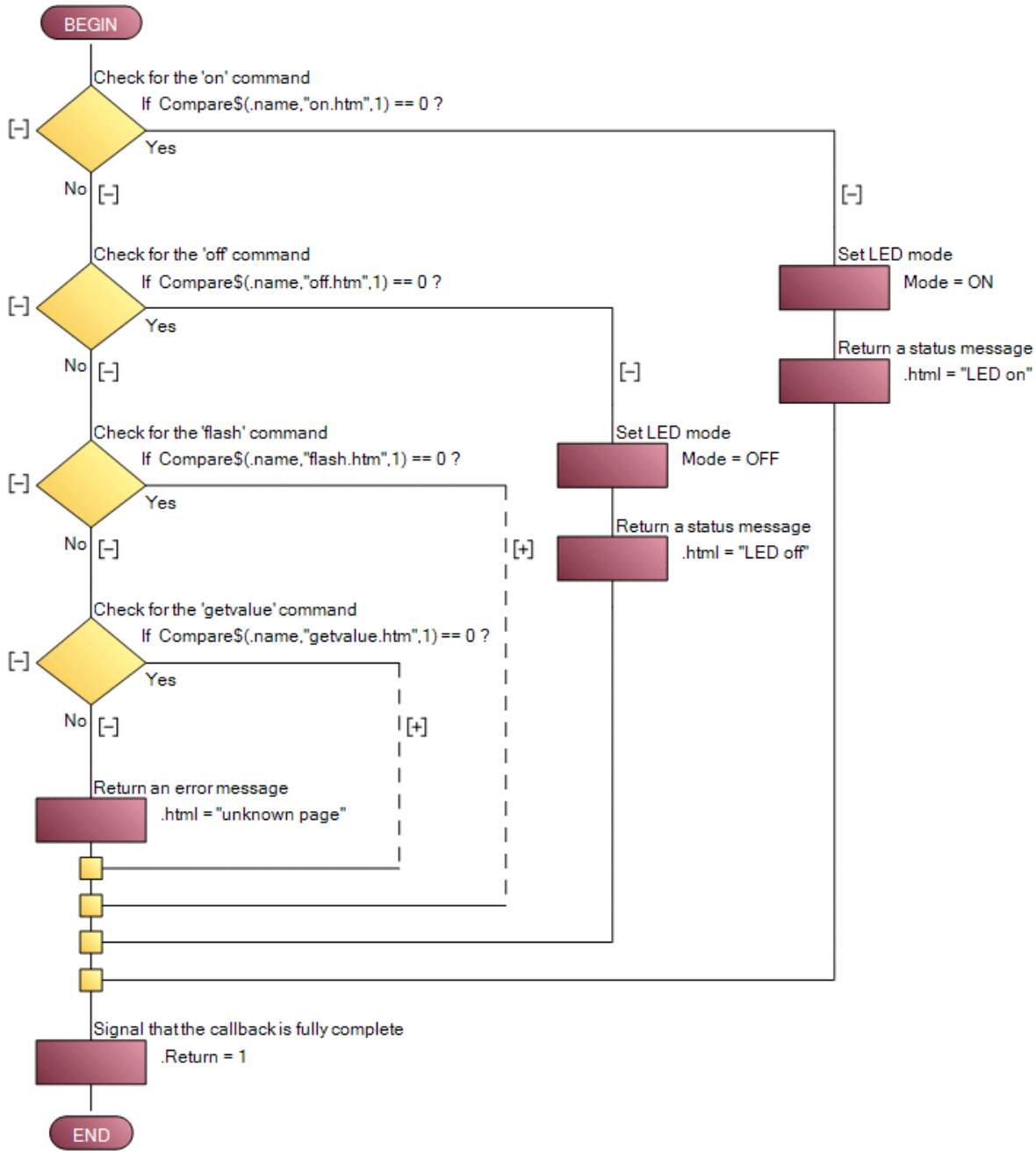


These global variables are initialised at the beginning of the Main macro and used within the infinite loop to adjust the LED output as follows:



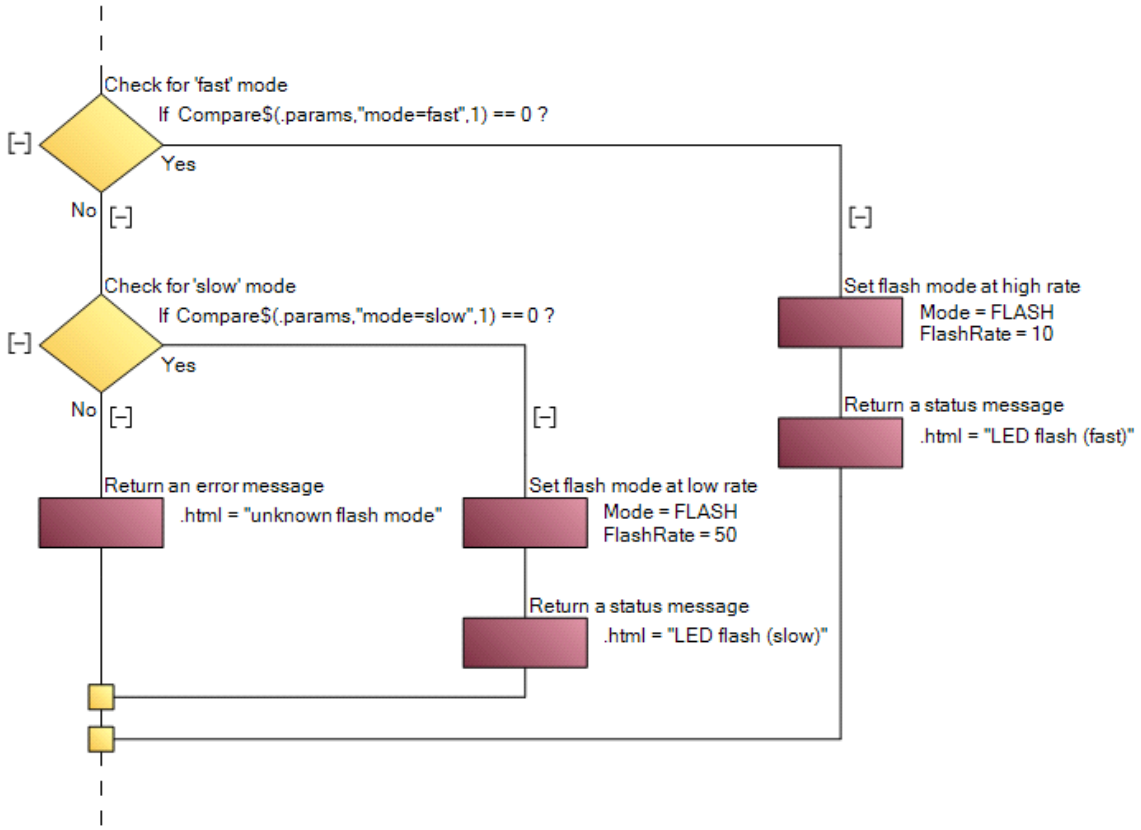
During this infinite loop, the WebServer component call checks for external page requests and this calls the HtmlCallback, which sets the values of these global variables according to the received page requests. These page requests are essentially the commands that will be sent by the web app.

As before, the page name is checked and if it is an expected command then the appropriate global variables are updated and the returned text response is set accordingly.



The dotted lines are hidden code which is explained below.

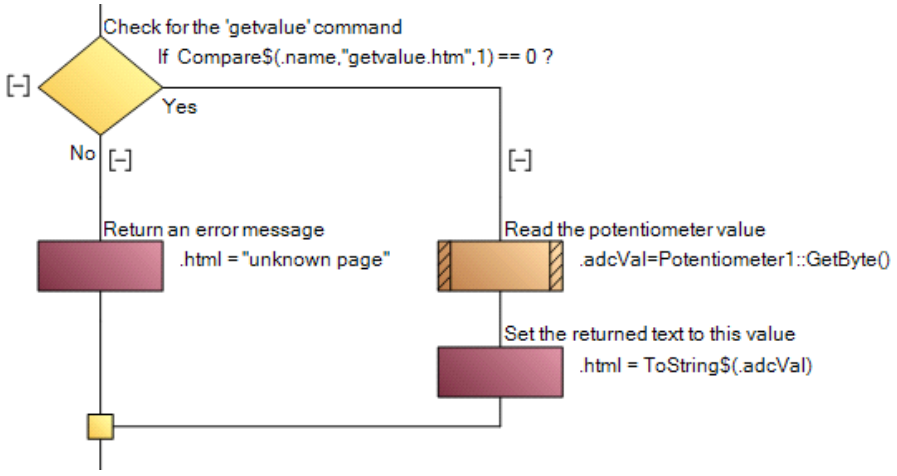
If the “flash” command is received, the URL parameter is checked and this determines the flash rate of the output.



URL parameters are commonly used to provide additional information when requesting a web page. This example uses web pages as commands and so adding the flash rate as a parameter to the “flash.htm” page makes sense. Here is how this app expects to receive a “flash slow” command:

- http://192.168.0.87/flash.htm?mode=slow

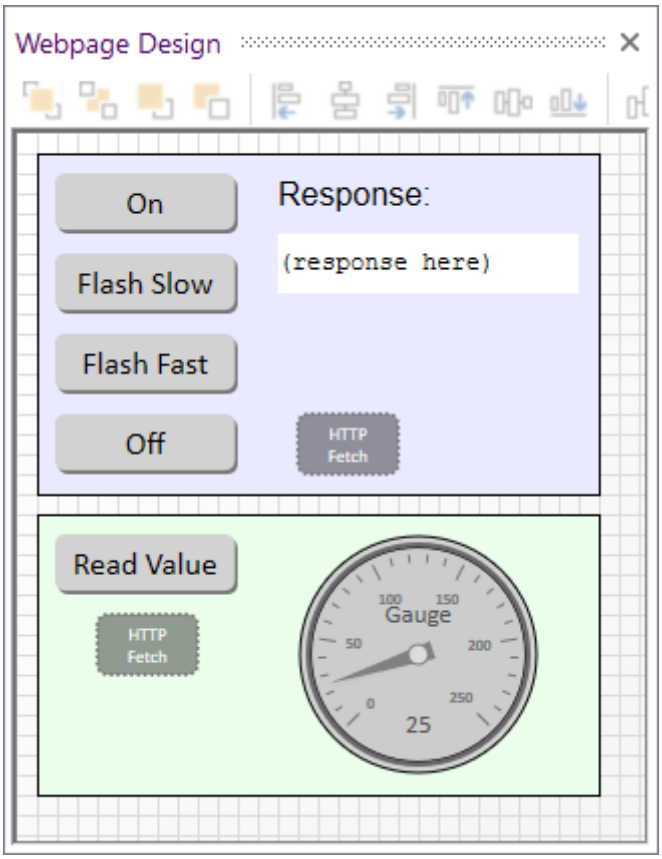
The “getvalue” command is handled by reading the potentiometer value and returning is as text in the HTML response:



This completes the intermediate embedded project. Follow the previous instructions for downloading and testing this embedded app.

Intermediate Web Developer Example (WebProject-02)

The intermediate embedded project adds commands to flash the output and also to read back a sensor value from the ESP32, so the first task is to extend the user interface panel of the Web Developer project to send these new commands. The two “Flash” commands will use a the same Fetch component as the “On” and “Off” commands, but a different Fetch component will be used for the “Read Value” command because the returned data needs to be interpreted differently. A Gauge component is also added to the panel to display this returned value.

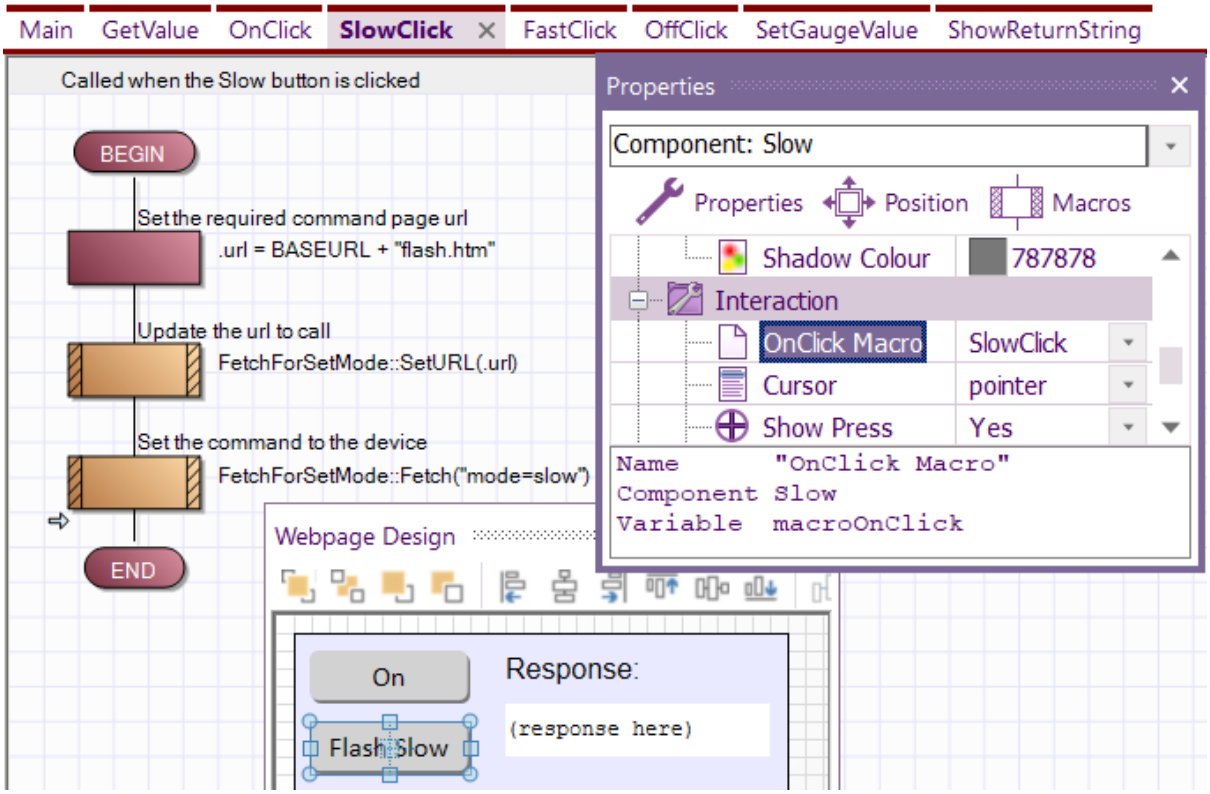


Properties for the Fetch component for the “On”, “Off” and “Flash” commands remains the same as the basic example, but the “Read Value” Fetch component is different. The URL is already known and so can be entered as a property, and a different macro is used to process the data:

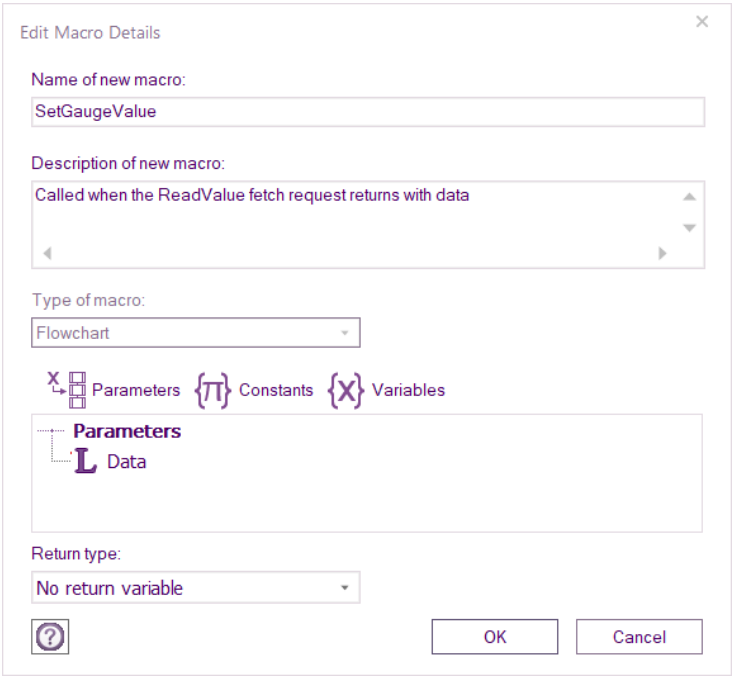
Component	
Handle	FetchForSetMode
Type	HTTP Fetch
Properties	
URL	
Call macro	ShowReturnString
DataType	Raw Text

Component	
Handle	FetchForReadVal
Type	HTTP Fetch
Properties	
URL	http://192.168.0.87/getvalue.htm
Call macro	SetGaugeValue
DataType	Raw Text

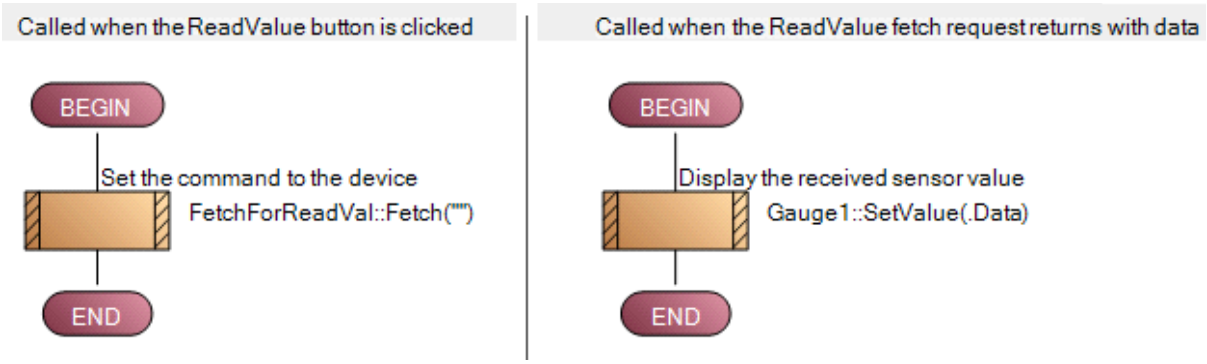
The two “Flash” buttons use similar properties and code as the “On” and “Off” buttons. The only significant change is the use of a URL parameter when calling the Fetch macro. The “Flash Slow” button is shown below, showing the “mode=slow” parameter when calling the Fetch macro:



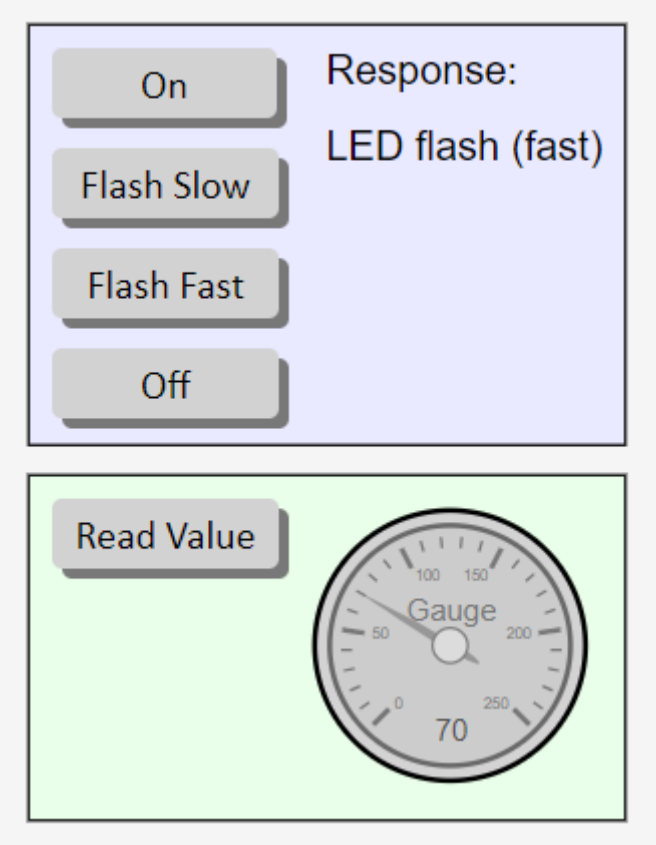
A new macro needs to be created for dealing with the Fetch return when reading values back from the ESP32 device. Again, this is similar to the one used by the other Fetch component, but this time the parameter used for the Data is set to be a number:



The macros for the “Read Value” button click and the associated Fetch return are simple and shown below.



After creating the web page and loading it into a web browser, the running web app will look like this:



Advanced Embedded Example (EmbeddedProject-03)

This will extend the intermediate example by receiving all commands via URL parameters and responding with JSON-formatted data.

The first step is to specify the JSON data format in the “Custom Header” entry of the Web Server component. Change this entry so is has the following text:

```
HTTP/1.0 200 OK
Access-Control-Allow-Origin: *
Content-type: application/json
```

The other change is to the HtmlCallback macro. This will be significantly different to the Basic and Intermediate Embedded examples because it uses parameters on the requested html page to determine the supported commands, and the response returned is now to be formatted as a JSON string.

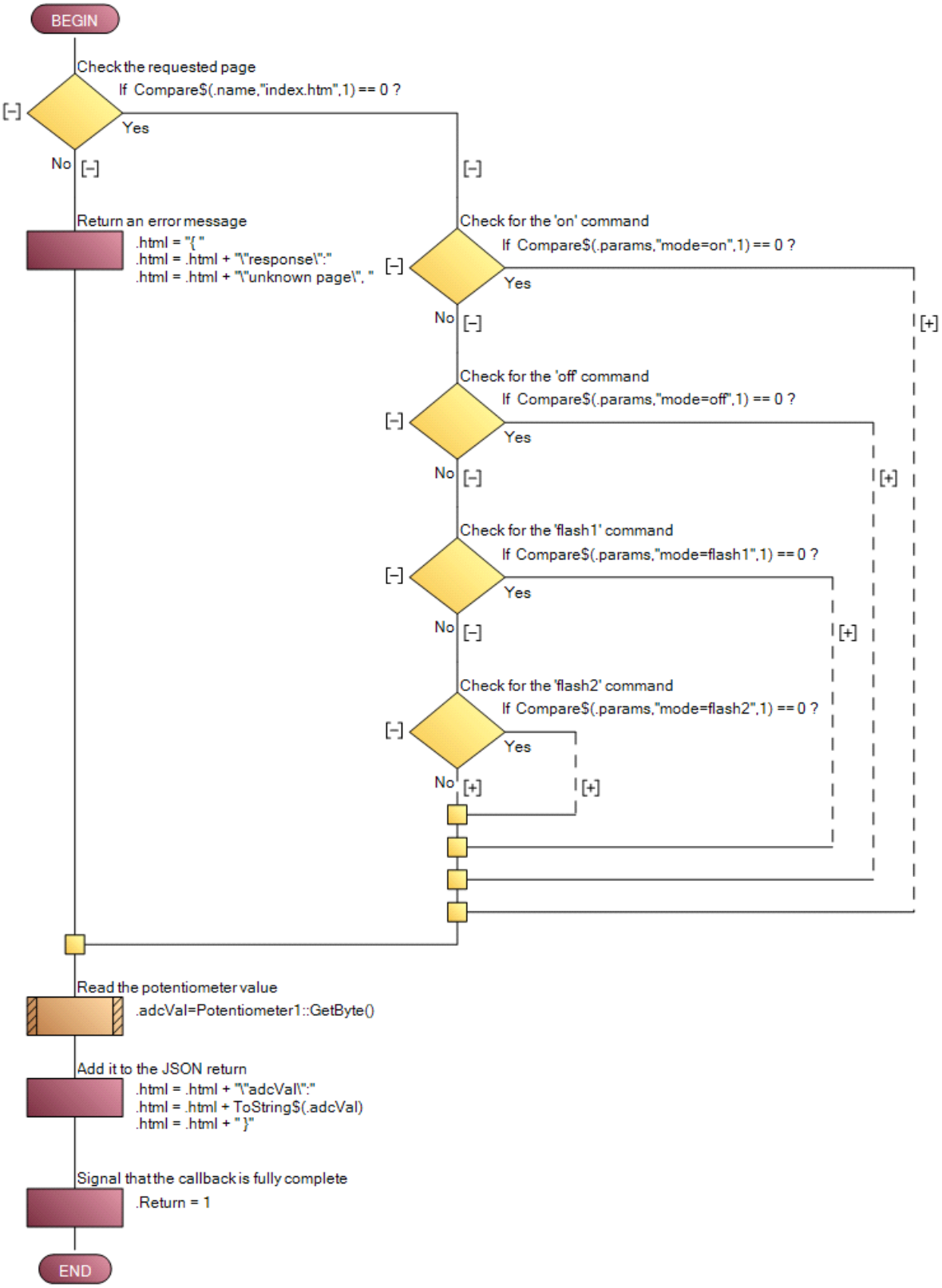
JSON is commonly used when transporting data across the internet. It is human-readable and easily converted into JavaScript objects which can be read by Web apps. Our app will respond with 2 pieces of information – a text response and a numeric value for the sensor. A typical JSON string response for this app will look like this:

```
{ "response":"LED on", "adcVal":158 }
```

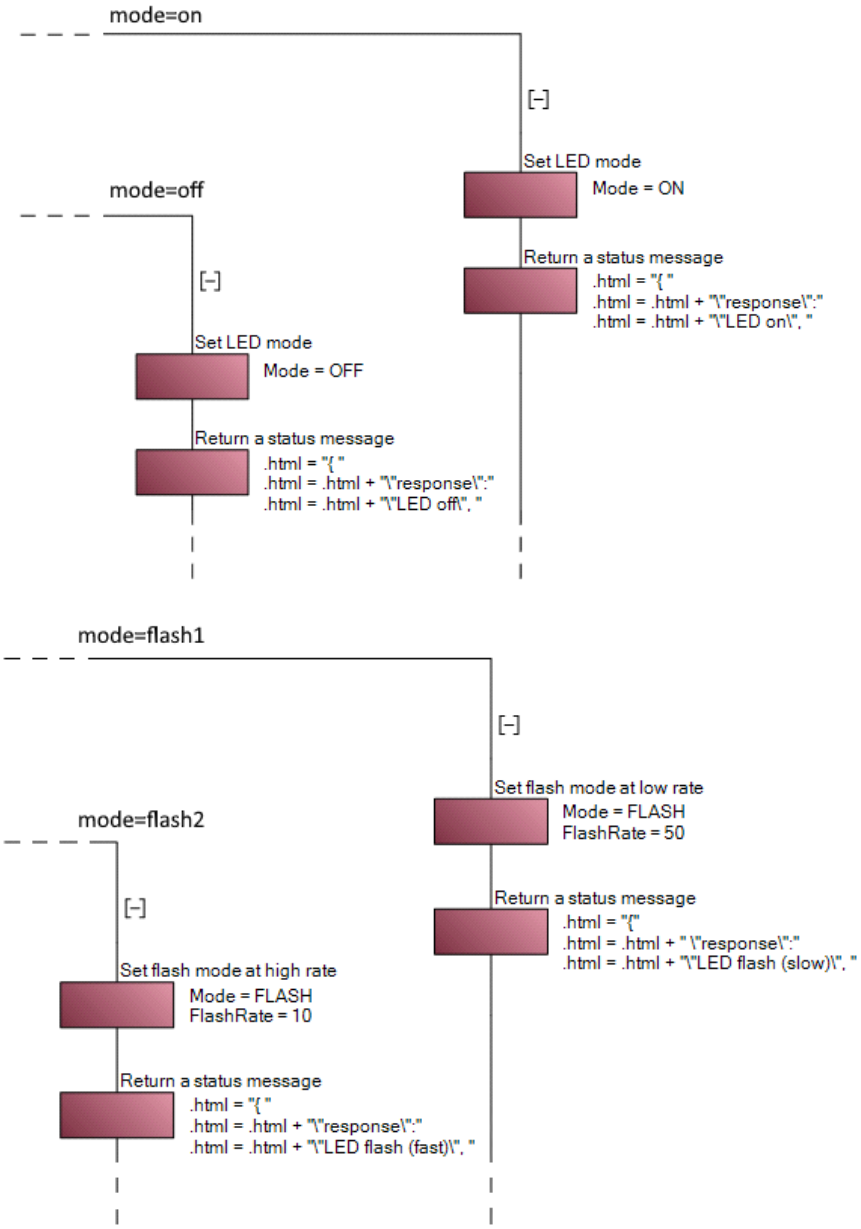
The web server in this app supports a single index.htm page, and commands are received via the parameters added to this page request by the web app. This parameter has 4 possible values:

mode=on	Turns the LED on
mode=off	Turns the LED off
mode=flash1	Sets the LED flash rate to slow
mode=flash2	Sets the LED flash rate to fast

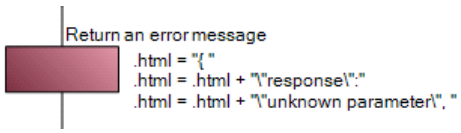
The HtmlCallback macro is shown below. First the page name is checked and if it is the index.htm page, the parameter string is checked for the valid commands in the table above. If the page is not index.htm, the JSON string “response” value is set to “unknown page”. In all cases, the potentiometer is read and the value is returned in the JSON string as “adcVal”.


BEGIN
Check the requested page
If Compare$(.name,"index.htm",1) == 0 ?
Yes
No
Return an error message
.html = "{ "
.html = .html + "\"response\":"
.html = .html + "\"unknown page\", "
Check for the 'on' command
If Compare$(.params,"mode=on",1) == 0 ?
Yes
No
Check for the 'off' command
If Compare$(.params,"mode=off",1) == 0 ?
Yes
No
Check for the 'flash1' command
If Compare$(.params,"mode=flash1",1) == 0 ?
Yes
No
Check for the 'flash2' command
If Compare$(.params,"mode=flash2",1) == 0 ?
Yes
No
Read the potentiometer value
.adcVal=Potentiometer1::GetByte()
Add it to the JSON return
.html = .html + "\"adcVal\":"
.html = .html + ToString$(.adcVal)
.html = .html + " }"
Signal that the callback is fully complete
.Return = 1
END

The dashed lines in the decision branches in the above image contain hidden code to set the global variables and set the “response” value of the JSON string. Here is the code hidden in these branches:



An unrecognised parameter leads to an error response similar to the unrecognised page:



When this is compiled and downloaded to the target ESP32, a browser can be used to test the app using parameters to the index.htm page similar to these:

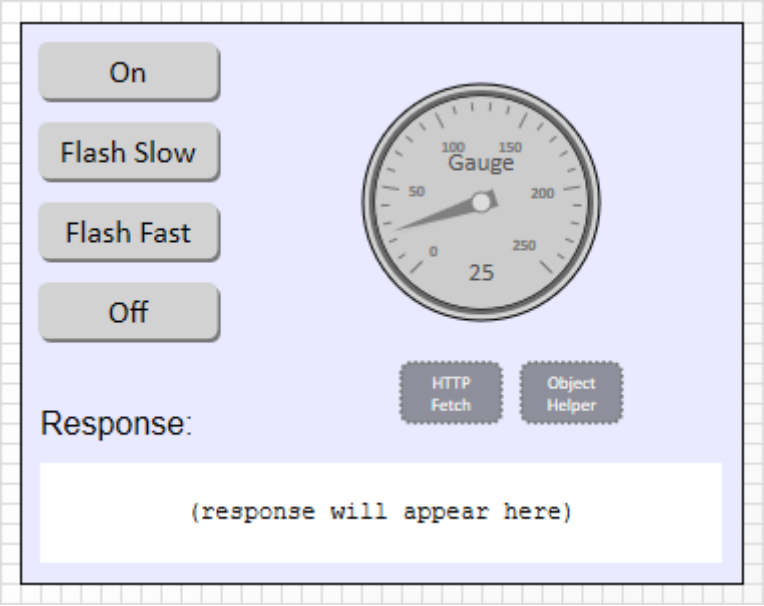
- http://192.168.0.87/index.htm?mode=on
- http://192.168.0.87/index.htm?mode=off
- http://192.168.0.87/index.htm?mode=flash1
- http://192.168.0.87/index.htm?mode=flash2

Advanced Web Developer Example (WebProject-03)

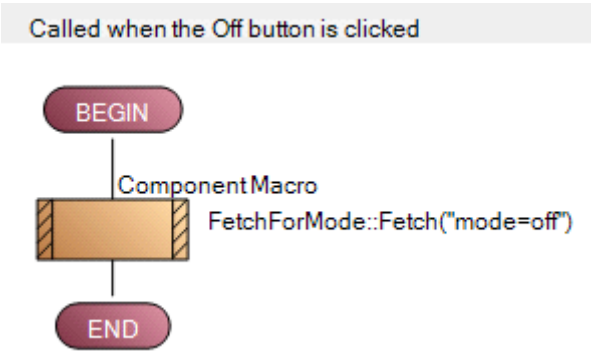
In some ways, this advanced example is simpler than the intermediate example because all commands sent to the embedded ESP32 app are consolidated to a single page request (the index.htm page) and there is a common return format sent back from the ESP32.

The trade-off for this simplicity is the web app now receives a JSON-encoded string in response to each page request, and this requires additional processing to extract the data received.

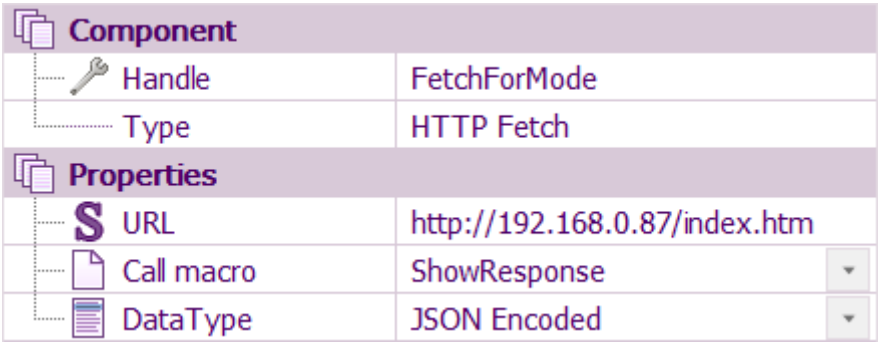
The first task is to remove the “Read Value” button and associated Fetch component because the sensor data is always returned and processed in the same way for all page requests. An Object Helper component is also required to help process the received JSON data, which is automatically converted to a JavaScript object. The helper component has no properties, but has useful macros to manipulate the JavaScript object received.



Each command sent to the hardware is a simple request to the “index.htm” page using parameters to specify the actual command. Therefore, the called macro code for each button is simply a call to the same Fetch component using its own parameter string. Here is the code for the macro called by the “Off” button:



Because the same page is requested by each command button, the actual URL for the page can be entered as a property for the Fetch component. The other change to the Fetch component is to specify the received data is in JSON-encoded format:



JSON is very common in web communication and is used to transfer data between apps. It is a string with a specific format that represents a JavaScript "object". These are essentially variables that have multiple parts, with each part having a "name" and a "value".

The "name" of each part is a text string, and the "value" of each part can be a string, number or even another object. This allows the object to be a very flexible container for data, and because objects can contain other objects, a lot of diverse information can be contained in them.

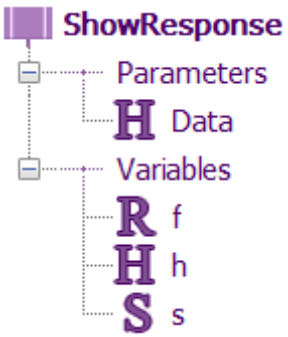
The JSON string expected from the embedded device in this example is very simple and consists of 2 pieces of information – a string "response" and a numerical "adcVal" sensor value. A typical reply from the embedded device looks like this:

```
{ "response":"LED on", "adcVal":158 }
```

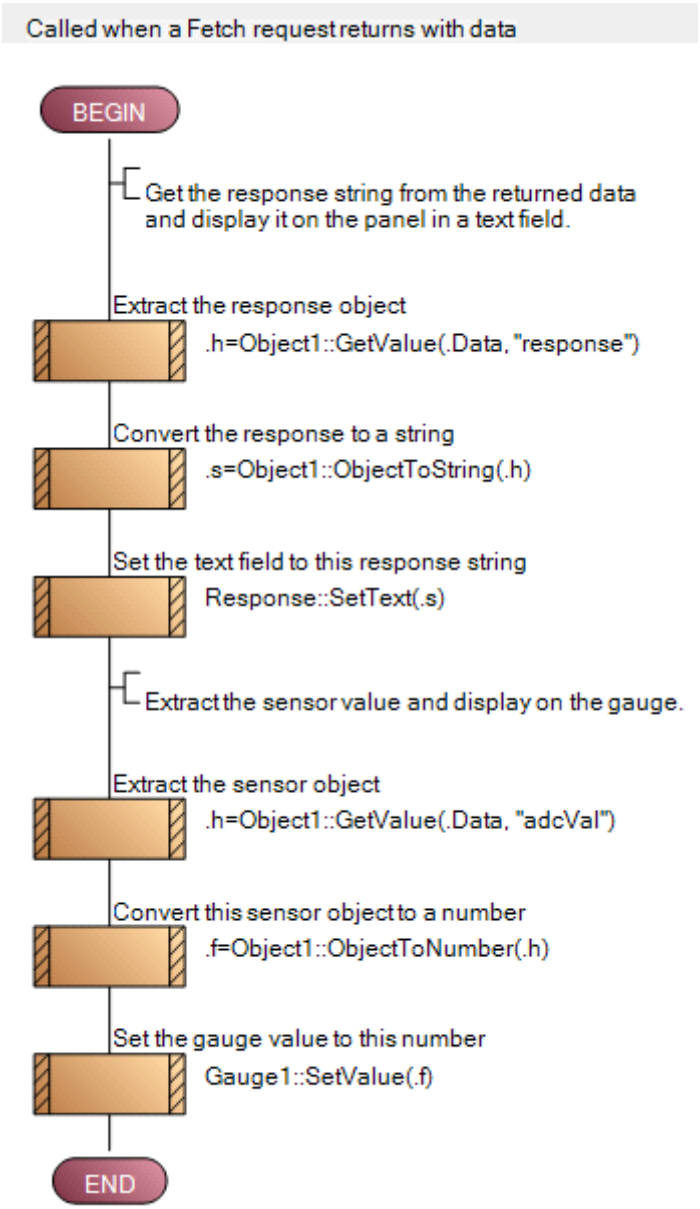
The DataType for the Fetch component has been specified as a JSON-encoded string and so the "Data" parameter to the "ShowResponse" macro will be a JavaScript object and this used Flowcode's "Object/Handle" variable type.

Several local variables are also required by this macro which can be specified when the macro is created or added later. These are:

- f = Floating point variable for the "adcVal" sensor value
- s = String variable for the "response" value
- h = Object/Handle variable used to extract data from the "Data" object



The code within the ShowResponse macro is shown below and consists of two similar sections, each using the Object Helper component macros to extract a part of the received Data object and display it on the panel:



The “GetValue” macro is used to extract specific values from within the JavaScript object, but because these values could be of any data type this macro returns another object. This is the reason for using the “.h” local variable.

Once extracted, this object is converted to its appropriate type – “response” is a string and “adcVal” is a number – and then displayed on the panel either in a text field or a gauge.

Conclusion

This set of worked examples provides a detailed look at simple communications between an embedded ESP32 device and a web browser using Flowcode to construct both the embedded app and the web app.

The examples use the HTTP comms protocol where commands are sent to the embedded device by requesting named webpages (with optional parameters) and data is returned using simple raw text and JSON-encoded strings.

It highlights the differences in techniques to construct web apps and embedded apps. Embedded projects are generally linear and procedural in nature and often consisting of a single infinite loop. Whereas web apps are event-driven and code in macros is executed when external events occur – such as the user clicking a button in the web app or data being returned after a “fetch” request.

Embedded components are used to simplify the connection to a local Wi-Fi network and expose the embedded device as a webserver to allow external access from a web browser on a PC or mobile phone.

Two important web-based components are introduced to simplify communication from the browser to the embedded device – “Fetch” and “Object Helper”. The Fetch component initiates web communication by requesting a web page on the embedded device and specifying a macro to process any data received from this request. The Object Helper aids the extraction of data from received JSON object using basic JavaScript object manipulation.

The techniques and information covered in this worked example form a solid foundation for using Flowcode to create interactive embedded and web apps and understand how to communicate between them using a simple web-based protocol.