Project Newton - Total Solution for Smart Home IoT Control

Introduction

"Smart home" is the term used for a wide range of functionalities that enable users to control and manage their home devices using their smartphones, tablets or laptops, for example, control their thermostats remotely from an app on their smartphones. However, many think smart homes have two main stumbling blocks: no unified solution and they're not practical yet. Project Newton, one of Intel's latest innovation projects, can help mitigate these issues.

Problems

"There's no unified solution": Users want to make their own choices. They want to buy smart devices, like TVs, fridges, air-conditionings, from Apple, Samsung, Tencent, Xiaomi, etc. But each manufacturer has its own solution (for example, different manufactures have different communication protocols to follow) for smart homes. It's almost impossible to standardize ecosystems.

"They're not practical yet": Previous smart home IoT control solutions included: Voice control, which might be impacted by background noise; Phone control or other remote controller, which can be inconvenient because it involves fetching a phone, launching the related app, searching for the IoT device you want to control, etc; Gesture recognition via camera, which can depend on the light environment and your position/direction in the room.

Project Newton

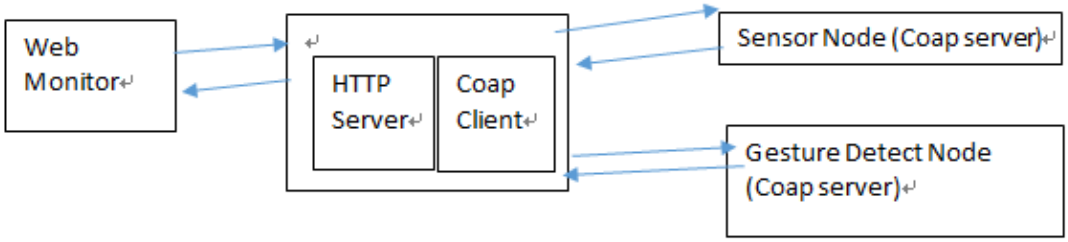
Project Newton is intended to provide an advanced total solution for smart home IoT control.

- Connect any smart home IoT device
- Does not depend on hand-held devices
- Low cost
- Not dependent on the environment, light intensity, noise level, or positional direction of the user

We developed an advanced total solution to smart home IoT control, named Project Newton. This system includes the connection of all main platforms (Intel® Core™ processor, Intel® Centrino* processor technology, Intel® Atom™ processor, ARM* mobile platforms) and all IoT platforms (Intel® Edison board, Intel® Galileo board, Raspberry*, Spark*, Mbed*, Freescale*, Arduino Uno*, etc.). Thus, Project Newton can connect platforms running all current mainstream OSs (Windows*, Linux*, Android*) and NonUI-OSs (Mbed*, Contiki*, RIOT*, Spark*, OpenWRT*, Yocto*, WindRiver*, VxWorks*, Raspbian*, etc.) in real time.

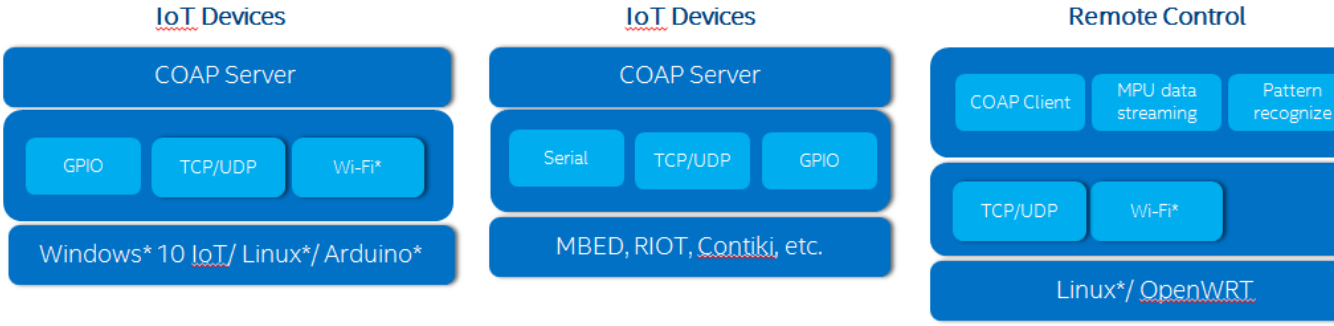
Smart home vendors usually define a set of application layer communication protocols, but

these protocols are relatively closed. In Project Newton, we use CoAP (Constrained Application Protocol), which is a software protocol intended to allow simple electronics devices to communicate interactively over the Internet. CoAP protocol, based on the RESTful framework, is converted into an HTTP protocol to build a smart gateway easily. The basic prototype of CoAP design in Project Newton is below.



CoAP Design of Project Newton

To classify all OSs supported in this idea, Intel® Galileo board, Intel® Edison board, UNO, and Spark boards support Arduino programming standards. Popular IoT LPC1768 boards support Mbed and RIOT operating systems. Mbed is a platform and operating system for Internet-connected devices based on ARM and is convenient for the operation of hardware resources by calling corresponding objects. RIOT is an open source operating system that supports multi-threading and several different development boards. It can be programmed using standard C language. An LPC1768 chip needs to use an external Wi-Fi* module for communication. Windows, Linux, and Android also easily adapt to CoAP. The software architecture of Project Newton is below.



Software Architecture of Project Newton

OS Connection Implementation in Project Newton

- **CoAP Implementation**

The source code for CoAP is open source code called MicroCoAP, which can be downloaded. It's lightweight and written in standard C code, so it's easy to transform it onto different platforms. MicroCoAP contains four files (CoAP.h, CoAP.c, endpoints.c, and main.c). CoAP.h and CoAP.c implement CoAP protocol, which is shown below, and endpoints.c includes response functions related to specific nodes.

```
void coap_dumpPacket(coap_packet_t *pkt);
int coap_parse(coap_packet_t *pkt, const uint8_t *buf, size_t buflen);
int coap_buffer_to_string(char *strbuf, size_t strbuflen, const coap_buffer_t *buf);
const coap_option_t *coap_findOptions(const coap_packet_t *pkt, uint8_t num, uint8_t *count
int coap_build(uint8_t *buf, size_t *buflen, const coap_packet_t *pkt);
void coap_dump(const uint8_t *buf, size_t buflen, bool bare);
int coap_make_response(coap_rw_buffer_t *scratch, coap_packet_t *pkt, const uint8_t *conten
int coap_handle_req(coap_rw_buffer_t *scratch, const coap_packet_t *inpkt, coap_packet_t *o
void coap_option_nibble(uint32_t value, uint8_t *nibble);
void coap_setup(void);
void endpoint_setup(void);
```

The main.c file builds a CoAP server. We mostly use CoAP_packet_t, CoAP_parse, CoAP_handle_req, and CoAP_build. CoAP_packet_t defines the data structure of a CoAP packet. The CoAP_parse function parses the hexadecimal data received from the network or serial port and converts the data to CoAP_packet_t structure. The CoAP_handle_req function analyzes received CoAP packets and makes proper responses. The CoAP_build function translates the response packets into hexadecimal data. The key source code to build a CoAP server is shown below.

```
    // read the packet into packetBufffer
    Udp.read(packetBuffer,UDP_TX_PACKET_MAX_SIZE);
    Serial.println("Contents:");
    //Serial.println(packetBuffer);

    if (0 != (rc = coap_parse(&pkt, packetBuffer, packetSize)))
    {
        Serial.print("Bad packet rc=");
        Serial.println(rc, DEC);
    }
    else
    {
        size_t rsplen = sizeof(packetBuffer);
        coap_packet_t rsppkt;
        coap_handle_req(&scratch_buf, &pkt, &rsppkt);
Serial.print("pkt = ");
Serial.println((char*)pkt.payload.p);
        memset(packetBuffer, 0, UDP_TX_PACKET_MAX_SIZE);
        Serial.print("rsppkt = ");
Serial.println((char*)rsppkt.payload.p);
        if (0 != (rc = coap_build(packetBuffer, &rsplen, &rsppkt)))
        {
            Serial.print("coap_build failed rc=");
            Serial.println(rc, DEC);
        }
        else
        {
            udp_send(packetBuffer, rsplen);
        }
    }
```

- **Arduino* Implementation**

Arduino is an open-source electronic prototyping platform that creates an Arduino software and hardware standard. The Intel® Edison board, the Intel® Galileo board, and the Spark core board follow the Arduino standard. It mainly consists of two functions: setup and loop. Function setup initializes the hardware and is called once before a loop function. Function loop is a dead loop that can be used to execute the main function.

To add COAP to an Arduino board, just add the three files (CoAP.h, CoAP.c, endpoints.c) into

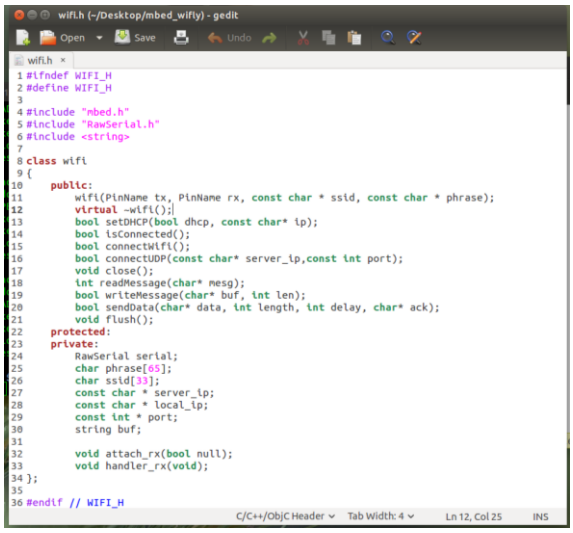
the project and modify the setup and loop functions according to main.c in microCoAP, which can be found below.

CoAP server on Arduino platform*

- **Mbed Implementation**

Mbed is an object-oriented C++ library developed for the ARM Cortex*-M processor. We can operate general purpose input/output (GPIO) and other hardware resources by using a related class. However, there is no default Wi-Fi module or library. Here, the UART-to-WIFI module is used to add Wi-Fi functionality to Mbed. Class WIFI is developed based on the datasheet of UART-WIFI module, and each function will send a relative string to the UART-WIFI module.

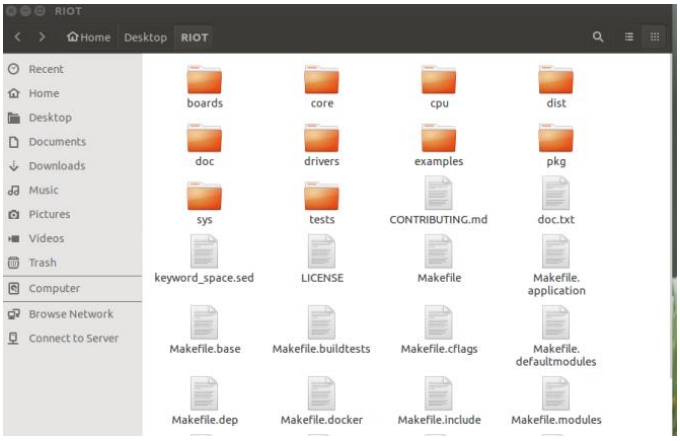
You need to add the three files (CoAP.h, CoAP.c, endpoints.c) into the project and modify main.c file according to main.c file in microCoAP project. To compile it, just use the make command—everything has been set properly in the Makefile, which is shown below.



CoAP server on Mbed platform

- **RIOT Implementation**

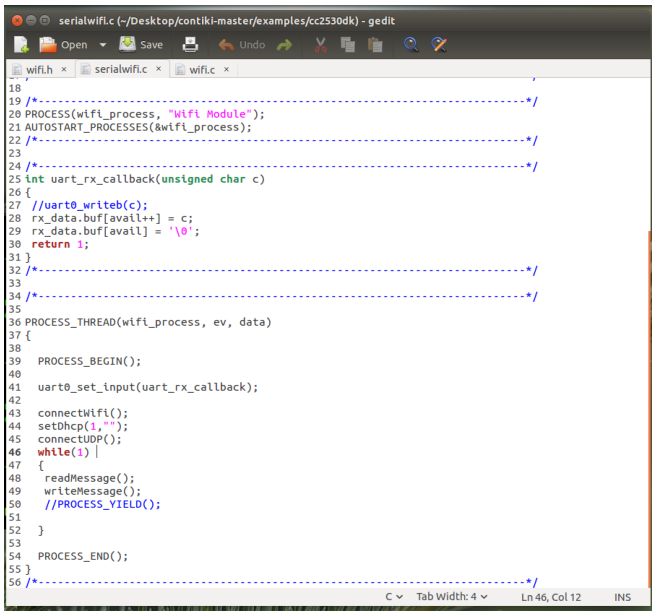
RIOT is an open source developer friendly operating system for IoT that can support several boards, such as the Mbed LPC1768 and the Spark Core development kit. It supports C and C++ language programming. The tests folder contains many sample APIs to connect with the hardware. Similar to Mbed, CoAP server can be implemented by changing the Wi-Fi class of Mbed into a C function and modifying APIs related to the serial port. However, it often crashes in our experiments, so it's just for testing. To compile it, use the "BOARD=Mbed_lpc1768 make clean all flash" command in the root folder and use "BOARD=Mbed_lpc1768 make term" to monitor the serial port and get print information from Mbed. See the CoAP server code structure for RIOT below.



RIOT code structure

- **Contiki Implementation**

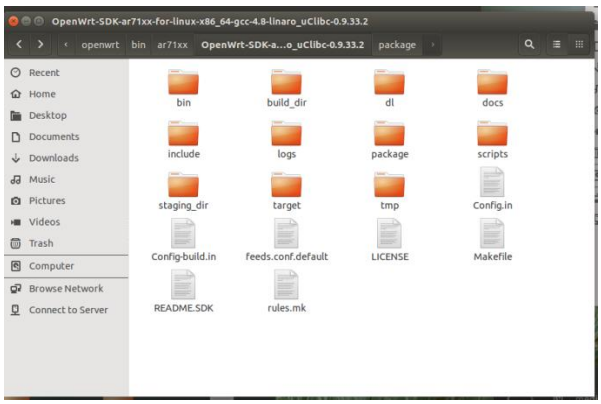
Contiki is a multi-process open source OS. An example folder contains several samples of using APIs to control hardware. It also does not have a default Wi-Fi module. To implement CoAP server, use the similar functions of RIOT and rewrite the UART sample. To compile it, use the "TARGET=cc2530dk make" command and download the binary program using jlink. Check out the sample code below.



Sample code of CoAP server on Contiki

- **OpenWRT Implementation**

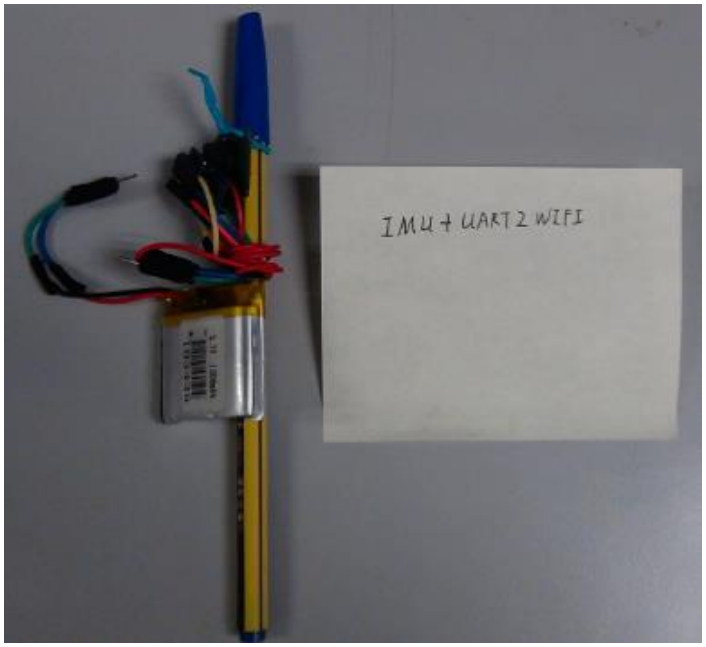
OpenWRT is an eMbedded Linux for routers. It supports standard Linux API. So the microCoAP code can be used directly. In the package folder, set up the microCoAP project and modify the Makefile according to the Makefile in other project. Then, run the make command in the root directory. The compiled app will be in the bin folder. The app can be uploaded into the development board by ftp or usb storage. Finally, use the opkg command install CoAP*.ipk to install the CoAP app, and CoAP will be installed in OpenWRT. Add "/usr/bin/CoAP &" into the /etc/rc.local file. See the CoAP server code structure for OpenWRT below.



OpenWRT code structure

Practical Solution in Project Newton

To be a practical solution in IoT, it must be able to control any IoT device by natural gestures, without wearing any electronic wearable devices and not be affected by environmental light/noise/etc. One way a solution like this could be implemented is by using a 9 axis gyroscope to detect the user's hand motions.



*A sample of 9 axis gyroscope with Wi-Fi**

A pattern recognition algorithm analyzes the data in real time and detects the user's position, direction, and gestures. By calculating the relative position/direction between users and IoT devices, the system can identify which IoT device the user is facing. By further calculating their hand direction and gestures, the targeted IoT device is selected and controlled

Show Cases of Project Newton

Here are the demo show cases in our lab, where Project Newton was used to connect and control a variety of devices with different OSs. These demo show cases include environment (Figure 1), controlling development boards (Intel® Edison board in Figure 2), controlling Android devices (Figure 3), controlling a robot arm (Figure 4), and controlling a robot car (Figure 5).

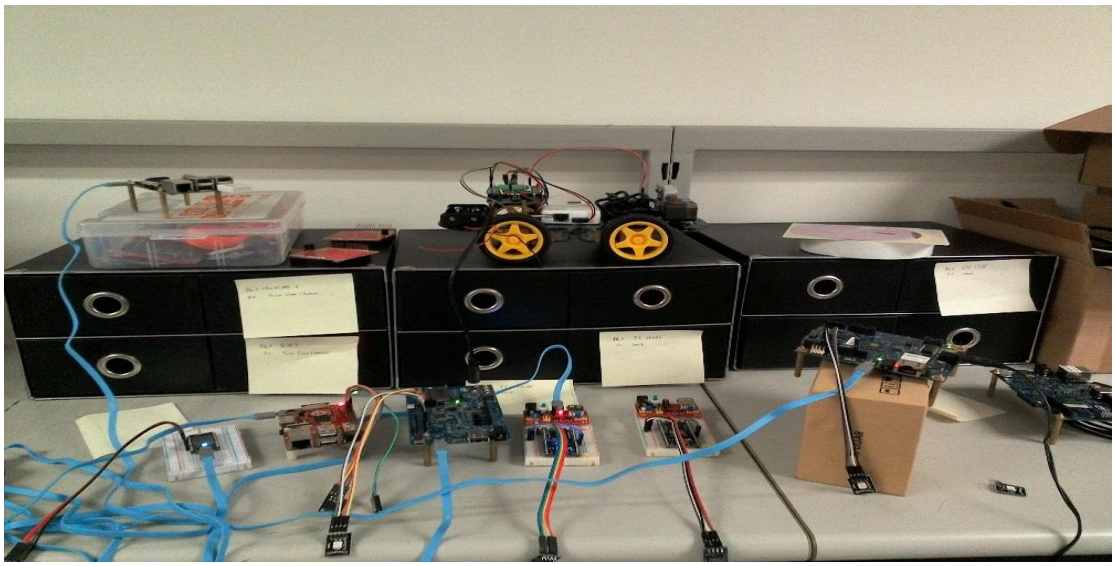


Figure 1: *The demo show cases environment in our lab*

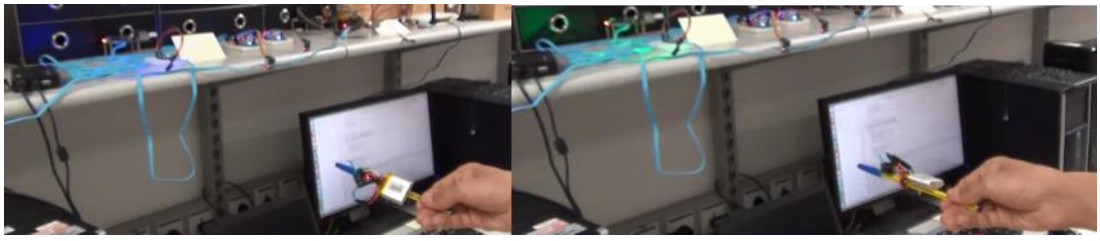


Figure 2: *Select (blue) and control (green) boards in Project Newton*

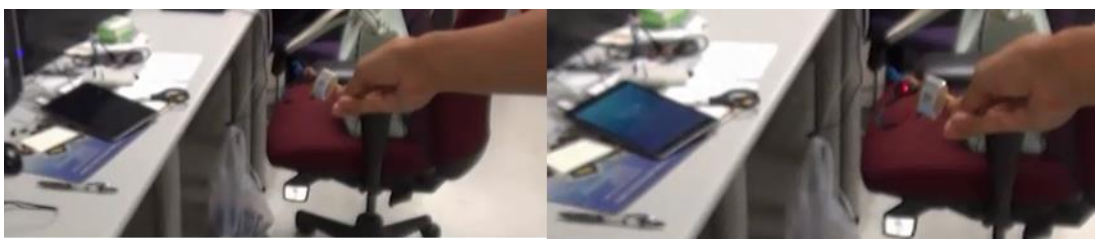


Figure 3: *Control Android devices in Project Newton*

Figure 4: *Control Robot Arm in Project Newton*

Figure 5: *Control Robot Car in Project Newton*

Going Further

Project Newton, an advanced total solution for smart home IoT control, is an easy-to-use solution that gives users the ability to connect any smart home IoT device they want. It's a low-cost solution that frees the user from having to use a hand-held device. And it does not rely on environmental parameters like light, noise level, etc.

However, like all open source projects, Project Newton could be even better. Our next steps are to improve gesture recognition, boost performance, and incorporate wearable devices. For example, the next-generation Google* Glass reportedly will sport a larger prism and will be powered by an Intel Atom processor, which could be a better practical IoT control solution for Project Newton.

About the Author

Zhen Zhou earned an MS degree in software engineering from Shanghai Jiaotong University. He joined Intel in 2011 as an application engineer in the Developer Relations Division Mobile Enabling Team. He works with internal stakeholders and external ISVs, SPs, and carriers in the new usage model initiative and prototype development on the Intel Atom processor, traditional Intel® architecture, and embedded Intel architecture platforms.

granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.