Lab Note #2 - Adding Resources to your project and adding features to resources

Contents:

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 sensor platform is running the mbed device library and code which creates an endpoint that connects to the mbed Device Server. This endpoint is itself a tiny web server that handles GET and PUT operations which originate from the mbed Device Server's 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.

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

Resource Registration

Each sensor or actuator is exposed as a Resource. This is a URL addressable entity which has a link that can be registered with mbed Device Server to make it discoverable and available to web applications connected to the Device Server.

For example, the first demo LED control project exposes a resource at the CoAP URL /11100/0/5900, which was exposed to HTTP clients at an address on the Device Server at /domain/endpoints/<endpointID>/11100/0/5900, where <endpointID> is the string your sensor platform uses to identify itself to the Device Server (currently the unique MAC address).

The sensor endpoint software running on the microcontroller board maintains a registration on the mbed Device Server. There is an initial registration each time the endpoint is started (power

on or reset), and there is a registration update performed every 30 seconds as a keep-alive function which also maintains the network connection through routers and firewalls.

The registration is done in a template routine that uses mbed device library functions to communicate with the server:

```
//
********************************************************************************
// Resource creation

static int create_resources()
{
    sn_nsdl_resource_info_s *resource_ptr = NULL;
    sn_nsdl_ep_parameters_s *endpoint_ptr = NULL;

    NSDL_DEBUG("Creating resources");

    /* Create resources */
    resource_ptr =
(sn_nsdl_resource_info_s*)nsdl_alloc(sizeof(sn_nsdl_resource_info_s));
    if(!resource_ptr)
        return 0;
    memset(resource_ptr, 0, sizeof(sn_nsdl_resource_info_s));

    resource_ptr->resource_parameters_ptr =
(sn_nsdl_resource_parameters_s*)nsdl_alloc(sizeof(sn_nsdl_resource_parameters
_s));
    if(!resource_ptr->resource_parameters_ptr)
    {
        nsdl_free(resource_ptr);
        return 0;
    }
    memset(resource_ptr->resource_parameters_ptr, 0,
sizeof(sn_nsdl_resource_parameters_s));

    // Static resources
    nsdl_create_static_resource(resource_ptr, sizeof("3/0/0")-1, (uint8_t*)
"3/0/0", 0, 0,  (uint8_t*) "mbedDEMO", sizeof("mbedDEMO")-1);
    nsdl_create_static_resource(resource_ptr, sizeof("3/0/1")-1, (uint8_t*)
"3/0/1", 0, 0,  (uint8_t*) "DEMO", sizeof("DEMO")-1);

    // Dynamic resources
    create_light_resource(resource_ptr);

        /* Register with NSP */
    endpoint_ptr = nsdl_init_register_endpoint(endpoint_ptr,
(uint8_t*)endpoint_name, ep_type, lifetime_ptr);
    if(sn_nsdl_register_endpoint(endpoint_ptr) != 0)
```

```
        pc.printf("NSP registering failed\r\n");
    else
        pc.printf("NSP registering OK\r\n");
    nsdl_clean_register_endpoint(&endpoint_ptr);

    nsdl_free(resource_ptr->resource_parameters_ptr);
    nsdl_free(resource_ptr);
    return 1;
}
// ************************************************
```

The part specific to your project is in bold above. Static resources are for attributes and resources that don't change and are read-only, like the device information above.

Dynamic resources are used for the active sensor and actuator endpoints, or for configuration parameters that may be changed. Dynamic resources are configured by calling a create resource function in a separate resource file.

Some other code related to registration and event processing is in the file nsdl_support.cpp

Resource Files

Each resource is implemented in another code template in a directory called resources. There is a file in the resources directory for each resource, conventionally named something descriptive of the resource, in this case "light". Another convention might be to use the resource URL, e.g. "11100_0_5900". It's up to you, the developer, to create these files from the template code.

The resource file contains code to create the resource, which builds links to the resource to register with the Device Server, and internally registers a CoAP request callback handler associated with the resource.

```
#define LIGHT_RES_ID    "11100/0/5900"
int create_light_resource(sn_nsdl_resource_info_s *resource_ptr)
{
    nsdl_create_dynamic_resource(resource_ptr, sizeof(LIGHT_RES_ID)-1,
(uint8_t*)LIGHT_RES_ID, 0, 0, 0, &light_resource_cb, (SN_GRS_GET_ALLOWED |
SN_GRS_PUT_ALLOWED));
    return 0;
}
```

The items in bold above are set according to each resource. The names are required to be unique within your program in order to distinguish between resources and resource file instances. In the example above, **`LIGHT_RES_ID`** is the local URI path for the resource, and **`&light_resource_cb`** points to the callback routine, also in the resource file, below. The callback routine is executed whenever a CoAP GET or PUT request is received with a URI path

component matching the resource path from the create resource step, for example: **"11100/0/5900"**

```
char leds[] = {"1111"}; //YGBR
/* Only GET and PUT method allowed */
static uint8_t light_resource_cb(sn_coap_hdr_s *received_coap_ptr,
sn_nsdl_addr_s *address, sn_proto_info_s * proto)
{
    sn_coap_hdr_s *coap_res_ptr = 0;

    //pc.printf("LED Strip callback\r\n");

    if(received_coap_ptr->msg_code == COAP_MSG_CODE_REQUEST_GET)
    {
        coap_res_ptr = sn_coap_build_response(received_coap_ptr,
COAP_MSG_CODE_RESPONSE_CONTENT);

        coap_res_ptr->payload_len = strlen(leds);
        coap_res_ptr->payload_ptr = (uint8_t*)leds;
        sn_nsdl_send_coap_message(address, coap_res_ptr);
    }
    else if(received_coap_ptr->msg_code == COAP_MSG_CODE_REQUEST_PUT)
    {
        //pc.printf("PUT: %d bytes\r\n", received_coap_ptr->payload_len);
        if(received_coap_ptr->payload_len == 4)
        {
            memcpy(leds, (char *)received_coap_ptr->payload_ptr,
received_coap_ptr->payload_len);

            leds[received_coap_ptr->payload_len] = '\0';
            pc.printf("PUT: %s\r\n",leds);

            //call LED strup update function here
            set_leds(leds);

            coap_res_ptr = sn_coap_build_response(received_coap_ptr,
COAP_MSG_CODE_RESPONSE_CHANGED);
            sn_nsdl_send_coap_message(address, coap_res_ptr);
        }
    }

    sn_coap_parser_release_allocated_coap_msg_mem(coap_res_ptr);
    return 0;
}
```

In the above example, the developer will create a data structure or function to provide the representation of the resource for GET and PUT requests. Here the variable leds is used in the GET handler, and the function set_leds(leds) is used in the PUT handler.

The GET handler needs to provide a representation in JSON compatible byte serial format. A character array without a string terminating value is passed to and from the mbed device library to be used as the CoAP response payload. The length of the payload (4 bytes in this example) is also passed to the device library to assemble the CoAP response packet.

The PUT handler checks the payload length to see if it is 4 bytes, and ignores the request if not. An error response could be returned instead.

If the payload length is 4 bytes, then the string terminating null '\0' is added and passed to the set_leds function as a string. a diagnostic printf is left in to help with observing the sequence.

Sensor and Actuator Drivers and Libraries

The resource request callback function is where function calls to device libraries would be made in order to interact with the sensors and actuators. In the LED example, the code to actuate the LEDs is included in the resource file instead of a separate library:

```
DigitalOut grn(LED1);
DigitalOut red(LED2);
DigitalOut blu(LED3);
DigitalOut yel(LED4);

void set_leds(char *leds)
{
int leds_int ;

    sscanf(leds, "%x", &leds_int);

    grn = ~leds_int & 1;
    red = ~leds_int >> 4 & 1;
    blu = ~leds_int >> 8 & 1;
    yel = ~leds_int >> 12 & 1;
}
```

This routine is pretty simple, take the string controlling the LED state, convert it to an integer using hex coding and examine the least significant bit of each 4 bit nibble (0 or 1) to see if the corresponding LED should be set on or off. The DigitalOut value is 0 to turn the LED on, so the bit state is inverted when writing the port:

```
grn = ~leds_int & 1;
```

For more complex sensors and actuators, there would 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 create resource function, which is called from the template code in main.cpp

Resource Configuration - Cache Control and Resource Observation

Some useful extensions to the basic resource handler are cache control and asynchronous notification. This section explains how to use more of the CoAP protocol features to customize the interaction between your device and applications.

Cache Control using max-age

Cache control is done in the callback handler, by returning the “max-age” option and a value for max-age to be applied to the response data. The proxy is expected to cache the data no longer than max-age. Setting max-age to zero will disable caching of the response data. Max-age is applied to each response, thus it’s inclusion in the callback handler. Here is an example of max-age being set to 0 to disable caching:

```
// ****************************************
uint8_t max_age = 0;
uint8_t content_type = 50;

// from resource request callback handler
    if(received_coap_ptr->msg_code == COAP_MSG_CODE_REQUEST_GET)
    {
        coap_res_ptr = sn_coap_build_response(received_coap_ptr,
COAP_MSG_CODE_RESPONSE_CONTENT);

        coap_res_ptr->payload_len = strlen(sliderPct);
        coap_res_ptr->payload_ptr = (uint8_t*)sliderPct;

        coap_res_ptr->content_type_ptr = &content_type;
        coap_res_ptr->content_type_len = sizeof(content_type);

        coap_res_ptr->options_list_ptr =
(sn_coap_options_list_s*)nsdl_alloc(sizeof(sn_coap_options_list_s));
        if(!coap_res_ptr->options_list_ptr)
            {
            pc.printf("cant alloc option list for max-age\r\n");
            coap_res_ptr->options_list_ptr = NULL; //FIXME report error and
recover
            }
        memset(coap_res_ptr->options_list_ptr, 0,
sizeof(sn_coap_options_list_s));
        coap_res_ptr->options_list_ptr->max_age_ptr = &max_age;
        coap_res_ptr->options_list_ptr->max_age_len = sizeof(max_age);

        sn_nsdl_send_coap_message(address, coap_res_ptr);
        nsdl_free(coap_res_ptr->options_list_ptr);
        coap_res_ptr->options_list_ptr = NULL;
```

```
        coap_res_ptr->content_type_ptr = NULL;// parser_release below tries
to free this memory

    }

    sn_coap_parser_release_allocated_coap_msg_mem(coap_res_ptr);

    return 0;
}
```

The code in bold above allocates memory for the option list, creates the max-age option and it's corresponding value, and releases the option list memory when done.

Note also in this example that the content-type is being set explicitly to 50 , the code for application/json. Please note this URL for the IANA registered codes associated with CoAP:

http://www.iana.org/assignments/core-parameters/core-parameters.xhtml#content-formats

Asynchronous Notification - Observe

Asynchronous Notification is done using the CoAP Observe option. This is sent in a GET request, and indicates that the requester is willing to receive multiple responses in a series, each representing an update of the sensor value. These responses are sent based on a periodic interval, or changes in sensor value, or both.

Here is an example of a resource that uses a library and supports CoAP Observe, using a periodic sampling interval to report sensor data:

```
// ************************************
// Temperature resource implementation

#include "mbed.h"
#include "rtos.h"
#include "LM75B.h"
#include "nsdl_support.h"
#include "temperature.h"

#define TEMP_RES_ID     "sen/temp"

static LM75B tmp(p28,p27);
/* stored data for observable resource */
static uint8_t obs_number = 0;
static uint8_t *obs_token_ptr = NULL;
static uint8_t obs_token_len = 0;
static char temp_val[5];
extern Serial pc;
```

```
/* Thread for calling libNsdl exec function (cleanup, resendings etc..) */
/* Node updates temperature every 10 seconds. Notification sending is done
here. */
static void exec_call_thread(void const *args)
{
    int32_t time = 0;
    while (true)
    {
        wait(1);
        time++;
        sn_nsdl_exec(time);
        if((!(time % 10)) && obs_number != 0 && obs_token_ptr != NULL)
        {
            obs_number++;
            sprintf(temp_val,"%2.2f" ,tmp.read());
            if(sn_nsdl_send_observation_notification(obs_token_ptr,
obs_token_len, (uint8_t*)temp_val, 5, &obs_number, 1,
COAP_MSG_TYPE_NON_CONFIRMABLE, 0) == 0)
                pc.printf("Observation sending failed\r\n");
            else
                pc.printf("Observation\r\n");
        }
    }
}

/* Only GET method allowed */
/* Observable resource */
static uint8_t temp_resource_cb(sn_coap_hdr_s *received_coap_ptr,
sn_nsdl_addr_s *address, sn_proto_info_s * proto)
{
    sprintf(temp_val,"%2.2f" ,tmp.read());
    sn_coap_hdr_s *coap_res_ptr = 0;

    pc.printf("temp callback\r\n");
    coap_res_ptr = sn_coap_build_response(received_coap_ptr,
COAP_MSG_CODE_RESPONSE_CONTENT);

    coap_res_ptr->payload_len = 5;
    coap_res_ptr->payload_ptr = (uint8_t*)temp_val;

    if(received_coap_ptr->token_ptr)
    {
        pc.printf("Token included\r\n");
        if(obs_token_ptr)
        {
            free(obs_token_ptr);
            obs_token_ptr = 0;
        }
```

```
        obs_token_ptr = (uint8_t*)malloc(received_coap_ptr->token_len);
        if(obs_token_ptr)
        {
            memcpy(obs_token_ptr, received_coap_ptr->token_ptr,
received_coap_ptr->token_len);
            obs_token_len = received_coap_ptr->token_len;
        }
    }

    if(received_coap_ptr->options_list_ptr->observe)
    {
        coap_res_ptr->options_list_ptr =
(sn_coap_options_list_s*)malloc(sizeof(sn_coap_options_list_s));
        memset(coap_res_ptr->options_list_ptr, 0,
sizeof(sn_coap_options_list_s));
        coap_res_ptr->options_list_ptr->observe_ptr = &obs_number;
        coap_res_ptr->options_list_ptr->observe_len = 1;
        obs_number++;
    }

    sn_nsdl_send_coap_message(address, coap_res_ptr);

    coap_res_ptr->options_list_ptr->observe_ptr = 0;
    sn_coap_parser_release_allocated_coap_msg_mem(coap_res_ptr);
    return 0;
}

int create_temperature_resource(sn_nsdl_resource_info_s *resource_ptr)
{
    static Thread exec_thread(exec_call_thread);

    nsdl_create_dynamic_resource(resource_ptr, sizeof(TEMP_RES_ID)-1,
(uint8_t*)TEMP_RES_ID, 0, 0, 1, &temp_resource_cb, SN_GRS_GET_ALLOWED);
    return 0;
}
// ****************************************
```

How library support for Observe is used

Code in bold is added to the callback handler to implement observation.

First, in the create resource function, the flag is set for Observable (`0, 0, 1,`) to indicate to the server that it may request asynchronous notifications from this resource.

Then in the GET callback handler, the token is checked and, if included, saved in the dynamic variable obs_token_ptr. The token is included in future notifications in order to enable the server to associate notifications with requests.

The callback handler then checks for the observe option and, if present, adds to the response with an option list including obs_number, which is initially 0. obs_number is incremented as a flag to the sampling loop to start sending notifications.

The function exec_call_thread is started up when the resource is created and waits for obs_number and the token to be set. At that time, the loop is enabled to read the sensor and send the notification every 10 seconds:

```
sprintf(temp_val,"%2.2f" ,tmp.read());
            if(sn_nsdl_send_observation_notification(obs_token_ptr,
obs_token_len, (uint8_t*)temp_val, 5, &obs_number, 1,
COAP_MSG_TYPE_NON_CONFIRMABLE, 0) == 0)
```

Notifications in this example are sent on a periodic basis. Notifications could also be sent when the sensor value changes, or changes significantly. There could be a default reporting period with changes reported more frequently. Only unusual conditions could be reported.

Exec function

Note especially the call to `sn_nsdl_exec(time);`

This is a library function that, when called, checks for long responses and re-sends messages if necessary. It also removes old messages from a list, including the notifications. It should be called about once a second when using notifications, and should only be called from one place, because calling the function requires setting a global time value which must monotonically increase in order for the resending and list purging to work properly.

It is called here out of convenience, but is a global function that should be called periodically when asynchronous notification is used.