XDK110: Getting Started Guide with MQTT (Paho Demo)

Cross-Domain Development Kit XDK110

Platform for Application Development

Bosch Connected Devices and Solutions

BOSCH
Invented for life

XDK110: Getting Started Guide with MQTT (Paho Demo)	
Document revision	1.4
Document release date	**September 30, 2016**
Document number	BCDS-XDK110-MQTT-PAHO-014
Technical reference code(s)	
Notes	Data in this document is subject to change without notice. Product photos and pictures are for illustration purposes only and may differ from the real product's appearance. This document is confidential and under NDA inherent with the purchase of an XDK110. **Advance information – Subject to change without notice**

XDK110

PLATFORM FOR APPLICATION DEVELOPMENT

General Description

The XDK is a wireless sensor device developed to enable rapid prototyping for the internet of things (IoT) space. It allows users to quickly realize their own use case (or “Proof of concept” project) and understand what the requirements of their ideal product are.

The XDK comes with a lithium ion battery, extension board, micro USB 2.0 cable and mounting plate and screws. The XDK is equipped with 7 physical sensors: Accelerometer, Gyroscope, Magnetometer, Inertial (Accelerometer and Gyroscope), Environmental (Pressure, Humidity, and Temperature), Ambient Noise, and Ambient Light. The XDK has a Micro-SD card slot and two antennas: one for wireless LAN and one for Bluetooth 4.0 communications. The XDK has 3 programmable LEDs, one LED for charge indication, and two programmable buttons.

Development with the XDK is easily done with the XDK Workbench; a programming suite based on the Eclipse Platform, which requires a PC with a Windows 7 or higher operating system. The XDK Workbench delivers all of the API’s, source code and demos necessary for new users to quickly and easily start development of their application with the XDK.

This document details how to setup the MQTT_Paho demo code in the XDK Workbench, flash the XDK with the demo firmware and how to connect to IBM’s demo MQTT broker. The MQTT_Paho code is developed to give the users an introduction to MQTT and shows how the user can quickly and easily connect to an MQTT broker and start streaming live data from the XDK.

This document assumes the user has been granted access to the XDK Community and has already downloaded and installed the XDK Workbench 1.7. It also assumes the user knows how to import a demo into the XDK Workbench. For help with how to download and install the XDK workbench or how to import a demo, please navigate to the XDK website (www.xdk.io), click on the Help & Learning tab and view the Guides. The download for the XDK workbench can be found on the XDK website under the Downloads tab.

Table of Contents

1. XDK MQTT Paho Demo Setup

The source code for the MQTT Paho demo is located on the XDK website at http://www.xdk.io . From the home page select the Downloads tab. Next select the Demos menu item. Scroll down until you find the MQTT Paho section. Click on Download and login or register for an account. Select where to download the files and select save to download the code and the latest version of this document. Extract the zip file onto a computer installed with the XDK workbench.

Before importing the project into the XDK Workbench, the Paho Client source files should be placed in the XDK project's source/paho folder. The version of paho used for the XDK is the embedded c paho client. To download this version, visit the website:

http://git.eclipse.org/c/paho/org.eclipse.paho.mqtt.embedded-c.git/

Under the summary tab, look for the Tag Section and select one of the files under Download (as of this release the latest file is org.eclipse.paho.mqtt.embedded-c-1.0.0). Save this file on the computer with the XDK Workbench. In a file explorer window, navigate to the folder where the downloaded paho folder is stored and extract the files. In the unzipped paho folder, navigate to the MQTTPacket/src folder and copy all the files over to the source/paho folder of the XDK project. Then navigate to the MQTTClient-C/src folder in the unzipped paho folder and copy the MQTTClient.c and MQTTClient.h files to the source/paho folder of the XDK project. Do not copy the subfolders into the project, the folder structure should remain the same after importing the files from the paho project to the XDK project.

For more information on paho please explore the paho website at http://www.eclipse.org/paho/ .

Once you have downloaded all the source code and placed the source files in the proper location, import the project into the XDK workbench. Please see the "Workbench Installation" or "XDK General Information" Guides in the Help & Learning section of the XDK website for more information on how to properly import a project. Once this is done, insure you also have the following items in order to confirm a working installation of the MQTT Paho Demo project.

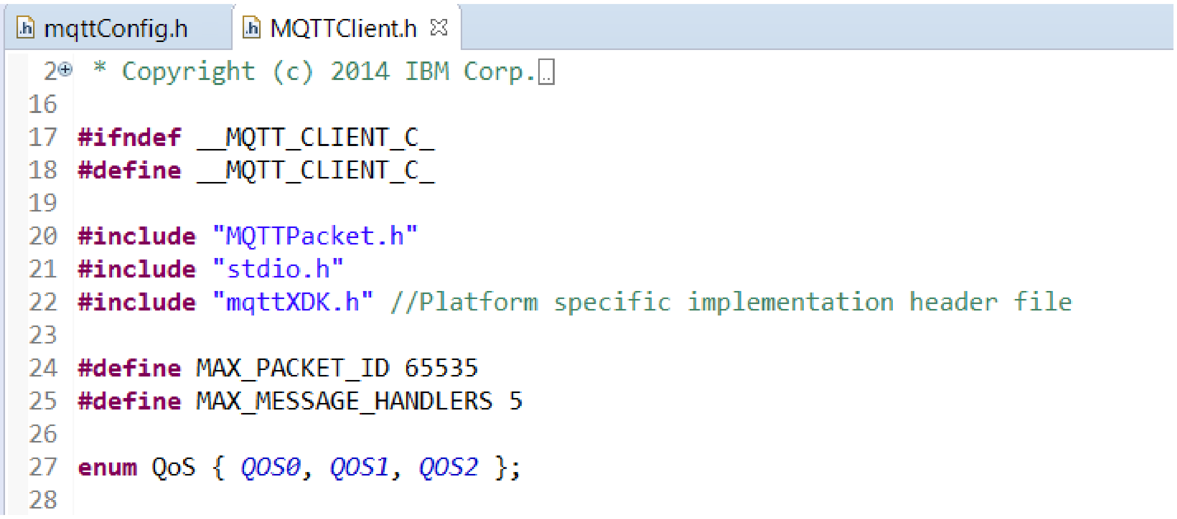
- XDK110 – Hardware Development Kit
- XDK Workbench – Necessary to configure the source code for the wireless network and to flash the XDK. The workbench can be downloaded from the XDK Community.
- Internet Accessible Device – An internet browser is needed to verify the XDK is operating properly and communicating with the MQTT Broker.

This rest of this section will show the user how to configure the code and flash the XDK using the XDK Workbench.

1.1 Paho Source Changes

The XDK code won't compile unless the paho code knows what platform it is working with. To configure the paho code properly, edit the MQTTClient.h (located in the projects paho folder) file to include the platform specific implementation header file. Currently this header is left blank. Edit the #include to be:

```
#include "mqttXDK.h" //Platform specific implementation header file
```



```
 * Copyright (c) 2014 IBM Corp.

#ifndef __MQTT_CLIENT_C_
#define __MQTT_CLIENT_C_

#include "MQTTPacket.h"
#include "stdio.h"
#include "mqttXDK.h" //Platform specific implementation header file

#define MAX_PACKET_ID 65535
#define MAX_MESSAGE_HANDLERS 5

enum QoS { QOS0, QOS1, QOS2 };

```

Figure 1-1. Paho's MQTTClient.h Edited for the MQTT_Paho Demo Code

1.2 Configuration Header (mqttConfig.h)

There are five network settings and six data collection settings. These settings can be configured through the configuration header, mqttConfig.h located under the source folder of the MQTT_Paho demo project.

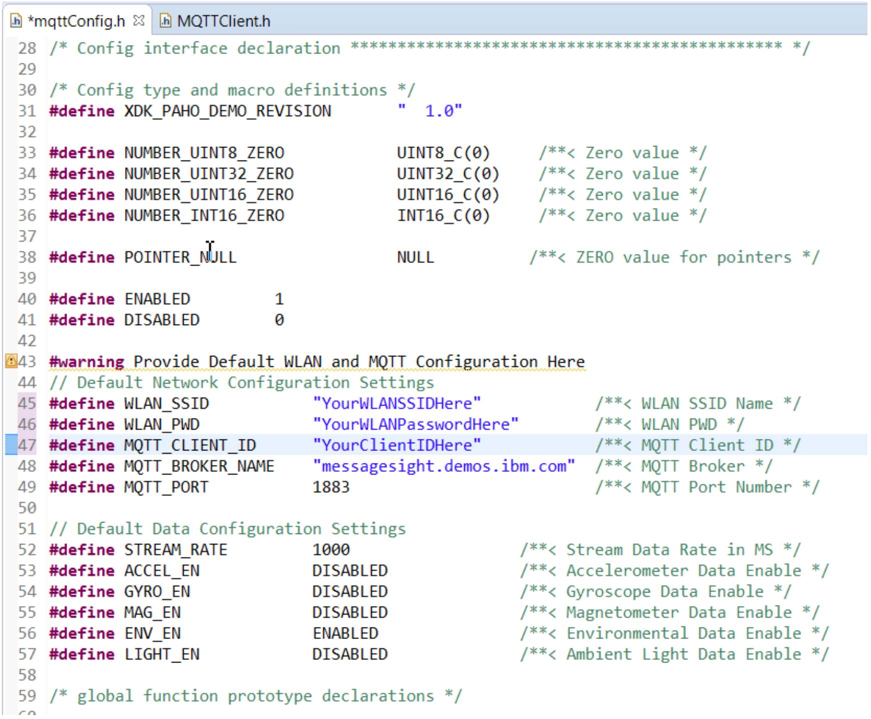
1.2.1 Configuring Network Settings

The five network settings will be located near the top of the configuration header file. Define these to match your wireless network's settings and MQTT Broker. Please keep the format as shown in figure 1-1, with the quotations around the first four settings. The MQTT_CLIENT_ID must be a unique ID for each XDK you implement. Below is the list of network settings:

- WLAN_SSID – Name of the WLAN network the XDK is on
- WLAN_PWD – Password of the WLAN network the XDK is on
- MQTT_CLIENT_ID – Unique ID for the XDK
- MQTT_BROKER_NAME – URL of the MQTT Broker
- MQTT_PORT – Port the MQTT Broker communicates over

To connect to the IBM Demo MQTT Broker set the following values:

- MQTT_BROKER_NAME = "messagesight.demos.ibm.com"
- MQTT_PORT = 1883



```
/* Config interface declaration ********************************************** */

/* Config type and macro definitions */
#define XDK_PAHO_DEMO_REVISION       "  1.0"

#define NUMBER_UINT8_ZERO            UINT8_C(0)     /**< Zero value */
#define NUMBER_UINT32_ZERO           UINT32_C(0)    /**< Zero value */
#define NUMBER_UINT16_ZERO           UINT16_C(0)    /**< Zero value */
#define NUMBER_INT16_ZERO            INT16_C(0)     /**< Zero value */

#define POINTER_NULL                 NULL          /**< ZERO value for pointers */

#define ENABLED         1
#define DISABLED        0

#warning Provide Default WLAN and MQTT Configuration Here
// Default Network Configuration Settings
#define WLAN_SSID           "YourWLANSSIDHere"            /**< WLAN SSID Name */
#define WLAN_PWD            "YourWLANPasswordHere"        /**< WLAN PWD */
#define MQTT_CLIENT_ID      "YourClientIDHere"            /**< MQTT Client ID */
#define MQTT_BROKER_NAME    "messagesight.demos.ibm.com"  /**< MQTT Broker */
#define MQTT_PORT           1883                          /**< MQTT Port Number */

// Default Data Configuration Settings
#define STREAM_RATE         1000                  /**< Stream Data Rate in MS */
#define ACCEL_EN            DISABLED              /**< Accelerometer Data Enable */
#define GYRO_EN             DISABLED              /**< Gyroscope Data Enable */
#define MAG_EN              DISABLED              /**< Magnetometer Data Enable */
#define ENV_EN              ENABLED               /**< Environmental Data Enable */
#define LIGHT_EN            DISABLED              /**< Ambient Light Data Enable */

/* global function prototype declarations */
```

Figure 1-2. Configuration Header

1.2.2 Configuring Data Collection Settings

The six data collection settings are located near the middle of the header file. These settings define the rate of the data stream and which sensors are enabled to be monitored by the MQTT Broker. Please keep the format as shown in figure 1-2. Below is the list of network settings:

- STREAM_RATE – Publish Rate of Live Data; Whole Number
- ACCEL_EN – Flag indicating if the accelerometer is being monitored
- GYRO_EN – Flag indicating if the gyroscope is being monitored
- MAG_EN – Flag indicating if the magnetometer is being monitored
- ENV_EN – Flag indicating if the environmental sensor (Temperature, Humidity and Pressure) is being monitored
- LIGHT_EN – Flag indicating if the ambient light is being monitored

1.3 Flash the XDK

Connect the XDK to the computer running the XDK workbench via the USB cable provided with the XDK. To insure the XDK comes up on bootloader mode, press and hold button 1 on the XDK, then turn on the XDK by flipping the On/Off Switch to the on position and release button 1.

In the project explorer section of the XDK workbench select the MQTT_Paho project. In the XDK Devices section, select the Flash action button. The XDK will now flash the MQTT_Paho firmware onto the XDK.

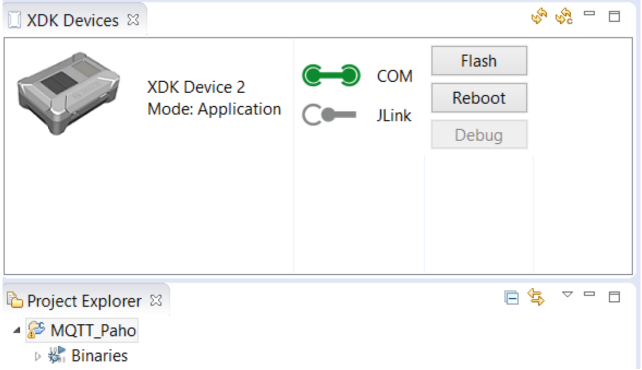


Figure 1-3. Flash the MQTT_Paho firmware on to the XDK

1.4 Demo Operation

After flashing the XDK the program will automatically run. The Red LED will flash once at power up and then all the LEDs will be off while the XDK reads its configuration settings and connects to the wireless network and the MQTT Broker. The Orange LED will turn on after a connection is established. If the LEDs remains off for more than 10 seconds, check your networks settings and confirm the WLAN and MQTT parameters on the XDK are correct.

If the LEDs on the XDK indicate proper connection, navigate to the IBM demo web client at http://m2m.demos.ibm.com/mqttclient/ . The site should be automatically set up to connect to the messagesight server, if not type in the server address (same as the MQTT_BROKER_NAME in the configuration file without the quotation marks) and click connect. Next expand the Subscribe section and subscribe to the following topic(s) ***(NOTE: MQTT_CLIENT_ID matches the defined MQTT_CLIENT_ID in the code without the quotation marks, see section 1.2 for reference):***

- XDK110/<*MQTT_CLIENT_ID*>/Data/Stream – Publishes the enabled data to the broker

There are two ways to publish the data stream. Press button 1 to stream the data at the set timer interval. The log at the bottom of the web client will now show the data being published by the XDK at the given rate. To stop the data stream press button 2. The second way to view the data is through the web client (see the topic definition below).

You can also test the subscribe capabilities of the XDK, by expanding the Publish field in the web client. You can Publish to the following four fields ***(NOTE: MQTT_CLIENT_ID matches the defined MQTT_CLIENT_ID in the code without the quotation marks, see section 1.2 for reference):***

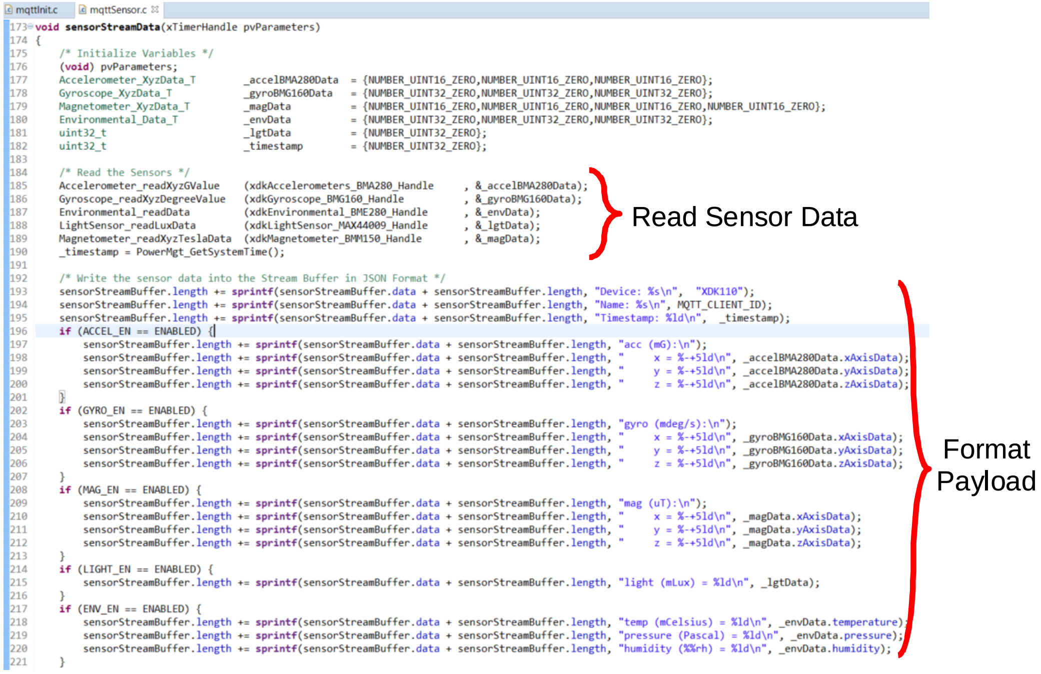
- XDK110/<*MQTT_CLIENT_ID*>/LED/Red - toggles the Red LED once
- XDK110/<*MQTT_CLIENT_ID*>/LED/Orange - toggles the Orange LED once
- XDK110/<*MQTT_CLIENT_ID*>/LED/Yellow - toggles the Yellow LED once
- XDK110/<*MQTT_CLIENT_ID*>/Data/Get – Immediately publishes the enabled data once

2. Expanding the MQTT_Paho Demo

As a demo, the MQTT Paho code gives the user the basic functionality that can be used as a starting point for their final solution. Functionalities such as the Buttons, LEDs, Sensor initialization, and the MQTT publish and subscribe commands are already written and can be easily modified to meet a specific use case. This demo is not, of course, meant to fit any specific end user application. The following sections will describe the major sections of the code that will need to be edited in order to expand the XDK's output payload and subscribe and publish topics. A deeper understanding of the code can be gathered by reviewing the code and the XDK's APIs which are located in the help documents that can be found though the workbench (Help\Help Contents\XDK API Documentation).

2.1 Editing the Data Stream Payload

The data payload that is sent out over the topic XDK110/<*MQTT_CLIENT_ID*>/Data/ can be found in the sensorStreamData function located in the mqttSensor.c source file. This is the timer callback for the data stream timer. This function reads the sensor data, then formats the buffer to send the data out. This can be expanded to include additional values to the payload or remove unnecessary values from the payload. It can also be reformatted to match any specific output file you need (i.e. JSON, CSV, XML, etc.).

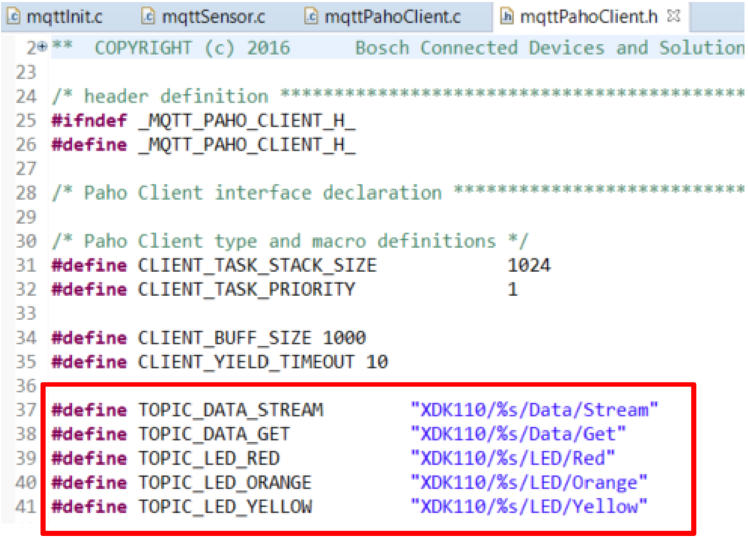


```
void sensorStreamData(xTimerHandle pvParameters)
{
    /* Initialize Variables */
    (void) pvParameters;
    Accelerometer_XyzData_T        _accelBMA280Data  = {NUMBER_UINT16_ZERO,NUMBER_UINT16_ZERO,NUMBER_UINT16_ZERO};
    Gyroscope_XyzData_T            _gyroBMG160Data   = {NUMBER_UINT32_ZERO,NUMBER_UINT32_ZERO,NUMBER_UINT32_ZERO};
    Magnetometer_XyzData_T         _magData          = {NUMBER_UINT16_ZERO,NUMBER_UINT16_ZERO,NUMBER_UINT16_ZERO,NUMBER_UINT16_ZERO};
    Environmental_Data_T           _envData          = {NUMBER_UINT32_ZERO,NUMBER_UINT32_ZERO,NUMBER_UINT32_ZERO};
    uint32_t                       _lgtData          = {NUMBER_UINT32_ZERO};
    uint32_t                       _timestamp        = {NUMBER_UINT32_ZERO};

    /* Read the Sensors */
    Accelerometer_readXyzGValue    (xdkAccelerometers_BMA280_Handle     , &_accelBMA280Data);
    Gyroscope_readXyzDegreeValue   (xdkGyroscope_BMG160_Handle          , &_gyroBMG160Data);
    Environmental_readData         (xdkEnvironmental_BME280_Handle      , &_envData);
    LightSensor_readLuxData        (xdkLightSensor_MAX44009_Handle      , &_lgtData);
    Magnetometer_readXyzTeslaData  (xdkMagnetometer_BMM150_Handle       , &_magData);
    _timestamp = PowerMgt_GetSystemTime();

    /* Write the sensor data into the Stream Buffer in JSON Format */
    sensorStreamBuffer.length += sprintf(sensorStreamBuffer.data + sensorStreamBuffer.length, "Device: %s\n",  "XDK110");
    sensorStreamBuffer.length += sprintf(sensorStreamBuffer.data + sensorStreamBuffer.length, "Name: %s\n", MQTT_CLIENT_ID);
    sensorStreamBuffer.length += sprintf(sensorStreamBuffer.data + sensorStreamBuffer.length, "Timestamp: %ld\n",  _timestamp);
    if (ACCEL_EN == ENABLED) {
        sensorStreamBuffer.length += sprintf(sensorStreamBuffer.data + sensorStreamBuffer.length, "acc (mG):\n");
        sensorStreamBuffer.length += sprintf(sensorStreamBuffer.data + sensorStreamBuffer.length, "     x = %-+5ld\n", _accelBMA280Data.xAxisData);
        sensorStreamBuffer.length += sprintf(sensorStreamBuffer.data + sensorStreamBuffer.length, "     y = %-+5ld\n", _accelBMA280Data.yAxisData);
        sensorStreamBuffer.length += sprintf(sensorStreamBuffer.data + sensorStreamBuffer.length, "     z = %-+5ld\n", _accelBMA280Data.zAxisData);
    }
    if (GYRO_EN == ENABLED) {
        sensorStreamBuffer.length += sprintf(sensorStreamBuffer.data + sensorStreamBuffer.length, "gyro (mdeg/s):\n");
        sensorStreamBuffer.length += sprintf(sensorStreamBuffer.data + sensorStreamBuffer.length, "     x = %-+5ld\n", _gyroBMG160Data.xAxisData);
        sensorStreamBuffer.length += sprintf(sensorStreamBuffer.data + sensorStreamBuffer.length, "     y = %-+5ld\n", _gyroBMG160Data.yAxisData);
        sensorStreamBuffer.length += sprintf(sensorStreamBuffer.data + sensorStreamBuffer.length, "     z = %-+5ld\n", _gyroBMG160Data.zAxisData);
    }
    if (MAG_EN == ENABLED) {
        sensorStreamBuffer.length += sprintf(sensorStreamBuffer.data + sensorStreamBuffer.length, "mag (uT):\n");
        sensorStreamBuffer.length += sprintf(sensorStreamBuffer.data + sensorStreamBuffer.length, "     x = %-+5ld\n", _magData.xAxisData);
        sensorStreamBuffer.length += sprintf(sensorStreamBuffer.data + sensorStreamBuffer.length, "     y = %-+5ld\n", _magData.yAxisData);
        sensorStreamBuffer.length += sprintf(sensorStreamBuffer.data + sensorStreamBuffer.length, "     z = %-+5ld\n", _magData.zAxisData);
    }
    if (LIGHT_EN == ENABLED) {
        sensorStreamBuffer.length += sprintf(sensorStreamBuffer.data + sensorStreamBuffer.length, "light (mLux) = %ld\n", _lgtData);
    }
    if (ENV_EN == ENABLED) {
        sensorStreamBuffer.length += sprintf(sensorStreamBuffer.data + sensorStreamBuffer.length, "temp (mCelsius) = %ld\n", _envData.temperature);
        sensorStreamBuffer.length += sprintf(sensorStreamBuffer.data + sensorStreamBuffer.length, "pressure (Pascal) = %ld\n", _envData.pressure);
        sensorStreamBuffer.length += sprintf(sensorStreamBuffer.data + sensorStreamBuffer.length, "humidity (%%rh) = %ld\n", _envData.humidity);
    }
```

Figure 2-1. sensorStreamData Timer Callback

2.2 Editing and Subscribing to Topics

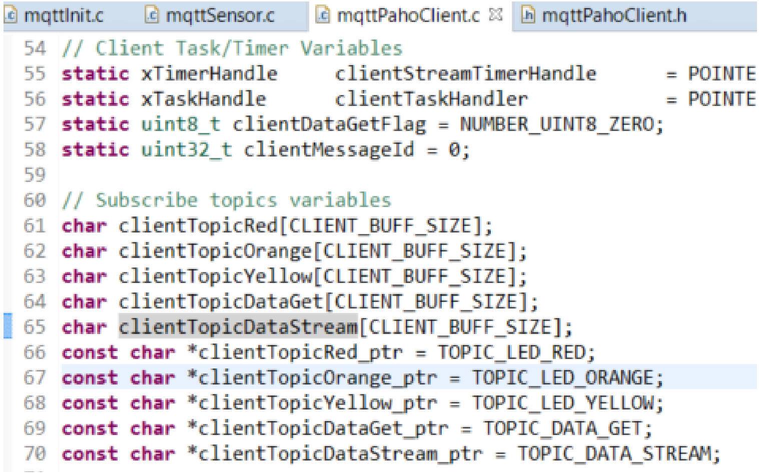
The default topics can also be edited to match user needs. The topic definitions can be found in the header mqttPahoClient.h. To edit the strings simple change the text inside the quotes, leaving the quotes. The %s is currently replaced later, when the code formats the string to include the defined Client ID. To add additional topics simply add a new topic macro. The values don't have to include a %s field to format later, it can be static if desired.



```
**  COPYRIGHT (c) 2016      Bosch Connected Devices and Solution

/* header definition ********************************************
#ifndef _MQTT_PAHO_CLIENT_H_
#define _MQTT_PAHO_CLIENT_H_

/* Paho Client interface declaration ***************************

/* Paho Client type and macro definitions */
#define CLIENT_TASK_STACK_SIZE            1024
#define CLIENT_TASK_PRIORITY              1

#define CLIENT_BUFF_SIZE 1000
#define CLIENT_YIELD_TIMEOUT 10

#define TOPIC_DATA_STREAM       "XDK110/%s/Data/Stream"
#define TOPIC_DATA_GET          "XDK110/%s/Data/Get"
#define TOPIC_LED_RED           "XDK110/%s/LED/Red"
#define TOPIC_LED_ORANGE        "XDK110/%s/LED/Orange"
#define TOPIC_LED_YELLOW        "XDK110/%s/LED/Yellow"
```

Figure 2-2. Topic Strings

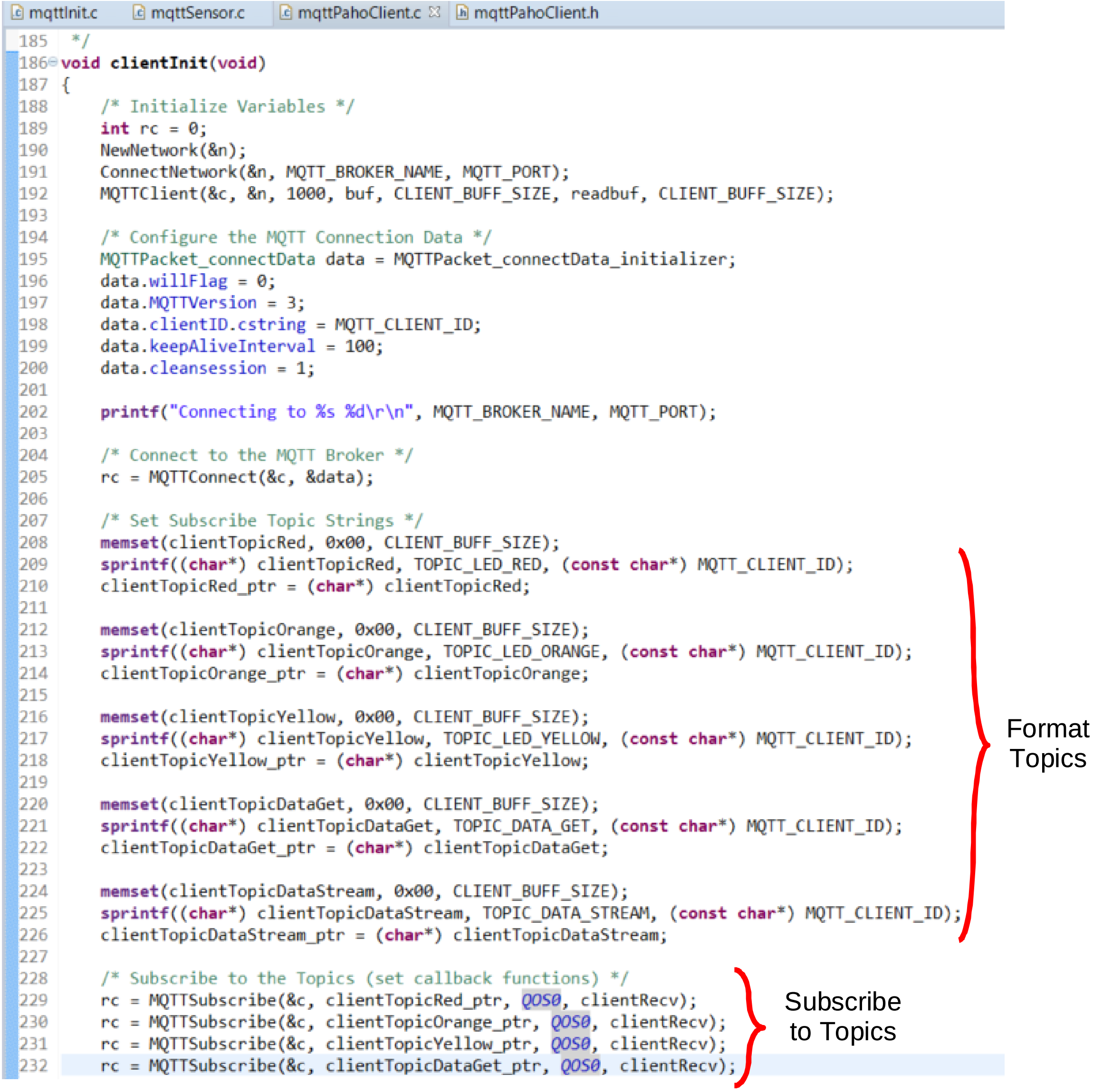
The topics currently defined in the macros are not the final topic the code publishes or subscribes to. Since the topics are edited, the code introduces a couple of variables to format the final topics. These can be found at the top of the mqttPahoClient.c file. The array is used to store the final topic name and the pointer points to this array. If you add another topic you can follow this same format. If the new topic is static (i.e. no %s) you don't need the array but can create a pointer that points straight to the macro.



```
// Client Task/Timer Variables
static xTimerHandle     clientStreamTimerHandle      = POINTE
static xTaskHandle      clientTaskHandler            = POINTE
static uint8_t clientDataGetFlag = NUMBER_UINT8_ZERO;
static uint32_t clientMessageId = 0;

// Subscribe topics variables
char clientTopicRed[CLIENT_BUFF_SIZE];
char clientTopicOrange[CLIENT_BUFF_SIZE];
char clientTopicYellow[CLIENT_BUFF_SIZE];
char clientTopicDataGet[CLIENT_BUFF_SIZE];
char clientTopicDataStream[CLIENT_BUFF_SIZE];
const char *clientTopicRed_ptr = TOPIC_LED_RED;
const char *clientTopicOrange_ptr = TOPIC_LED_ORANGE;
const char *clientTopicYellow_ptr = TOPIC_LED_YELLOW;
const char *clientTopicDataGet_ptr = TOPIC_DATA_GET;
const char *clientTopicDataStream_ptr = TOPIC_DATA_STREAM;
```

Figure 2-2. Topic Variables

The topics are formatted in the clientInit function of the mqttPahoClient.c file. These only need to be formatted if you want to add a variable such as the Client ID to the string. A static topic does not need this step. The format clears the array variable from above, writes the topic string to the array following the printf formatting rules, and sets the pointer to point to the array. In the current code, %s is replaced with the string of the variable MQTT_CLIENT_ID.



```
 */
void clientInit(void)
{
    /* Initialize Variables */
    int rc = 0;
    NewNetwork(&n);
    ConnectNetwork(&n, MQTT_BROKER_NAME, MQTT_PORT);
    MQTTClient(&c, &n, 1000, buf, CLIENT_BUFF_SIZE, readbuf, CLIENT_BUFF_SIZE);

    /* Configure the MQTT Connection Data */
    MQTTPacket_connectData data = MQTTPacket_connectData_initializer;
    data.willFlag = 0;
    data.MQTTVersion = 3;
    data.clientID.cstring = MQTT_CLIENT_ID;
    data.keepAliveInterval = 100;
    data.cleansession = 1;

    printf("Connecting to %s %d\r\n", MQTT_BROKER_NAME, MQTT_PORT);

    /* Connect to the MQTT Broker */
    rc = MQTTConnect(&c, &data);

    /* Set Subscribe Topic Strings */
    memset(clientTopicRed, 0x00, CLIENT_BUFF_SIZE);
    sprintf((char*) clientTopicRed, TOPIC_LED_RED, (const char*) MQTT_CLIENT_ID);
    clientTopicRed_ptr = (char*) clientTopicRed;

    memset(clientTopicOrange, 0x00, CLIENT_BUFF_SIZE);
    sprintf((char*) clientTopicOrange, TOPIC_LED_ORANGE, (const char*) MQTT_CLIENT_ID);
    clientTopicOrange_ptr = (char*) clientTopicOrange;

    memset(clientTopicYellow, 0x00, CLIENT_BUFF_SIZE);
    sprintf((char*) clientTopicYellow, TOPIC_LED_YELLOW, (const char*) MQTT_CLIENT_ID);
    clientTopicYellow_ptr = (char*) clientTopicYellow;

    memset(clientTopicDataGet, 0x00, CLIENT_BUFF_SIZE);
    sprintf((char*) clientTopicDataGet, TOPIC_DATA_GET, (const char*) MQTT_CLIENT_ID);
    clientTopicDataGet_ptr = (char*) clientTopicDataGet;

    memset(clientTopicDataStream, 0x00, CLIENT_BUFF_SIZE);
    sprintf((char*) clientTopicDataStream, TOPIC_DATA_STREAM, (const char*) MQTT_CLIENT_ID);
    clientTopicDataStream_ptr = (char*) clientTopicDataStream;

    /* Subscribe to the Topics (set callback functions) */
    rc = MQTTSubscribe(&c, clientTopicRed_ptr, QOS0, clientRecv);
    rc = MQTTSubscribe(&c, clientTopicOrange_ptr, QOS0, clientRecv);
    rc = MQTTSubscribe(&c, clientTopicYellow_ptr, QOS0, clientRecv);
    rc = MQTTSubscribe(&c, clientTopicDataGet_ptr, QOS0, clientRecv);
```

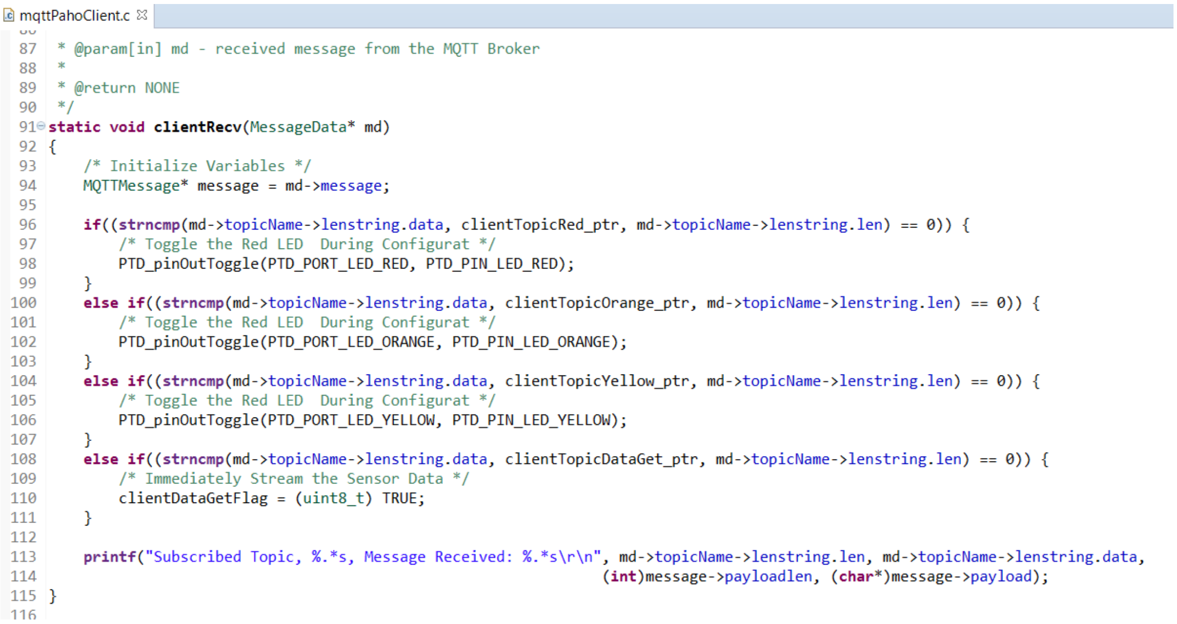
Figure 2-3. Topic Formatting and Subscribing

Finally, if you wish the XDK to subscribe to the topic, the MQTTSubscribe function must be called. This is shown in the clientInit functions just after the topics are formatted as shown in Figure 2-3. ***Note: do not subscribe to topics you wish to publish data on.***

2.3 Receiving Payload from Subscribed Topics

The clientRecv function in the mqttPahoClient.c file is the callback function for subscribed topics. However this callback function can be changed with the last variable of the MQTTSubscribe function call used when subscribing to the topic as shown in figure 2-3. So each topic can have its own callback function if desired.

In this example clientRecv compares the topic name payload with the topic stored in each pointer. If the name payload matches a pointer, the function can now read the message payload and perform specific tasks based on the topic and the message payload. This example doesn't take the message payload into account, rather just looks at the topic and performs tasks such as toggling an LED. However the printf function does print out the topic name and the message payload received to the console window. This can give the user some insight on how to read the message payload to use for their own functions.



```
 * @param[in] md - received message from the MQTT Broker
 *
 * @return NONE
 */
static void clientRecv(MessageData* md)
{
    /* Initialize Variables */
    MQTTMessage* message = md->message;

    if((strncmp(md->topicName->lenstring.data, clientTopicRed_ptr, md->topicName->lenstring.len) == 0)) {
        /* Toggle the Red LED  During Configurat */
        PTD_pinOutToggle(PTD_PORT_LED_RED, PTD_PIN_LED_RED);
    }
    else if((strncmp(md->topicName->lenstring.data, clientTopicOrange_ptr, md->topicName->lenstring.len) == 0)) {
        /* Toggle the Red LED  During Configurat */
        PTD_pinOutToggle(PTD_PORT_LED_ORANGE, PTD_PIN_LED_ORANGE);
    }
    else if((strncmp(md->topicName->lenstring.data, clientTopicYellow_ptr, md->topicName->lenstring.len) == 0)) {
        /* Toggle the Red LED  During Configurat */
        PTD_pinOutToggle(PTD_PORT_LED_YELLOW, PTD_PIN_LED_YELLOW);
    }
    else if((strncmp(md->topicName->lenstring.data, clientTopicDataGet_ptr, md->topicName->lenstring.len) == 0)) {
        /* Immediately Stream the Sensor Data */
        clientDataGetFlag = (uint8_t) TRUE;
    }

    printf("Subscribed Topic, %.*s, Message Received: %.*s\r\n", md->topicName->lenstring.len, md->topicName->lenstring.data,
                                                                 (int)message->payloadlen, (char*)message->payload);
}
```

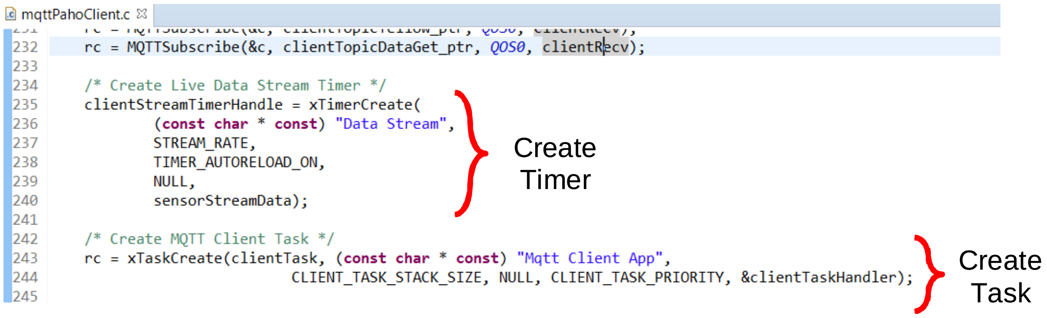
Figure 2-4. Receiving Payload from Subscribed Topics

2.4 Publishing to Topics

You can edit the topics to publish to, by following the same steps outlined in section 2.2. However you should not subscribe to these topics, other clients will subscribe to these topics to receive the data you send out on the topic. There are various ways for the system to determine when to publish to a specific topic (i.e. button presses, timers, tasks, sensor values, variables, etc.). This tutorial outlines one way: a timer function. It should be noted that this tutorial also sets up the buttons in the mqttButton files. In this case the buttons start and stop the timer function, but this can easily be changed so the button presses call on different functions that could publish data to the MQTT Server.

2.4.1 Setting up a Timer

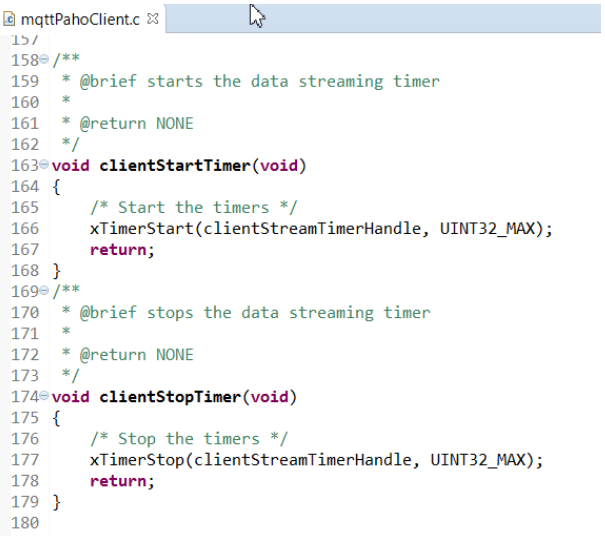
The timer is setup using the xTimerCreate function call. As shown in the clientInit function. The first variable in the functions call names the timer, the second is the period of the timer, the third determines if the timer goes off once or is continuous, the forth is the timer id and the fifth variable is the callback function. The timer create function returns the timer handler.



```
mqttPahoClient.c

    rc = MQTTSubscribe(&c, clientTopicDataGet_ptr, QOS0, clientRecv);

    /* Create Live Data Stream Timer */
    clientStreamTimerHandle = xTimerCreate(
            (const char * const) "Data Stream",
            STREAM_RATE,
            TIMER_AUTORELOAD_ON,
            NULL,
            sensorStreamData);

    /* Create MQTT Client Task */
    rc = xTaskCreate(clientTask, (const char * const) "Mqtt Client App",
                        CLIENT_TASK_STACK_SIZE, NULL, CLIENT_TASK_PRIORITY, &clientTaskHandler);
```

Figure 2-5. clientInit Functions: Timer and Task Setup

After setting up the timer you will need to start the timer for it to run with the xTimerStart function. You can also stop the timer if you choose at some point in your code with the xTimerStop function. In both cases the first variable is the handle to the timer to start or stop and the second variable is the time the calling task will be blocked while the start or stop command is executed. Both functions also return fail or success state. In this example the mqttPahoClient holds these function calls within its own fuctions clientStartTimer and clientStopTimer, respectively.



```
mqttPahoClient.c

/**
 * @brief starts the data streaming timer
 *
 * @return NONE
 */
void clientStartTimer(void)
{
    /* Start the timers */
    xTimerStart(clientStreamTimerHandle, UINT32_MAX);
    return;
}
/**
 * @brief stops the data streaming timer
 *
 * @return NONE
 */
void clientStopTimer(void)
{
    /* Stop the timers */
    xTimerStop(clientStreamTimerHandle, UINT32_MAX);
    return;
}

```

Figure 2-6. Timer Start and Stop Functions

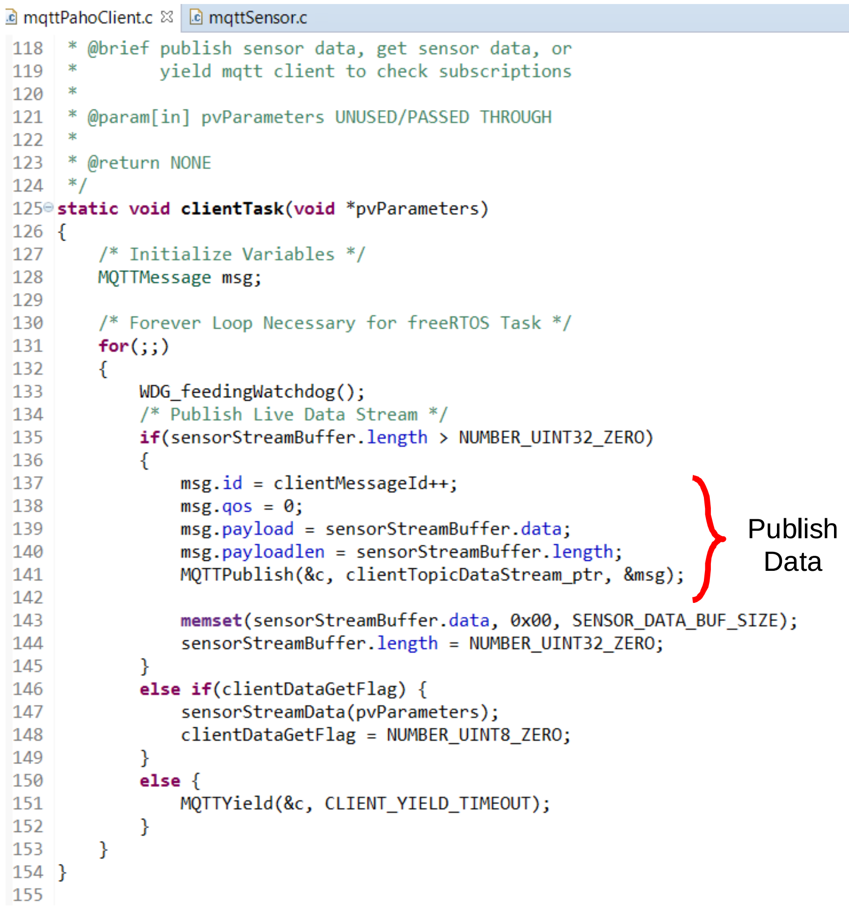
Finally the timer will call the functions defined when it was created. This function will need to be defined and declared in the code. In this example the callback function, sensorStreamData, is located in the mqttSensor files. This functions is described in more detail in section 2.1. This functions simple sets up a buffer and prepares the data to publish.

2.5 Main Task to Publish/Subscribe

The above sections describe how to set up the topics, how to set up the functions to receive the data from topics the XDK is subscribed to and how to set up the data to publish to the MQTT Server.

In this demo there is a main task that is set up to actually publish the data and read if a topics have been written to the Server from another client. The clientInit function in mqttPahoClient.c sets up this task as shown in figure 2-5. The first variable gives the callback function, the second names the task, the third gives the stack size, the fourth is NULL, the fifth is the task priority, and the sixth variable sets up the task handler. This function returns a status.

After this has been setup the task will run yielding to other tasks based on its priority. The task should always be set up as a forever for loop. In this example the task will feed a watchdog timer then check if the buffer from the timer call back function is empty or not. If it is not empty it will publish the data to the MQTT Server then clear the buffer. Otherwise it will check if the flag has been set to immediately get and send sensor data to the server. Finally if nothing else holds true it will yield the client to check if any of the subscribed topics has new data.



```c
 * @brief publish sensor data, get sensor data, or
 *        yield mqtt client to check subscriptions
 *
 * @param[in] pvParameters UNUSED/PASSED THROUGH
 *
 * @return NONE
 */
static void clientTask(void *pvParameters)
{
    /* Initialize Variables */
    MQTTMessage msg;

    /* Forever Loop Necessary for freeRTOS Task */
    for(;;)
    {
        WDG_feedingWatchdog();
        /* Publish Live Data Stream */
        if(sensorStreamBuffer.length > NUMBER_UINT32_ZERO)
        {
            msg.id = clientMessageId++;
            msg.qos = 0;
            msg.payload = sensorStreamBuffer.data;
            msg.payloadlen = sensorStreamBuffer.length;
            MQTTPublish(&c, clientTopicDataStream_ptr, &msg);

            memset(sensorStreamBuffer.data, 0x00, SENSOR_DATA_BUF_SIZE);
            sensorStreamBuffer.length = NUMBER_UINT32_ZERO;
        }
        else if(clientDataGetFlag) {
            sensorStreamData(pvParameters);
            clientDataGetFlag = NUMBER_UINT8_ZERO;
        }
        else {
            MQTTYield(&c, CLIENT_YIELD_TIMEOUT);
        }
    }
}
```

Figure 2-7. clientTask Function

3. Document History and Modification

Rev. No.	Chapter	Description of modification/changes	Date
1.0		Initial Release	2016-04-11
1.2	1; 2	Edited to match new website layout; Added for Clarification	2016-07-06
1.3		Edited for updated website and grammer	2016-08-31
1.4		Edited for grammer	2016-09-30