Lab Note - Adding Resources to your project and adding features to resources

Introduction

Resources are the way that physical sensors and actuators are exposed from your sensor platform (microcontroller board) to the rest of the system.

The endpoint device runs the mbed device library and code which creates an endpoint that connects to mbed Device Server. This endpoint is itself a tiny web server that handles GET and PUT operations which originate from the mbed Device Server proxy, on behalf of your web application. That is, when your web application does a HTTP GET to a location on the Device Server to access data on your sensor, the Device Server either returns the data from its cache, or it does a CoAP GET to the corresponding URL on your sensor endpoint to retrieve the data, update the cache, and return the updated data to the HTTP client web application.

This lab note will explain how to use the mbed library to create new Resources in your project to expose the sensors and actuators you will be using.

Resource Design

The first step is to design the resource and it's data model. If there is an existing IPSO Smart Object that is applicable, the resource design is mostly done. You will need to specify a URI consisting of Object ID, Object Instance, and Resource ID, a set of allowable operations, and a resource type string for each resource.

The **URI part** is constructed from Object and Resource ID numbers, in the case of IPSO Smart Objects these are officially registered with OMNA. The registry is publicly viewable at: http://technical.openmobilealliance.org/Technical/technical-information/omna/lightweight-m2m-lwm2m-object-registry

ID numbers 10241 – 32768 are available for experimental use. Thus a URI of for example 11000/0/22002 may be constructed and used in your project. The IPSO Smart Object IDs and resource IDs are registered in this registry.

Allowable operations are a subset of the REST verbs GET, PUT, POST and DELETE.

The **resource type** string can be a string without whitespace or special characters (See RFC 6690) that is used to construct the Resource Type relation (rt=). This is useful for embedding semantic information about resources that can be used in discovery.

The resource should be a meaningful abstraction of the data provided by the sensor or consumed by the actuator. For example, a resource reading the current state (ON,OFF) of a light beam interrupter type presence sensor may be useful in some cases, but a resource that counts the number of interruptions or the frequency of interruptions may be even more useful for

some applications. There are examples of objects in the IPSO Smart Object Guideline that have both current state information and aggregate information such as min, max, and count.

Once you have selected or decided the URI (Object/Instance/Resource), the allowable operations, the data type and resource type, you may create the resource file and edit main.cpp to configure the resources.

Files needed to implement resources

Each resource constructor is implemented in a header file in a directory called mbedEndpointResources. There is a file in the resources directory for each resource, conventionally named something descriptive of the resource, in the example project there is a resource called “TempResource”. It’s up to you as the developer to create or edit these files for the resources you wish to implement. You may use the example files as templates to create your resource files. The necessary changes are **highlighted in bold** in the examples below.

Sensor and Actuator Drivers and Libraries

In the TempResource example, the code to read the physical I/O pin is included in the resource file instead of a separate library. For more complex sensors and actuators, there may be a separate library file, with a small wrapper code in the resource file to interface to the library. Any necessary library initialization can be performed in the resource constructor, which is called from main.cpp when the resource is built during initialization. In the LightResource example, there is a separate library Chainable_RGB_LED included.

Resource.h file

Each resource is defined by a resource.h file that contains the resource constructor. This file contains the definitions specific to the resource, such as the URI, Allowable operations mask, and resource type string, and also contains definitions for resource request callback functions. These callback functions are invoked when CoAP REST API calls are made to the device by Device Server on behalf of application software.

The resource request callback functions are where function calls to device libraries would be made in order to interact with the physical sensors and actuators connected to the device via GPIO pins.

Refer to the file TempResource.h in the example project

First are the references and constructors for external objects and libraries:

```
// Temperature Sensor connected to A0
AnalogIn temp_in(A0);
```

In this case, it’s a simple AnalogIn() function from the built-in mbed library.

Next the constructor for the base DynamicResource class:

```
public:
    /**
    Default constructor
    @param logger input logger instance for this resource
    @param name input the resource name
    @param observable input the resource is Observable (default: FALSE)
    */
    TempResource(const Logger *logger,const char *name,const bool observable = false)
:
        DynamicResource(logger,name,"Temperature", SN_GRS_GET_ALLOWED,observable) {
    }
```

Here we only need to set the Resource Type (“Temperature”) and allowable operations (GET only in this case). Any initialization code can be put into the constructor initializer code between the curly braces.

Finally, we need to implement the callbacks for each allowable operation, in this case we only implement get:

```
    Get the value of the Temperature Sensor
    @returns string containing the temperature value in Fahrenheit
    */
    virtual string get() {
        char temp_str[8];
        int B = 3975;
        memset(temp_str,0,8);
        float temp_read = temp_in.read();
        float resistance = (1-temp_read) * 10000/temp_read;
        float temp_c = 1/( logf(resistance/10000)/B + 1/298.15 ) - 273.15;
        float temp_f = (1.8 * (double) temp_c) + 32;
        sprintf(temp_str,"%8.2f", temp_f);
        return string(temp_str);
    }
```

In this example, the binary data from the GPIO wrapper is scaled to Fahrenheit, converted to a string using sprintf() and returned to be sent back to the client in the payload of the response packet.

For resources which implement PUT, POST, and DELETE, there would need to be corresponding implementations of callbacks for these operations.

Resource configuration in main.cpp

The resource constructors in the resource.h files are invoked from main.cpp in the demo example.

Static resources are read-only and never change. These are defined for information about the endpoint or other resources. The constructor for static resources takes the resource name or path (“3/0/0”) and the resource value (“Freescale”).

```
// Static Resources
#include "StaticResource.h"
StaticResource mfg(&logger,"3/0/0","Freescale");
StaticResource model(&logger,"3/0/1","K64F mbed Ethernet demo");
```

Dynamic resources are constructed from the resource declarations in the resource.h files. Note that more than 5 dynamic resources requires changing a table size definition.

```
//
// Dynamic Resource Note:
//
//  mbedConnectorInterface supports up to IPT_MAX_ENTRIES
//  (currently 5) independent dynamic resources.
//
//  You can increase this (at the cost of memory)
//  in mbedConnectorinterface.h
//

// Light Resource
#include "LightResource.h"
LightResource light(&logger,"3311/0/5706");

// LED Resource
#include "OnBoardLED.h"
LEDResource led(&logger,"3311/1/5706");

// Temperature Resource
#include "TempResource.h"
TempResource temp(&logger,"3303/0/5700", true); /* true if observable */
```

Here is where the constructors are invoked and the URI for the resource is passed to the constructor. This enables more than one instance of a particular resource to be constructed, e.g. “3303/0/5700” and “3311/0/5706” using the same resource subclass.

Note that observability is enabled here, for each resource instance as desired. In this example, only the slider resource is observable.

After the constructors are called, the endpoint is started up, and configure_endpoint() is called. Here we need to call .addResource for each resource we want to register with Device Server.

```
// called from the Endpoint::start() below to create resources and the endpoint
internals...
Connector::Options *configure_endpoint(Connector::OptionsBuilder &config)
{
    // Build the endpoint configuration parameters
    logger.log("configure_endpoint: building endpoint configuration...");

    temp.setMaxAge(0);
```

```
/* MaxAge = 0 to disable caching of the temperature value in the Device Server */

    return config.setEndpointNodename(MY_ENDPOINT_NAME)
// custom endpoint name
                .setNSPAddress(my_nsp_address)
// custom NSP address
                .setDomain(MY_NSP_DOMAIN)
// custom NSP domain
                .setNSPPortNumber(my_nsp_coap_port)
// custom NSP CoAP port

                // add the static resource representing this endpoint
                .addResource(&mfg)
                .addResource(&model)

                // Add my specific physical dynamic resources...
                .addResource(&light)
                .addResource(&temp, 1000)
                .addResource(&led)

                // finalize the configuration...
                .build();
}
```

Setting Cache Control and the Observe option

There are two options being set here in main.cpp. First, **`temp.setMaxAge(0);`** sets the cache control for the temperature resource instance to disable caching by setting max-age to zero. If not set to zero, this sets the maximum time data will be valid in the Device Server cache.

Secondly, the observation polling interval is specified for the temperature resource as 1000 milliseconds using **`.addResource(&temp, 1000)`**. This will cause an update of the slider resource to be sent to the Device Server and application every 1 second whenever observation is activated for that resource.