Självständigt arbete på grundnivå

Independent degree project - first cycle

Datateknik
Computer Engineering

Remote measuring and logging
Evaluation of the Quectel OpenCPU technology

Simon Nilsson
Remote measuring and logging - Evaluation of the Quectel OpenCPU technology
Simon Nilsson 2015-06-11

MID SWEDEN UNIVERSITY
Department of Computer and System Science (DSV)

Examiner: Ulf Jennehag, ulf.jennehag@miun.se
Supervisor: Börje Hansson, borje.hansson@miun.se
Author: Simon Nilsson, sini1201@student.miun.se
Degree programme: Programvaruteknik, 180 credits
Main field of study: Computer Engineering
Semester, year: Spring, 2015
Abstract

The company Sensefarm develops in the field of precision agriculture, more specifically a product for measuring and logging information on local environmental conditions. The collected data is transmitted to a remote server using GSM technology. They are looking into ways to improve their product. The purpose of this project is to evaluate the M66 GSM module and the accompanied OpenCPU technology. In the current product the GSM module is only used for the communication with the server. All other processing is done by a dedicated processor. The OpenCPU technology allows for running customized firmware on the GSM module, possibly allowing it to take over the responsibilities of the dedicated processor. An application implementing all the necessary functionality was developed to test this capability. The finished application was able to fulfill all the requirements specified, although it was not able to be tested in the field.

**Keywords:** Precision agriculture, GSM, OpenCPU, C, Embedded system, Sensors
Acknowledgements

I want to give my thanks to the people at Sensefarm for giving me the opportunity to do my thesis work there. Special thanks to Sensefarms founder/CTO Anders Hedberg for taking time in his busy schedule to provide guidance.

I also want to thank Jonathan Bjarnason who also did his thesis at Sensefarm for providing support when things did not go my way.
Table of Contents

Abstract ................................................................. iii
Acknowledgements ....................................................... iv
Terminology ............................................................... vii
1 Introduction ......................................................... 1
  1.1 Background and problem motivation ............................ 1
  1.2 Overall aim .......................................................... 2
  1.3 Scope ........................................................................................... 2
  1.4 Detailed problem statement .............................................. 2
  1.5 Outline ....................................................................................... 3
2 The OpenCPU Platform .................................................. 4
  2.1 Application Programming Interfaces ............................ 4
    2.1.1 System API ................................................................. 4
    2.1.2 Time API .............................................................................. 4
    2.1.3 Timer API ........................................................................... 5
    2.1.4 Power Management API .................................................. 5
    2.1.5 Memory API ......................................................................... 5
    2.1.6 File System API .............................................................. 5
    2.1.7 Hardware Interface API .................................................... 5
    2.1.8 GPRS API ............................................................................ 6
    2.1.9 Socket API ............................................................................ 6
    2.1.10 Watchdog API ............................................................... 6
    2.1.11 FOTA API ........................................................................... 6
    2.1.12 Debug API .......................................................................... 7
    2.1.13 RIL API ............................................................................... 7
  2.2 Example Application ................................................... 7
    2.2.1 Compiling ......................................................................... 9
    2.2.2 Downloading to module ................................................... 9
    2.2.3 Running .......................................................................... 9
3 Methodology ................................................................. 10
  3.1 Documentation ........................................................... 10
3.2 Development ........................................... 10
3.3 Tools ................................................. 10

4 Implementation ......................................... 12
4.1 State management ................................. 12
4.2 IMEI and Battery voltage ....................... 12
4.3 Sensors .............................................. 13
4.4 Global Positioning System ....................... 13
4.5 Data transmission ................................. 14
4.6 Firmware Over-The-Air ........................... 16
4.7 Power Management ................................. 16

5 Results .................................................. 18
5.1 Requirements met .................................. 18
5.2 Requirements partially met ..................... 19

6 Conclusions ............................................. 20
6.1 Challenges during development ............... 20
6.2 Unfinished SDK ...................................... 20
6.3 Ethical aspects ...................................... 20
6.4 Future work ......................................... 21

References ............................................... 22

Appendix A: Source code .............................. 24
Terminology

**ADC** Analog to Digital Converter. 6

**API** Application Programming Interface. 4, 7, 10, 12, 18, 19

**EINT** External Interrupt. 6

**FOTA** Firmware Over-The-Air. 3, 7, 16–18

**FTP** File Transfer Protocol. 16

**GCC** GNU Compiler Collection. 10

**GPIO** General-Purpose Input/Output. 5

**GPRS** General Packet Radio Service. 6, 17

**GPS** Global Positioning System. 3, 12–14, 17, 18

**GSM** Global System for Mobile communications. 1, 4, 7, 16

**HTTP** HyperText Transfer Protocol. 12, 14–18

**I/O** Input/Output. 1, 5

**I^2C** Inter-Integrated Circuit. 3, 6, 13, 17, 18, 20

**IDE** Integrated Development Environment. 10

**IMEI** International Mobile station Equipment Identity. 12, 17

**JSON** JavaScript Object Notation. 12, 14–18

**NMEA** National Marine Electronics Association. 14

**PWM** Pulse-Width Modulation. 6

**RAM** Random Access Memory. 5

**REST** Representational State Transfer. 14
Remote measuring and logging - Evaluation of the Quectel OpenCPU technology
Simon Nilsson

RIL  Radio Interface Layer. 7

RTC  Real Time Clock. 4, 16, 18

SDK  Software Development Kit. 4, 9–11, 16, 20

SPI  Serial Peripheral Interface. 6

TCP  Transmission Control Protocol. 6, 20

UART  Universal Asynchronous Receiver/Transmitter. 5, 11

UDP  User Datagram Protocol. 6

UFS  User File System. 5

URL  Uniform Resource Locator. 14, 16

USB  Universal Serial Bus. 9
1 Introduction

Sensefarm AB [1] is a small but ambitious company based in Lund, Sweden. They develop solutions for remotely measuring and logging information on local environmental conditions. Their products are primarily made for the agricultural sector but have also been adopted by for example a golf course and Malmö Municipality.

The company’s product consists of a base station and a variable amount of connected sensors. The base stations come in two forms, one that is placed underground and is completely reliant on the included battery, and one that is mounted above ground with a solar cell.

The conditions that can be measured include temperature, humidity, soil moisture, precipitation and location. The base stations transmit collected data to a server using GSM-technology where GSM stands for ”Global System for Mobile communications”, the same technology that is used in mobile phones.

The data is presented to the user via a web-interface that can be accessed using a computer or mobile device. The end user can use this information in order to estimate for example, when the next irrigation should be done based on the soil moisture.

1.1 Background and problem motivation

The current solution used in the base stations has an Arduino-processor that takes care of I/O-operations (Input/Output) and processing. The Arduino communicates with a separate GSM-module in order to transmit the data to the server.

The company feels the need to make their product more affordable and want to know if it is possible to combine the functionality of the Arduino-processor and the GSM-module. The GSM-module currently in use is the M10-module developed by a company called Quectel. This module is not programmable and can therefore not fulfill this task.

The M66-module [2] is another GSM-module also made by Quectel. The physical module as seen in Figure 1.1 measures at only 15.8 × 17.7 × 2.3 millimeters. The special thing about this module is that it contains a technology called OpenCPU [3], this technology allows for the programming of custom firmware for the module, possibly allowing it to take over the Arduino-processor’s responsibilities.
1.2 **Overall aim**

The project’s overall aim is to investigate if it is possible for the M66 module in combination with the OpenCPU technology to replace the current solution. In order to accomplish this goal, a software application equivalent to the Arduino counterpart will be developed. The application should preferably work for both product types, the base units with or without solar cells, without any modification to the code.

If successful the Arduino will become redundant and can be excluded entirely. The result will be a smaller, cheaper and more energy efficient solution. Energy efficiency is especially important to improve the product’s battery life.

1.3 **Scope**

The scope of this thesis is limited to the process of developing the firmware and testing it using a provided evaluation kit that has all the necessary hardware. The hardware itself will not be covered in-depth.

1.4 **Detailed problem statement**

In order for the M66 module to work as a replacement the following criteria must be met:

1. Be able to run on a battery for up to three years without human intervention.
2. Be able to communicate with and read information from I²C-devices.

3. Be able to read positioning data from an external GPS-device.

4. Be able to transmit data to the company’s servers and receive a response.

5. Be able to receive updates using Firmware Over-The-Air (FOTA).

6. Be able to power off and wake up at a set time.

If these criteria can be met the M66 module and the OpenCPU platform can be considered an acceptable replacement for the existing solution.

1.5 Outline

Chapter 2 describes the OpenCPU platform and what is needed to get started developing with it. Chapter 3 describes the method used for developing the application. Chapter 4 describes how each major feature of the application was implemented. Chapter 5 summarizes the result of the project and compares the result to the requirement specification. Chapter 6 contains an analysis of the result.
2 The OpenCPU Platform

OpenCPU is what Quectel choose to call their solution that lets the developer upload a custom application to some of their GSM-modules. This is not to be confused with the system for embedded scientific computing and reproducible research also called OpenCPU [4] which is a completely separate technology.

Quectel supplies a user guide for how to use the OpenCPU platform [5]. OpenCPU applications are written using the C programming language. When developing in C one might be used to taking advantage of the standard C libraries like stdio.h and stdlib.h, these are not available here and the programmer is instead limited to the supplied library functions of the SDK. Variants of common functions like atoi() and sprintf() are available through the ql_stdb.h header.

2.1 Application Programming Interfaces

A set of Application Programming Interface (API) functions are included with the SDK. These functions give the developer access to things like hardware interfaces, connection management, a file system, power management, and more. The APIs are separated into multiple categories described below.

2.1.1 System API

Header file: ql_system.h

Contains system-related functionality to get system information and multitasking utilities like mutex and semaphores. Also has the essential functionality required to set up the required message loop.

2.1.2 Time API

Header file: ql_time.h

Functions to get and set the time of the built in Real Time Clock (RTC). The RTC keeps track of the time even when the module is turned off and draws a very low amount of power.
2.1.3 **Timer API**

Header file: *ql_timer.h*

Gives access to timer functionality. There are 2 different kinds of timers available, a common timer and a fast timer. Only one fast timer can be used at a time while you can have 10 common timers simultaneously. The fast timer has the advantage of better accuracy.

2.1.4 **Power Management API**

Header file: *ql_power.h*

Functions for powering off the module and enabling a low power mode (Sleep Mode). Also allows for customization of the PWRKEY input pin.

2.1.5 **Memory API**

Header file: *ql_memory.h*

Allows for dynamic memory management. Has a function for allocating memory and one for freeing memory.

2.1.6 **File System API**

Header file: *ql_fs.h*

Gives access to a User File System (UFS) in the flash storage or ability to store files directly in Random Access Memory (RAM). Functions exists for writing and reading files and directories.

2.1.7 **Hardware Interface API**

Header file: *ql_uart.h*

Universal Asynchronous Receiver/Transmitter (UART) allows for communication using one of the three available UART-ports.

Header file: *ql_gpio.h*

General-Purpose Input/Output (GPIO) allows for individual control of the available I/O-pins.

Header file: *ql_eint.h*
External Interrupt (EINT) allows for registration of interrupt handlers for 2 pins with interrupt support.

Header file: `ql_pwm.h`

Pulse-Width Modulation (PWM) allows for PWM-output on one of the pins.

Header file: `ql_adc.h`

Analog to Digital Converter (ADC) allows measuring external voltage on a pin in the range 0–2800mV.

Header file: `ql_iic.h`

Inter-Integrated Circuit (I²C) allows for communication using the I²C protocol.

Header file: `ql_spi.h`

Serial Peripheral Interface (SPI) allows for communication using the SPI protocol.

### 2.1.8 GPRS API

Header file: `ql_gprs.h`

Functionality for enabling and configuring the GPRS.

### 2.1.9 Socket API

Header file: `ql_socket.h`

Support for creating sockets for User Datagram Protocol (UDP) or Transmission Control Protocol (TCP) communication over GPRS.

### 2.1.10 Watchdog API

Header file: `ql_wtd.h`

For working with an external watchdog timer. Watchdog timers are used to detect and recover from malfunctions in hardware or software.

### 2.1.11 FOTA API

Header file: `ql_fota.h`
2.1.12 **Debug API**

Header file: `ql_trace.h`

Only has one simple function for outputting text to the second (DEBUG) serial port.

2.1.13 **RIL API**

Header file: `ril.h`

Radio Interface Layer (RIL) gives access to the GSM functionality of the module. A lot of the functionality is wrapped in API functions, other features can be accessed with AT commands. Using AT commands is the traditional way to communicate with a GSM module when using an external processor, in that case the commands are transmitted using the serial interface. Using the RIL API, AT commands can be sent in code using a function call.

Header files: `ril_telephony.h, ril_sms.h, ril_network.h, ril_location.h, ril_system.h`

These headers contain wrapper functions around AT commands for their respective areas.

2.2 **Example Application**

Figure 2.1 shows the least code needed for an OpenCPU application. The `proc_main_task()` method is equivalent to the main method in a C application. The application needs to read messages from the message queue. If the messages are not read the queue will fill up and eventually crash the application.
#include "ql_type.h"
#include "ql_system.h"
#include "ql_trace.h"

/**
 * Application main method.
 */
void proc_main_task(s32 taskId)
{
    // Print a message to see that the application works.
    Ql_Debug_Trace("Hello World!\n");

    ST_MSG msg;

    // Message loop
    while(TRUE)
    {
        // Blocks until a message is available in queue.
        Ql_OS_GetMessage(&msg);

        switch(msg.message)
        {
            // Handle any messages relevant to application.
            default:
                break;
        }
    }
}

Figure 2.1: A simple OpenCPU application example.

Messages are read using the Ql_OS_GetMessage() function with a pointer to a ST_MSG structure as a parameter. The definition of ST_MSG can be seen in Figure 2.2. u32 is an unsigned 32-bit integer.

typedef struct {
    u32 message;
    u32 param1;
    u32 param2;
    u32 srcTaskId;
} ST_MSG;

Figure 2.2: The definition of ST_MSG struct.

Messages can be sent by the module to describe its state but customized messages can also be sent by the application itself.
2.2.1 **Compiling**

The SDK comes with a preconfigured makefile and a Windows Batch file. By calling the Batch file with the commands "make clean" and "make new" a new binary file will be produced in the build directory.

2.2.2 **Downloading to module**

When the application has been compiled into a binary file it is ready to be downloaded onto the module. This is done using the software called QFlash supplied by Quectel. A converter cable between Universal Serial Bus (USB) and RS-232 is used to connect to the module. RS-232 is the standard for serial communication. In order to start the download, the following process should be followed:

1. Connect the cable between the MAIN UART port of the evaluation board and a free USB port on your computer.
2. Start QFlash and select the correct serial port.
3. Press "Browse" and select the configuration file that was generated along with the binary file when compiling.
5. Supply power to the evaluation board and put the D/L switch in the ON state.

The download process should now start. The time to complete depends on the size of the application. When the process has finished the module can be turned off and the QFlash application can be closed.

2.2.3 **Running**

After the application has been downloaded to the module it is ready for testing. Supply the module with power, hold the power button pressed down for two seconds, then the module will start. In the case of the example application in Section 2.2 the output through the DEBUG UART port should be as in Figure 2.3.

```
OpenCPU_GS3_SDK_V1.2
Hello World!
```

Figure 2.3: Serial output from example application.
3 **Methodology**

This chapter describes the methodology used when implementing the project.

3.1 **Documentation**

Documentation for the M66 module supplied by Quectel will be studied. More specifically the OpenCPU platform and the accompanied API. Documentation comes in form of PDF-files.

3.2 **Development**

Development is done according to the principles of agile development with focus on implementing one feature at a time. Short meetings are held each morning at the company where the current progress and upcoming steps are discussed.

When all features have been implemented individually they will be combined to form a working implementation to be used as a proof of concept.

Version control with git is used in the form of a private repository on GitHub [6]. This gives several benefits. In addition to the obvious benefits such as keeping track of the code and allowing rollbacks, it also encourages working on one part at a time in order to make meaningful commits.

3.3 **Tools**

The Software Development Kit (SDK) supplied by Quectel for the OpenCPU platform only supports Windows operating systems. Windows 7 will be used as operating system. The SDK comes in two variations, one with the Eclipse Integrated Development Environment (IDE) and a command-line version without any IDE.

The command-line version will be used because it allows for full control and it is not dependant on the editor used. The programming language is C in combination with a GCC compiler. The application gets compiled into a .bin file. This file can be uploaded to the M66 module using a tool called QFlash also supplied by Quectel.

The editor used to write the code is Atom, ”A hackable text editor for the 21st
Century” [7]. This is a text editor that can be extended trough a wide range of packages.

The only way to debug the application once uploaded to the module is trough one of the three on-board UART ports. UART stands for Universal Asynchronous Receiver/Transmitter, and is a standard for serial communication, a common way to communicate by sending characters between two devices. Although any software capable of serial communication should work, an application called QCOM from Quectel will be used as it is a part of the SDK.

The evaluation kit supplied by Quectel has all the necessary hardware required to develop and test applications on the M66 module. The board seen in Figure 3.1 is the main part of the kit. It gives access to all the inputs and outputs of the module and has some on-board hardware for simple connection of power and serial communication. This evaluation board can be used for several different modules. In Figure 3.1 a M10 module is mounted on the board.
4 Implementation

This chapter goes through each part of the implementation. Figure 4.4 shows a flowchart of the application to be implemented.

As C is a procedural language and does not support classes or namespaces, a naming convention was adopted to separate parts of the application. For example, the function to initiate GPS input is named `Sf_GPS_Init()`, `Sf` is the application wide "namespace" and `GPS` means it is a part of the GPS implementation.

The API works as an abstraction layer against the module’s hardware. The real work consists of implementing the collection of data (battery status, IMEI, sensors, GPS), formatting the data so it can be sent over the network (JSON), sending and receiving HTTP-messages, and power management. In addition to this it is very important that the application is stable and prepared for any conditions that might cause malfunction.

The complete source code can be viewed in Appendix A.

4.1 State management

Because the application must constantly listen for new messages in the message loop, only short operations can be reasonably executed in synchronous fashion. Longer operations like data transmission and serial communication is done asynchronously using callbacks. The application needs to keep track of both the state of the module and the state of the application itself. To comply with these requirements some kind of state management is needed.

It made sense to use the API’s built in support for sending customized messages. This way all logic can be run from the message loop. After an operation is completed the application sends a message to itself containing the new state and then returns control to the message loop.

4.2 IMEI and Battery voltage

International Mobile station Equipment Identity (IMEI) [9] is a 15 digit string that can be used to uniquely identify a mobile device. In the case of this application the IMEI is sent along with every request to the server to let the server know which unit is sending the data. No API function is provided to read the IMEI of the module. An AT Command is however available "AT+GSN”, this command
returns the IMEI as a string.

The current battery voltage is also sent along with requests to the server. This allows the user to see the status of the battery and determine if it requires a recharge. An API function is available to get the battery status \textit{RIL\_GetPowerSupply()}, this function did not work as expected and returned an incorrect value, an underlying AT command had to be used here also. In this case it was the ”AT+CBC” command.

4.3 Sensors

To communicate with the sensor Inter-Integrated Circuit (I²C) is used \cite{11}. I²C is a protocol primarily used to communicate between circuits on the same board, but also works for shorter distances using wires. It uses two bidirectional lines, DCA (data) and SCL (clock signal). Even if many other types of sensors have the I²C-protocol in common, they all use different message protocols and therefore require specific code implementations.

Support for one type of sensor was implemented. Although the product is meant to work with multiple sensors, implementing one is enough to evaluate the platform. The SHT2x \cite{10} is a series of sensors for measuring temperature and relative humidity. The series has 3 different models with varying accuracy. The middle variant called SHT21 is used for this implementation. However, since the message protocol of the sensors in the series are identical, they should all be compatible.

The I²C-protocol defines two types of devices, ”master” and ”slave”. The SHT2x can only act as a slave and the module thus takes the role of master. The master device generates the clock signal and initiates communication with a slave device by its address. The address is a 7-bit integer value, in the case of SHT2x the address is the decimal value 80. By sending various commands it is possible to read the sensor’s unique id, the measured temperature, and the measured relative humidity.

There is much more to the I²C protocol which will not be explained here as it is not the subject of this thesis.

4.4 Global Positioning System

Global Positioning System (GPS) should be familiar to most people as it is a technology used frequently in today’s society. It is essentially a system that determines the current position by using satellites. The application should be able to read location data from an external GPS device that is connected to the auxiliary UART-
port. The GPS device used in this implementation is called FGPMOPA6H and is made by GlobalTop Technology. This device outputs data in National Marine Electronics Association (NMEA)-format [12].

The NMEA standard defines several different message strings, called sentences [13]. Each sentence contains different values and not all sentences are supported by every device. This implementation only parses data from the $GPGGA$ and $GPRMC$ sentences from the GPS device.

Longitude, latitude and altitude is read from $GPGGA$. Speed and course is read from $GPRMC$.

4.5 Data transmission

In order to upload gathered data the module needs to communicate with Sense-farms server. The server is running what is known as a Representational State Transfer (REST) service [14] using HyperText Transfer Protocol (HTTP) [15] and the JavaScript Object Notation (JSON) format [16].

To implement HTTP the Socket API was used to create a TCP connection to the server. The protocol is fairly simple to implement. The client (module) sends a request to the server and the server sends a response back through the connection. The messages follow a certain format, starting with headers followed by optional content. The header and content is separated with a double newline. Figure 4.1 shows how the HTTP message might look including the JSON data.

We can see that the request is of type **PUT** and the target Uniform Resource Locator (URL) is **in.example.com/test**. Spacing and newline-characters are added in the JSON data for clarity, but are not included when sent from the application. This is to make the message as small as possible. The **datastreams** array contains one entry for the battery voltage, a temperature reading from a sht2x sensor, and a relative humidity reading from the same sensor.
PUT /test HTTP/1.1
Host: in.example.com
Content-Length: 247
Content-Type: application/json
Connection: close

{
  "version": "1.2.2",
  "IMEI": "863071010199115",
  "datastreams": [
    {
      "id": "Power",
      "val": "4.18"
    },
    {
      "id": "008016B14598010-t",
      "val": "17.0"
    },
    {
      "id": "008016B14598010-r",
      "val": "40.4"
    }
  ]
}

Figure 4.1: HTTP message example.

The response from the server seen in Figure 4.2 also contains JSON. This response only has a number called \( s \) with a value of 3600, which tells the module that the next transmission should be made after 3600 seconds (or 1 hour). This value can be changed via the web-interface.

Connection: close
Content-Type: application/json
Date: Fri, 22 May 2015 12:32:46 GMT
Server: nginx

{
  "s": 3600
}

Figure 4.2: HTTP response from server.
4.6 **Firmware Over-The-Air**

The module has built in support for updating the firmware over-the-air. By using the code included with the SDK it is accomplished simply by calling a function with the URL to a binary file on either a HTTP-server or a File Transfer Protocol (FTP)-server [17]. The module will then download the binary file and install it. When the operation is completed the module will restart and boot up using the new firmware.

As Sensefarm’s current product does not support FOTA there is no existing implementation on the server side. It could be implemented by adding a field to the JSON-response as seen in Figure 4.3.

```json
{
  "s": 3600,
  "update": "http://download.example.com/update.bin"
}
```

Figure 4.3: Proposal for message format when updating firmware.

4.7 **Power Management**

In order to conserve battery the module needs to completely power off between measurements. By using the built in Real Time Clock (RTC) an alarm can be set to power on the module at a certain time. When the RTC alarm fires, the module is powered on in a limited mode called "ALARMT MODE". This mode disables most of the module’s functionality. In order to use GSM functionality the application can switch to a full functionality mode.
Figure 4.4: Application flowchart
5 Results

Out of the six points in the requirement specification, five were fully met and one partially met. Based on these results the M66-module and the OpenCPU platform is a viable replacement for the existing solution.

5.1 Requirements met

The following list shows the requirements that were successfully implemented.

- **Application can communicate with and read information from I²C-devices.**
  
  By using the Hardware Interface API in combination with a code implementation to communicate with the SHT2x sensor this requirement was met.

- **Application can read positioning data from an external GPS-device.**
  
  The application was able to read data from the GPS device using the auxiliary serial port of the module. This data was successfully parsed in order to produce longitude, latitude, altitude, speed and course values.

- **Application can transmit data to the company’s servers and receive a response.**
  
  By using the Socket API in combination with code implementations for HTTP and JSON this requirement was met.

- **Application can receive updates using Firmware Over-The-Air (FOTA).**
  
  The module has built in support for FOTA. The implementation checks the JSON in the server response for the presence of the "update" field, if the field exists a function is called in the modules API in order to start the update.

- **Application can power off and wake up at a set time.**
  
  The wake up functionality was accomplished by setting an alarm in the modules RTC. Powering of the module is as simple as calling a function in
5.2 Requirements partially met

The following list shows the requirement that was not fully met.

- **Be able to run on a battery for up to three years without human intervention.**

  This requirement is met in a theoretical sense but time constraints do not allow for the possibility to test this in practical use.
6 Conclusions

Based on the results of developing using the evaluation kit, I would say that the M66 module and the OpenCPU platform would work very well as a replacement for the current solution.

In order to fully evaluate the advantages of using this new solution a prototype would have to be made using components similar to that used in finished product. This would allow comparisons to be made of power usage, production cost, and physical size of the finished product.

6.1 Challenges during development

One of the major challenges during development was debugging. When something went wrong leading to a crash, the module just started writing a long string of incomprehensible characters to the debug serial output. Without any error messages to rely on, the most effective way to find problems was to place as many debug traces as possible to find the place where the error occurred.

The lack of better debugging support was most noticeable when I was trying to download a larger file using a TCP-socket. Because the code used a busy-waiting loop, the message queue overflowed which resulted in a crash. This took a very long time to figure out as the crash could happen at different times depending on how fast the file was downloading.

6.2 Unfinished SDK

The OpenCPU platform is still relatively new and Quectel is still developing the SDK. This was made evident when I was unable to get either the I²C-functionality or floating point variables to work at all, until a technician from Quectel supplied me with a newer unreleased version of the SDK.

6.3 Ethical aspects

If the solution evaluated in this work is to be implemented in real use it would lead to a product with less components, thus minimizing the environmental impact. The product would also be made more cost effective, allowing wider utilization. A wider utilization of the product would mean that waste of resources could be avoided on a larger scale. By waste of resources I mean things like irrigating
when it is not really needed. This is important because the world’s population is growing meaning there are more and more people in need of food. Farms need to be made as efficient as possible to account for this.

The data sent between the module and server is not encrypted or protected by any security measures beyond those implemented by the network operator, and is thus susceptible to unauthorized access. The data in this case may not be considered very sensitive, but things like location data might not be something you want outsiders to get access to.

### 6.4 Future work

Future work may involve implementing security measures. There is also the possibility to make use of the module’s file-system for intermediate storage to enable frequent logging with occasional transmissions.
References

[1] Sensefarm,  
http://www.sensefarm.com/  


[3] Quectel Wireless Solutions, ”OpenCPU”,  

https://www.opencpu.org/  
Retrieved 2015-05-06.

http://www.quectel.com/qdownload/M66_OpenCPU_SDK.html  

[6] GitHub,  
https://github.com/  

[7] Atom,  
https://atom.io/  

[8] Quectel Wireless Solutions, ”EVB Toolkits”,  

http://en.wikipedia.org/wiki/International_Mobile_Station_Equipment_Identity  

[10] Sensirion, ”SHT2x - Digital Humidity & Temperature Sensor”,  

http://en.wikipedia.org/wiki/I²C

http://en.wikipedia.org/wiki/NMEA_0183

[13] Glenn Baddeley, ”GPS - NMEA sentence information”,
http://aprs.gids.nl/nmea/

http://en.wikipedia.org/wiki/Representational_state_transfer


[16] JSON, ”Introducing JSON”,
http://json.org/

Appendix A: Source code

The file config.h is not included as it contains company sensitive information.

```
main.c
/*
 * main.c
 * *
 * Created on: 27 mar 2015
 * Author: Simon Nilsson
 */
#include "ql_common.h"
#include "ql_system.h"
#include "ql_type.h"
#include "ql_trace.h"
#include "ql_stdio.h"
#include "ql_uart.h"
#include "ql_memory.h"
#include "ql_error.h"
#include "ql_power.h"
#include "ql_gprs.h"
#include "ql_fs.h"
#include "ql_socket.h"
#include "fota_main.h"
#include "ril.h"
#include "ril_util.h"
#include "ril_system.h"
#include "ril_network.h"
#include "sf_config.h"
#include "sf_types.h"
#include "sf_utility.h"
#include "sf_iic.h"
#include "sf_sht21.h"
#include "sf_json.h"
#include "sf_tcp.h"
#include "sf_http.h"
#include "sf_gps.h"
#define MSG_ID_SF_STATE (MSG_ID_USER_START + 100)
typedef enum
{
    SF_STATE_STARTED,
    SF_STATE_RIL_READY,
```
Remote measuring and logging - Evaluation of the Quectel OpenCPU technology
Simon Nilsson 2015-06-11

 SF_STATE_FULL_FUNC,
 SF_STATE_GPRS_READY,
 SF_STATE_DATA_READY,
 SF_STATE_DATA_SENT,
 SF_STATE_DATA_RECEIVED,
 SF_STATE_UPGRADE,
 SF_STATE_FINISHED,
 SF_STATE_FAILURE
} Enum_Sf_State;

char data[512];
char response[515];
s32 dataLength;
s32 sent, received;

// Forward declarations
// ------------------------
static void setState(Enum_Sf_State state);
static void prepareData(void);
static void sendData(void);
static void handleResponse(void);
static void upgrade(const char* url);
static void failure(void);
static void finish(void);

void send_callback(s32 error);
void receive_callback(s32 error);

// ------------------------
static void setState(Enum_Sf_State state)
{
    Ql_Debug_Trace("<-- setState %d -->\n", state);
    Ql_OS_SendMessage(0, MSG_ID_SF_STATE, state, 0);
}

static void prepareData(void)
{
    Ql_Debug_Trace("<-- Preparing data -->\n");
    ST_SENSOR datastreams[20];
u8 streamCount = 0;

    // Add battery voltage as a datastream
    // Power: <voltage>
    Ql_Debug_Trace("<-- Reading battery voltage -->\n");
    float voltage;
    if(Sf_GetBatteryVoltage(&voltage))
    {
        Ql_Debug_Trace("Battery voltage: %f\n", voltage);
        if(voltage < SF_LOW_VOLTAGE_LIMIT)
{  
    Ql_Debug_Trace("Low voltage, powering down!\r\n");
    setState(SF_STATE_FAILURE);
    return;
}

// Set id
Ql_strncpy(datastreams[streamCount].id, "Power");

// Set value
datastreams[streamCount].value = voltage;
++streamCount;
}
else return setState(SF_STATE_FAILURE);

// Read IMEI number.
Ql_Debug_Trace("<-- Fetching IMEI -->\r\n");
char imei[16];
if(Sf_GetIMEI(imei))
    Ql_Debug_Trace("IMEI: %s\r\n", imei);
else
    return setState(SF_STATE_FAILURE);

// Check IIC devices
Ql_Debug_Trace("<-- Checking IIC-devices -->\r\n");
if(Sf_IIC_Init())
{
    // Check for SHT21
    Sf_SHT21_Read(datastreams, &streamCount);
    // Uninitialize IIC
    Sf_IIC_Uninit();
}

// GPS
Sf_GPS_Read(datastreams, &streamCount);
Sf_GPS_Uninit();

// Construct JSON
Ql_Debug_Trace("<-- Building JSON -->\r\n");
dataLength = Sf_JSON_Build(data, imei, datastreams, streamCount);
Ql_Debug_Trace("%s\r\n", data);
setState(SF_STATE_DATA_READY);
}

static void sendData(void)
{
    Ql_Debug_Trace("<-- Sending data -->\r\n");
sent = 0;
received = 0;

// Activate GPRS
if (!Sf_ActivateGPRS())
{
    Ql_Debug_Trace("Failed to activate GPRS!

setState(SF_STATE_FAILURE);
return;
}

if (!Sf_HTTP_SendRequest(SF_HTTP_SERVER, SF_REQUEST_TYPE,
dataLength, send_callback))
{
    Ql_Debug_Trace("Failed to send request!

setState(SF_STATE_FAILURE);
return;
}

static void handleResponse(void)
{
    Sf_DeactivateGPRS();
    Ql_Debug_Trace("<-- Handle response -->

Ql_Debug_Trace("%s\r\n", response);

char update[256];
s32 s, log;

// Parse JSON
Sf_JSON_Parse(response, &s, &log, update);

Ql_Debug_Trace("s: %d\r\n", s);
Ql_Debug_Trace("log: %d\r\n", log);
Ql_Debug_Trace("update: %s\r\n", update);

if(Ql_strlen(update) > 0)
{
    setState(SF_STATE_UPGRADE);
    upgrade(update);
    return;
}

// Set wake up timer.
if(s)
{
    Sf_SetWakeupDelay(s);
    setState(SF_STATE_FINISHED);
}
else
setState(SF_STATE_FAILURE); }

static void upgrade(const char* url) {
    Ql_Debug_Trace("<-- UPGRADING -->\r\n");
    ST_GprsConfig config = { SF_APN, ",", ",", 0, NULL, NULL };;
    s32 err = Ql_FOTA_StartUpgrade((u8*)url, &config, NULL);
    if(err) {
        Ql_Debug_Trace("FOTA update failed!\r\n");
        setState(SF_STATE_FAILURE);
    }
}

static void failure() {
    Ql_Debug_Trace("<-- FAILURE -->\r\n");
    // Retry later
    Sf_SetWakeupDelay(SF_RETRY_DELAY);
    // Power down
    Ql_Debug_Trace("<-- Powering down -->\r\n");
    Ql_PowerDown(1);
}

static void finish() {
    Ql_Debug_Trace("<-- FINISH -->\r\n");
    // Show time spent
    u64 timeSincePwrOn = Ql_GetMsSincePwrOn();
    Ql_Debug_Trace("<-- Operation took: %dms -->\r\n", (u32)
        timeSincePwrOn);
    // Power down
    Ql_Debug_Trace("<-- Powering down -->\r\n");
    Ql_PowerDown(1);
}

/**
 * Application Entry Point
 * @param taskId The task id.
 */
void proc_main_task(s32 taskId) {
    Ql_Debug_Trace("Sensefarm Application 1.0\r\n");
    Ql_Debug_Trace("Made by: Simon Nilsson 2015\r\n");
    Enum_Sf_State state = SF_STATE_STARTED;
// Start listening to GPS
sf_GPS_Init();

ST_MSG msg;

// Message loop
while (TRUE) {
    Ql_OS_GetMessage(&msg);
    if(msg.message == MSG_ID_SF_STATE) {
        state = msg.param1;
        switch (state) {
            case SF_STATE_FULL_FUNC:
                prepareData();
                break;
            case SF_STATE_GPRS_READY:
                sendData();
                break;
            case SF_STATE_DATA_SENT:
                Sf_HTTP_GetResponse(receive_callback);
                break;
            case SF_STATE_DATA_RECEIVED:
                handleResponse();
                break;
            case SF_STATE_FAILURE:
                failure();
                return;
            case SF_STATE_FINISHED:
                finish();
                return;
            default:
                break;
        }
    } else if(msg.message == MSG_ID_RIL_READY) {
        Ql_Debug_Trace("<-- RIL is ready -->\r\n");
        Ql_RIL_Initialize();
        Ql_Debug_Trace("<-- Synchronizing time -->\r\n");
        Sf_SyncTime();
    }
setState(SF_STATE_RIL_READY);
}

else if(msg.message == MSG_ID_URC_INDICATION)
{
    switch (msg.param1)
    {
        case URC_CFUN_STATE_IND:
            Ql_Debug_Trace("<-- CFUN Status:%d -->\r\n", msg.
            param2);
            // Minimum functionality (when started from ALARM)
            if(msg.param2 == 0)
            {
                Ql_Debug_Trace("<-- Enabling full functionality -->\r\n");
                Sf_EnableFullFunctionality();
            }
            setState(SF_STATE_FULL_FUNC);
            break;
        case URC_SYS_INIT_STATE_IND:
            Ql_Debug_Trace("<-- Sys Init Status %d -->\r\n", msg.
            param2);
            break;
        case URC_SIM_CARD_STATE_IND:
            if(SIM_STAT_READY == msg.param2)
            {
                Ql_Debug_Trace("<-- SIM card is ready -->\r\n");
            }
            else
            {
                Ql_Debug_Trace("<-- SIM card is not available, cause
                :%d -->\r\n", msg.param2);
                /* cause: 0 = SIM card not inserted
                * 2 = Need to input PIN code
                * 3 = Need to input PUK code
                * 9 = SIM card is not recognized
                */
            }
            break;
        case URC_GSM_NW_STATE_IND:
            break;
        case URC_GPRS_NW_STATE_IND:
            break;
    }
}
if (NW_STAT_REGISTERED == msg.param2 || msg.param2 == NW_STAT_NOT_ACTIVE)
{
    Ql_Debug_Trace("<- GPRS ready -->\n\n");
    if (state == SF_STATE_DATA_READY)
        setState(SF_STATE_GPRS_READY);
    break;
} else {
    Ql_Debug_Trace("<- GPRS network status:%d -->\n\n", msg.param2);
    /* status: 0 = Not registered, module not currently
       search a new operator
     *  2 = Not registered, but module is
       currently searching a new operator
     *  3 = Registration denied
     */
    u32 rssi;
    u32 ber;
    RIL_NW_GetSignalQuality(&rssi, &ber);
    Ql_Debug_Trace("<- Signal strength:%d, BER:%d -->\n\n", rssi, ber);
    break;
}
default:
    Ql_Debug_Trace("<- Other URC: type=%d\r\n", msg.
    param1);
    break;
}
}
void send_callback(s32 error)
{
    if (error)
    {
        Ql_Debug_Trace("Failed to send data! [%d]\r\n", error);
        Sf_DeactivateGPRS();
        return;
    }
    // Send header
while(sent < dataLength)
{
    s32 ret = Sf_TCP_Send((u8*)data + sent, dataLength - sent);
    // Blocking
    if(ret == SOC_WOULDBLOCK)
        return;
    // Error
    else if(ret < 0)
    {
        Ql_Debug_Trace("Failed to send data! [%d]\r\n", ret);
        Sf_DeactivateGPRS();
        setState(SF_STATE_FAILURE);
        return;
    }
    sent += ret;
}
setState(SF_STATE_DATA_SENT);

void receive_callback(s32 error)
{
    if(error)
    {
        Ql_Debug_Trace("Failed to receive data! [%d]\r\n", error);
        setState(SF_STATE_FAILURE);
        return;
    }
    // Read header
    s32 ret;
    do
    {
        ret = Sf_TCP_Recv((u8*)response + received, 511 - received);
        // Blocking
        if(ret == SOC_WOULDBLOCK)
            return;
        // Error
        else if(ret < 0)
        {
            Ql_Debug_Trace("Failed to receive data! [%d]\r\n", ret);
            setState(SF_STATE_FAILURE);
            return;
        }
        received += ret;
    }
Remote measuring and logging - Evaluation of the Quectel OpenCPU technology
Simon Nilsson

447 while(ret != 0);
448
449 // Terminate response string.
450 response[received] = '\0';
451 setState(SF_STATE_DATA_RECEIVED);
452 }
453
454 #endif

sf_types.h

1 /*
2 * sf_types.h
3 *
4 * Created on: 7 apr 2015
5 * Author: Simon Nilsson
6 */
7
8 #ifndef SF_TYPES_H_
9 #define SF_TYPES_H_
10
11 #include <math.h>
12
13 /**
14 * Structure to hold a id/value-pair.
15 */
16 typedef struct {
17     char id[20];
18     float value;
19 } ST_SENSOR;
20
21 #endif /* SF_TYPES_H_ */

sf_utility.c

1 /*
2 * sf_utility.c
3 *
4 * Created on: 30 mar 2015
5 * Author: Simon Nilsson
6 */
7
8 #include "ril.h"
9 #include "ril_util.h"
10 #include "ql_trace.h"
11 #include "ql_stdio.h"
12 #include "ql_time.h"
13 #include "ql_error.h"
14
15 #include "sf_utility.h"
16
17 typedef enum {

33
REPEAT_NONE = 0,
REPEAT_DAY = 1,
REPEAT_WEEK = 2,
REPEAT_MONTH = 3
}
}

typedef enum {
    POWER_NONE = 0,
    POWER_OFF = 1,
    POWER_ON = 2
} Enum_AlarmPower;

static s32 imei_response(char* line, u32 len, void* userdata)
{
    char* imei = userdata;

    if (Ql_RIL_FindLine(line, len, "ERROR")
        || Ql_RIL_FindString(line, len, "+CME ERROR:"
        || Ql_RIL_FindString(line, len, "+CMS ERROR:"))
    {
        return RIL_ATRSP_FAILED;
    }
    else if(len > 0)
    {
        // Copy data
        Ql_strncpy(imei, line + 2, len - 4);
        // Terminate string
        Ql_strcpy(imei + (len - 4), "\0");
        return RIL_ATRSP_SUCCESS;
    }
    else
    {
        return RIL_ATRSP_CONTINUE; //continue to wait
    }
}

static s32 battery_response(char *line, u32 len, void *userdata)
{
    float *voltage = userdata;

    char *head = Ql_RIL_FindString(line, len, "+CBC:";
    if(head)
    {
        char strTmp[10];
        char *start, *end;

        // Seek past first parameter
        head = Ql_strlen(head, ",");

        // Seek past second parameter
        start = Ql_strchr(head + 1, "\");

        // ...
// Find end
end = Ql_strstr(start + 1, "\r\n");

if (start && end)
{
    Ql_memset(strTmp, 0x0, sizeof(strTmp));
    Ql_memcpy(strTmp, start + 1, end - start - 1);
    *voltage = Ql_atof(strTmp) / 1000.0;
}
return RIL_ATRSP_CONTINUE;

head = Ql_RIL_FindLine(line, len, "OK");
if (head)
{
    return RIL_ATRSP_SUCCESS;
}

head = Ql_RIL_FindLine(line, len, "ERROR");
if (head)
{
    return RIL_ATRSP_FAILED;
}

head = Ql_RIL_FindString(line, len, "+CME ERROR:");
if (head)
{
    return RIL_ATRSP_FAILED;
}
return RIL_ATRSP_CONTINUE;

bool Sf_ErrorCheck(s32 ret, s32 expect, const char* msg)
{
    if (ret != expect)
    {
        Ql_Debug_Trace("%s [%d]\r\n", msg, ret);
        return FALSE;
    }
    return TRUE;
}

bool Sf_GetIMEI(char* imei)
{
    return Sf_ErrorCheck(
        Ql_RIL_SendATCmd("AT+GSN", 6, imei_response, imei, 300),
        RIL_AT_SUCCESS,
        "Failed to get IMEI!"
    );
}
bool Sf_GetBatteryVoltage(float *voltage)
{
    
    return Sf_ErrorCheck(
        Ql_RIL_SendATCmd("AT+CBC", 6, battery_response, voltage, 300),
        RIL_AT_SUCCESS,
        "Failed to get battery voltage!"
    );
}

bool Sf_SyncTime(void)
{
    
    return Sf_ErrorCheck(
        Ql_RIL_SendATCmd("AT+CTZU=3", 9, NULL, NULL, 300),
        RIL_AT_SUCCESS,
        "Failed to synchronize local time!"
    );
}

bool Sf_EnableFullFunctionality(void)
{
    
    return Sf_ErrorCheck(
        Ql_RIL_SendATCmd("AT+CFUN=1", 9, NULL, NULL, 15000),
        RIL_AT_SUCCESS,
        "Failed to enable full functionality!"
    );
}

s32 Sf_SetWakeupDelay(u32 seconds)
{
    ST_Time currentTime, wakeUpTime;

    // Get current ST_TIME.
    Ql_GetLocalTime(&currentTime);

    Ql_Debug_Trace("<-- Time: %d/%d/%d %d:%d:%d %d --\r\n",
        currentTime.year,
        currentTime.month,
        currentTime.day,
        currentTime.hour,
        currentTime.minute,
        currentTime.second,
        currentTime.timezone
    );

    // Convert ST_TIME to seconds.
    u64 currentMKTime = Ql_Mktime(&currentTime);

    // Make a new ST_TIME offset by seconds parameter.
    Ql_MKTime2CalendarTime(currentMKTime + seconds, &wakeUpTime)
    ;
}
// Build AT command.
char cmd[39];
Ql_sprintf(
    cmd,
    "AT+QALARM=1,%02d/%02d/%02d,%02d:%02d:%02d%03d",%d,%d",
    wakeUpTime.year % 100,
    wakeUpTime.month,
    wakeUpTime.day,
    wakeUpTime.hour,
    wakeUpTime.minute,
    wakeUpTime.second,
    wakeUpTime.timezone,
    REPEAT_NONE,
    POWER_ON
);

// Send command.
s32 ret = QL_RIL_SendATCmd(cmd, 38, NULL, NULL, 300);
if (ret != QL_RET_OK)
{
    Ql_Debug_Trace("Failed to set alarm! [%d]\r\n", ret);
}
else
{
    Ql_Debug_Trace("Module set to wake up in %d seconds.\r\n",
    (u32)seconds);
    return ret;
}

sf_utility.h

/*
 * sf_utility.h
 *
 * Created on: 30 mar 2015
 * Author: Simon Nilsson
 */

#ifndef SF_UTILITY_H_
#define SF_UTILITY_H_

#include <math.h>
#include "ql_type.h"

/**
 * Used to check return value from function.
 * @param ret Value returned by API function.
 * @param expect Expected value.
 * @param msg Error message.
 * @return TRUE if expected return value.
 */
bool Sf_ErrorCheck(s32 ret, s32 expect, const char* msg);
remotemeasuringandlogging-EvaluationoftheQuectelOpenCPU

technology
SimonNilsson 2015-06-11

23 /*
24 * Gets the modules IMEI-number.
25 * @param imei [out] IMEI-number, 16 bytes including null
26 * termination.
27 * @return TRUE if successful.
28 */
29 bool Sf_GetIMEI(char* imei);
30
31 /*
32 * Gets the battery voltage level in Volts.
33 * @param voltage [out] voltage level.
34 * @return TRUE if successful.
35 */
36 bool Sf_GetBatteryVoltage(float*voltage);
37
38 /*
39 * Requests network time synchronization.
40 * @return TRUE if successful.
41 */
42 bool Sf_SyncTime(void);
43
44 /*
45 * Enables full functionality mode.
46 * @return TRUE if successful.
47 */
48 bool Sf_EnableFullFunctionality(void);
49
50 /*
51 * Sets an alarm to wake up the module in a certain number of
52 * seconds.
53 * @param seconds Seconds until alarm.
54 * @return 0 on success or a value from Enum_ATSndError.
55 */
56 s32 Sf_SetWakeupDelay(u32 seconds);
57
58 #endif /* SF.Utility.H_*/
bool Sf_GPS_Init(void);

void Sf_GPS_Uninit(void);

bool Sf_GPS_Read(ST_SENSOR datastreams[], u8 *streamCount);

bool Sf_GPS_GetActive(void);

float Sf_GPS_GetLatitude(void);

float Sf_GPS_GetLongitude(void);

float Sf_GPS_GetAltitude(void);

float Sf_GPS_GetSpeed(void);

float Sf_GPS_GetCourse(void);
```c
#define UART_BUFFER_SIZE 83

static char uart_buffer[UART_BUFFER_SIZE];
static char gprmc[81];
static char gpgga[81];

static void uart_callback(Enum_SerialPort port, Enum_UARTEventType msg, bool level, void* customizedPara)
{
    if(msg == EVENT_UART_READY_TO_READ)
    {
        Ql_memset(uart_buffer, 0, UART_BUFFER_SIZE);
        char c = 0;
        while(Ql_UART_Read(port, (u8*)&c, 1))
        {
            if(c == '\r')
            {
                if(Ql_strstr(uart_buffer, "$GPRMC"))
                {
                    Ql_strcpy(gprmc, uart_buffer);
                }
                else if(Ql_strstr(uart_buffer, "$GPGGA"))
                {
                    Ql_strcpy(gpgga, uart_buffer);
                }
            }
            else if(c == '\n')
            {
                Ql_memset(uart_buffer, 0, UART_BUFFER_SIZE);
            }
        }
    }
```
else
{
    Ql_strncat(uart_buffer,&c,1);
}
}
}
}
}
}

static bool get_field(char *sentence, char *buffer, s8 position, s8 size)
{
    Ql_memset(buffer, 0, size);
    char *start = sentence;
    for(u8 i = 0; i < position; i++)
    {
        start = Ql_strchr(start,','');
        if(start)
            start++;
        else
            return FALSE;
    }
    char *del = Ql_strchr(start,'','');
    s8 len;
    if(del)
        len = del - start;
    else
        len = Ql_strlen(start);
    if(len > 0 && len <= size)
    {
        Ql_memcpy(buffer, start, len);
        return TRUE;
    }
    return FALSE;
}

bool Sf_GPS_Init(void)
{
    if(!Sf_ErrorCheck(
        Ql_UART_Register(UART_PORT3, uart_callback, NULL),
        QL_RET_OK,
        "Failed to register UART for GPS!"
    )) return FALSE;
    return Sf_ErrorCheck(

Ql_UART_Open(UART_PORT3, 9600, FC_NONE),
QL_RET_OK,
"Failed to open UART for GPS!"
);
}

void Sf_GPS_Uninit(void)
{
  Ql_UART_Close(UART_PORT3);
}

bool Sf_GPS_Read(ST_SENSOR datastreams[], u8 *streamCount)
{
  if(Sf_GPS_GetActive())
  {
    // Latitude
    Ql_strcpy(datastreams[*streamCount].id, "lat");
datastreams[*streamCount].value = Sf_GPS_GetLatitude();
*streamCount += 1;

    // Longitude
    Ql_strcpy(datastreams[*streamCount].id, "long");
datastreams[*streamCount].value = Sf_GPS_GetLongitude();
*streamCount += 1;

    // Altitude
    Ql_strcpy(datastreams[*streamCount].id, "alt");
datastreams[*streamCount].value = Sf_GPS_GetAltitude();
*streamCount += 1;

    // Speed
    Ql_strcpy(datastreams[*streamCount].id, "speed");
datastreams[*streamCount].value = Sf_GPS_GetSpeed();
*streamCount += 1;

    // Course
    Ql_strcpy(datastreams[*streamCount].id, "course");
datastreams[*streamCount].value = Sf_GPS_GetCourse();
*streamCount += 1;

    return TRUE;
  }

  return FALSE;
}

bool Sf_GPS_GetActive(void)
{
  char status;
  return (get_field(gprmc, &status, 2, 1) && status == 'A');
}
float Sf_GPS_GetLatitude(void)
{
    char lat[10];
    char dir;
    if(get_field(gprmc, lat, 3, 10))
    {
        char degrees[3] = "";
        Ql_strncpy(degrees, lat, 2);
        float latitude = Ql_atoi(degrees) + (Ql_atof(lat + 2) / 60);
        if(get_field(gprmc, &dir, 4, 1) && dir == 'S')
            latitude = -latitude;
        return latitude;
    }
    return 0;
}

float Sf_GPS_GetLongitude(void)
{
    char lng[11];
    char dir;
    if(get_field(gprmc, lng, 5, 11))
    {
        char degrees[4] = "";
        Ql_strncpy(degrees, lng, 3);
        float longitude = Ql_atoi(degrees) + (Ql_atof(lng + 3) / 60);
        if(get_field(gprmc, &dir, 6, 1) && dir == 'W')
            longitude = -longitude;
        return longitude;
    }
    return 0;
}

float Sf_GPS_GetAltitude(void)
{
    char alt[11];
    if(get_field(gpgga, alt, 9, 11))
    {
        return Ql_atof(alt);
    }
}
Remote measuring and logging - Evaluation of the Quectel OpenCPU technology
Simon Nilsson 2015-06-11

return 0;
}

float Sf_GPS_GetSpeed(void)
{
    char knots[11];
    if(get_field(gprmc, knots, 7, 11))
    {
        return Ql_atof(knots);
    }
    return 0;
}

float Sf_GPS_GetCourse(void)
{
    char course[11];
    if(get_field(gprmc, course, 8, 11))
    {
        return Ql_atof(course);
    }
    return 0;
}

sf_iic.h

_REMOVE

#ifndef SF_IIC_H_
#define SF_IIC_H_
#include "ql_type.h"
#include "ql_gpio.h"

/**
 * Writes data to IIC bus.
 * @param address 7-bit address (upper 7 bits).
 * @param data The data.
 * @param length Length of data.
 * @return If no error, return the length of the write data.
 *         Negative integer indicates this function fails.
 */
s32 Sf_IIC_Write(u8 address, u8* data, u32 length);

/**
 * Reads data from IIC bus.
 * @param address 7-bit address (upper 7 bits).
 * @param buffer [out] Buffer to hold data.
 * @param length Bytes to read.
 * @return If no error, return the length of the read data.
 * Negative integer indicates this function fails.
 */
s32 Sf_IIC_Read(u8 address, u8* buffer, u32 length);

/**
 * Combines a write and read operation on IIC bus.
 * @param address 7-bit address (upper 7 bits).
 * @param data The data.
 * @param dataLength Length of data.
 * @param buffer [out] Buffer to hold data.
 * @param length Bytes to read.
 * @return If no error, return the length of the read data.
 * Negative integer indicates this function fails.
 */
s32 Sf_IIC_WriteRead(u8 address, u8 *data, s32 dataLength, u8 *buffer, u32 length);

/**
 * Initializes the IIC bus.
 * @return TRUE if successful.
 */
bool Sf_IIC_Init(void);

/**
 * Configures the IIC bus for using address.
 * @param address 7-bit address (upper 7 bits).
 * @return TRUE if successful.
 */
bool Sf_IIC_Config(u8 address);

/**
 * Uninitializes the IIC bus.
 * @return TRUE if successful.
 */
bool Sf_IIC_Uninit(void);

#endif /* SF_IIC_H_ */
Remote measuring and logging - Evaluation of the Quectel OpenCPU technology
Simon Nilsson 2015-06-11

* sf_iic.c
*
* Created on: 1 apr 2015
* Author: Simon Nilsson
*/

#include "ql_type.h"
#include "ql_gpio.h"
#include "ql_iic.h"
#include "ql_error.h"
#include "ql_trace.h"
#include "ql_stdlib.h"
#include "ql_system.h"
#include "sf_iic.h"
#include "sf_utility.h"

/**
 * Channel, can be 0~254.
 */
static const u32 IIC_CHANNEL = 0;
/**
 * Controller only supports reading 7 bytes in one read,
 * not enough for SHT21.
 */
static const u32 IIC_USE_CONTROLLER = FALSE;
/**
 * Speed is only relevant if using controller
 */
static const u32 IIC_SPEED = 100;

s32 Sf_IIC_Write(u8 address, u8* data, u32 length) {
    return Ql_IIC_Write(IIC_CHANNEL, address, data, length);
}

s32 Sf_IIC_Read(u8 address, u8* data, u32 length) {
    return Ql_IIC_Read(IIC_CHANNEL, address, data, length);
}

s32 Sf_IIC_WriteRead(u8 address, u8 *data, s32 dataLength, u8 *buffer, u32 length) {
    return Ql_IIC_Write_Read(IIC_CHANNEL, address, data, dataLength, buffer, length);
}

bool Sf_IIC_Init(void)
Remote measuring and logging - Evaluation of the Quectel OpenCPU technology
Simon Nilsson 2015-06-11

```c
52     {
53         return Sf_ErrorCheck(
54             Ql_IIC_Init(IIC_CHANNEL, PINNAME_RX, PINNAME_DCD,
55             IIC_USE_CONTROLLER),
56             QL_RET_OK,
57             "Failed to initialize IIC!" );
58     }
59
60     bool Sf_IIC_Config(u8 address)
61     {
62         return Sf_ErrorCheck(
63             Ql_IIC_Config(IIC_CHANNEL, TRUE, address, IIC_SPEED),
64             QL_RET_OK,
65             "Failed to configure IIC!" );
66     }
67
68     bool Sf_IIC_Uninit(void)
69     {
70         return Sf_ErrorCheck(
71             Ql_IIC_Uninit(IIC_CHANNEL),
72             QL_RET_OK,
73             "Failed to uninitialize IIC!" );
74     }
75
    sf_sht21.h
1     /*
2     * sf_sht21.h
3     *
4     * Created on: 7 apr 2015
5     *    Author: Simon Nilsson
6     */
7
8     #ifndef SF_SHT21_H_
9     #define SF_SHT21_H_
10
11     #include "ql_type.h"
12     #include "sf_types.h"
13
14    */
15    * Attempts to read data from a SHT21 device.
16    * @param sensors [out] Array to hold sensor values read
17    *    from devices.
18    * @param sensorCount [out] Counter for sensors
19    * @return TRUE if successful.
20    */
21    bool Sf_SHT21_Read(ST_SENSOR sensors[], u8 *sensorCount);
22#endif /* SF_SHT21_H_ */
```
sf_sht21.c

/*
* sf_sht21.c
*
* Created on: 7 apr 2015
* Author: Simon Nilsson
*/

#include "ql_stdlib.h"
#include "ql_error.h"
#include "ql_trace.h"
#include "ql_system.h"
#include "sf_utility.h"
#include "sf_sht21.h"
#include "sf_iic.h"

static const u8 SHT21_ADDR = 0x80;

static const u16 POLYNOMIAL = 0x131;

static bool check_crc(u8 data[], u8 length, u8 checksum)
{
    u8 crc = 0;

    //calculates 8-Bit checksum with given polynomial
    for (u8 i = 0; i < length; i++)
    {
        crc ^= (data[i]);

        for (u8 bit = 8; bit > 0; --bit)
        {
            if (crc & 0x80) crc = (crc << 1) ^ POLYNOMIAL;
            else crc = (crc << 1);
        }
    }

    return crc == checksum;
}

static bool readID(char serial[])
{
    u8 registerAddr[2] = {0xFA, 0x0F};
    u8 data[8];
    u8 buffer[8];

    if(!Sf_ErrorCheck(Sf_IIC_WriteRead(SHT21_ADDR, registerAddr, 2, buffer, 8),
    8,
    "Failed to read register 0xFA 0x0F of SHT21!"
)) return FALSE;
bool valid = TRUE;

data[5] = buffer[0];
valid &= check_crc(buffer, 1, buffer[1]);
data[4] = buffer[2];
valid &= check_crc(buffer + 2, 1, buffer[3]);
data[3] = buffer[4];
valid &= check_crc(buffer + 4, 1, buffer[5]);
data[2] = buffer[6];
valid &= check_crc(buffer + 6, 1, buffer[7]);

registerAddr[0] = 0xFC;
registerAddr[1] = 0xC9;
if (!Sf_ErrorCheck(Sf_IIC_Write(SHT21_ADDR, &cmd, 1), 1, 4)) return FALSE;

if (!valid) {
    Ql_Debug_Trace("SHT21 ID CRC Error.");
    return FALSE;
}

// Convert serial to hex string.
valid = FALSE;
for(int i = 0; i < 8; i++) {
    if(data[i] != 0) valid = TRUE;
    Ql_sprintf(serial + i * 2, "%02X", data[7-i]);
}
return valid;

static bool readTemp(float *temp) {
    u8 cmd = 0xF3;
    u8 buffer[3];

    // Send temperature measure command.
    if (!Sf_ErrorCheck(Sf_IIC_Write(SHT21_ADDR, &cmd, 1), 1, 4)) return FALSE;
    }
"Failed to write temperature command to SHT21!"
)) return FALSE;

ing i = 0;
// Poll device untill data can be read.
do
{
   // Timeout after 200ms.
   if (++i >= 20)
   {
      Ql_Debug_Trace("SHT21 temperature read timed out.\r\n");
      return FALSE;
   }

   // Sleep 10ms.
   Ql_Sleep(10);
} while(Sf_IIC_Read(SHT21_ADDR, buffer, 3) != 3);

// Check CRC
if (!check_crc(buffer, 2, buffer[2]))
{
   Ql_Debug_Trace("SHT21 CRC Error while reading temperature.\r
\n");
   return FALSE;
}

// Convert data bytes to u16 and clear bits [1..0] (status bits)
u16 result = (buffer[0] << 8) | (buffer[1] & 0xFC);

// Calculate temperature [C]
*temp = -46.85 + 175.72/65536 * (float)result;
return TRUE;
}

static bool readHumidity(float *rh)
{
    u8 cmd = 0xF5;
    u8 buffer[3];

    // Send moisture measure command.
    if (!Sf_ErrorCheck(      
        Sf_IIC_Write(SHT21_ADDR, &cmd, 1),
        i,
        "Failed to write moisture command to SHT21!"
    )) return FALSE;
    u8 i = 0;
// Poll device until data can be read.
do
{
    // Timeout after 200ms.
    if (++i >= 20)
    {
        Ql_Debug_Trace("SHT21 moisture read timed out.\r\n");
        return FALSE;
    }
    // Sleep 10ms.
    Ql_Sleep(10);
} while (Sf_IIC_Read(SHT21_ADDR, buffer, 3) != 3);
// Check CRC
if (!check_crc(buffer, 2, buffer[2]))
{
    Ql_Debug_Trace("SHT21 CRC Error while reading moisture.\r\n");
    return FALSE;
}
// Convert data bytes to u16 and clear bits [1..0] (status bits)
u16 result = (buffer[0] << 8) | (buffer[1] & 0xFC);
// Calculate relative humidity [%RH]
*rh = -6.0 + 125.0/65536 * (float)result;
return TRUE;
}
bool Sf_SHT21_Read(ST_SENSOR sensors[], u8 *sensorCount)
{
    if (Sf_IIC_Config(SHT21_ADDR))
    {
        char serial[17] = '\0';
        float value = 0;
        Ql_Debug_Trace("Checking for SHT21 device...\r\n");
        if (readID(serial))
        {
            Ql_Debug_Trace("SHT21 ID: %s\r\n", serial);
            // Add temperature measurement
            if (readTemp(&value))
            {
                Ql_sprintf(sensors[*sensorCount].id, "%s-t", serial);
                sensors[*sensorCount].value = value;
            }
        }
    }
}
**Q1_Debug_Trace**("SHT21 Temp: %f\r\n", sensors[*sensorCount].value);
*sensorCount += 1;
}

// Add relative humidity measurement
if(readHumidity(&value)) {
    Q1_sprintf(sensors[*sensorCount].id, "%s-r", serial);
sensors[*sensorCount].value = value;
Q1_Debug_Trace("SHT21 Moisture: %f\r\n", sensors[*sensorCount].value);
*sensorCount += 1;
}

return TRUE;
}
#else
Q1_Debug_Trace("Failed to configure IIC for SHT21!\r\n");
return FALSE;
}

---

**sf_json.h**

/*
 * sf_json.h
 *
 * Created on: 23 apr 2015
 * Author: Simon Nilsson
 */

#ifndef SF_JSON_H_
#define SF_JSON_H_

#include "ql_type.h"
#include "sf_types.h"

/**
 * Compiles data into JSON.
 * @param json [out] Buffer large enough to hold data.
 * @param imei IMEI-number.
 * @param datastreams Array of datastreams.
 * @param sensorCount Length of datastreams-array in bytes.
 * @return Length of output in bytes.
 */
s32 Sf_JSON_Build(char *json, const char *imei, ST_SENSOR datastreams[], u32 sensorCount);

/**
 * Tries to extract parameters from JSON data.
 * @param data The data.
 */
bool Sf_JSON_Parse(const char *data, s32 *s, s32 *log, char *update);

static char* find(const char *json, const char *name)
{
    char searchString[32];
    // Build search string.
    s32 len = Ql_sprintf(searchString, "\%s\":", name);
    // Find start
    char *start = Ql_strstr(json, searchString);
    if(start)
        return start + len;
    return start;
}

static bool parseInteger(const char *json, const char *name, s32 *value)
{
    *value = 0;
    // Find start
    char *start = find(json, name);
    if(start)
    {
        // Calculate length.
    }
// Find whatever comes first ',', '}' or '}'.
s32 curlyBrace = Qlstrstr(start, '}') - start;
s32 len = Qlstrstr(start, ',') - start;
if (curlyBrace < len) len = curlyBrace;

if (len > 0 && len < 11)
{
    // Extract string.
    char strTemp[11];
    Qlstrncpy(strTemp, start, len);
    strTemp[len] = '\0';

    // Parse value.
    *value = Qlatoi(strTemp);

    return TRUE;
}
return FALSE;

static bool parseString(const char *json, const char *name, char *buffer)
{
    buffer[0] = '\0';

    // Find start
    char *start = find(json, name);
    if (start)
    {
        // step over ".
        start++;

        // Calculate length.
        // Find ending ".
        char *end = Qlstrstr(start, "\n");

        if (end > start)
        {
            // Extract string.
            Qlstrncpy(buffer, start, end - start);
            buffer[end - start] = '\0';

            return TRUE;
        }
    }
    return FALSE;
}

s32 Sf_JSON_Build(char *json, const char *imei, ST_SENSOR datastreams[], u32 sensorCount)
// Keep track of length.
s32 len = 0;

// Start of data.
len += Ql_sprintf(
    // Buffer
    json,
    // Format
    "{\"version\":\"%s\",\"IMEI\":\"%s\",\"datastreams\":[",
    // API Version.
    SF_VERSION,
    // IMEI
    imei
    );

// Sensor data.
for (u32 i = 0; i < sensorCount; i++)
{
    len += Ql_sprintf(
        // Buffer
        json + len,
        // Format
        "{\"id\":\"%s\",\"val\":\"%f\"}%s",
        // Id
        datastreams[i].id,
        // Value
        datastreams[i].value,
        // Add comma if not last item.
        (i < sensorCount - 1) ? ",," : ""
    );
}

// End of data.
len += Ql_sprintf(json + len, "}");

// Return data length.
return len;
}

bool Sf_JSON_Parse(const char *data, s32 *s, s32 *log, char *update)
null
Remote measuring and logging - Evaluation of the Quectel OpenCPU technology
Simon Nilsson 2015-06-11

bool Sf_GetIP(const char* hostname, u32 *ip);

bool Sf_TCP_Connect(u32 ip, u16 port, void (*read)(s32), void (*write)(s32));

bool Sf_TCP_Disconnect(void);

s32 Sf_TCP_Send(u8 *data, s32 length);

s32 Sf_TCP_Recv(u8 *buffer, s32 length);

sf_tcp.c

/*
 * sf_tcp.c
 */

* Created on: 7 apr 2015
* Author: Simon Nilsson
*/

#include "custom_feature_def.h"
#include "ril.h"
Remote measuring and logging - Evaluation of the Quectel OpenCPU technology
Simon Nilsson

```c
#include "ql_type.h"
#include "ql_gprs.h"
#include "ql_socket.h"
#include "ql_trace.h"
#include "ql_error.h"
#include "ql_stdlib.h"
#include "ql_system.h"
#include "sf_config.h"
#include "sf_tcp.h"

static ST_GprsConfig config = {
    // APN
    SF_APN,
    // Username
    "",
    // Password
    "",
    // Authentication type
    0,
    // Reserved
    NULL,
    NULL,
};

/**
 * Not used
 */
static ST_PDPContxt_Callback callback_gprs_func = {
    NULL,
    NULL
};

/**
 * Callbacks
 */
static void (*callback_read)(s32 error);
static void (*callback_write)(s32 error);

static void tcp_read(s32 socketId, s32 errCode, void* customParam)
{
    callback_read(errCode);
}

static void tcp_write(s32 socketId, s32 errCode, void* customParam)
{
```
}
{
callback_write(errCode);
}

static ST_SOC_Callback callback_soc_func = {
    NULL,
    NULL,
    tcp_read,
    tcp_write
};

/**
 * PDP Context Id
 */
static s32 pdpCtxId;

/**
 * Socket Id
 */
static s32 socketId;

bool Sf_ActivateGPRS(void)
{
    s32 ret;

    // Get PDP Context.
    ret = Ql_GPRS_GetPDPContextId();
    if (ret == GPRS_PDP_ERROR)
    {
        Ql_Debug_Trace("No PDP context available!\n\n");
        return FALSE;
    }
    else
    {
        pdpCtxId = ret;
    }

    // Register callbacks
    ret = Ql_GPRS_Register(pdpCtxId, &callback_gprs_func, NULL);
    if (ret != GPRS_PDP_SUCCESS)
    {
        Ql_Debug_Trace("Failed to register GPRS callbacks! [%d]\n\n", ret);
        return FALSE;
    }

    // Configure PDP.
    ret = Ql_GPRS_Config(pdpCtxId, &config);
    if (ret != GPRS_PDP_SUCCESS)
Ql_Debug_Trace("Failed to configure GPRS PDP! [%d]\r\n", ret);
    return FALSE;
}

// Activate GPRS.
ret = Ql_GPRS_ActivateEx(pdpCtxId, TRUE);
if(ret != GPRS_PDP_SUCCESS)
{
    Ql_Debug_Trace("Failed to activate GPRS! [%d]\r\n", ret);
    return FALSE;
}

return TRUE;

bool Sf_DeactivateGPRS(void)
{
    // Deactivate GPRS.
s32 ret = Ql_GPRS_DeactivateEx(pdpCtxId, TRUE);
    if(ret != GPRS_PDP_SUCCESS)
    {
        Ql_Debug_Trace("Failed to deactivate GPRS! [%d]\r\n", ret);
        return FALSE;
    }

    return TRUE;
}

bool Sf_GetIP(const char* hostname, u32 *ip)
{
    s32 ret = 0;
    u32 ipCount = 0;
    u32 ipAddress[5];

    // Get server IP.
    ret = Ql_IpHelper_GetIPByHostNameEx(pdpCtxId, 0, (u8*)hostname, ipAddress, &ipCount);
    if(ret != SOC_SUCCESS || ipCount < 1)
    {
        Ql_Debug_Trace("Failed to get IP for hostname \"%s\"! [%d]\r
\n", hostname, ret);
        return FALSE;
    }

    *ip = ipAddress[0];
    return TRUE;
}

bool Sf_TCP_Connect(u32 ip, u16 port, void (*read)(s32), void (*write)(s32))
{
s32 ret;
callback_read = read;
callback_write = write;

// Register socket callbacks.
ret = Ql_SOC_Register(callback_soc_func, NULL);
if (ret != SOC_SUCCESS)
{
    Ql_Debug_Trace("Failed to register socket callbacks! [%d]\n", ret);
    return FALSE;
}

// Create socket.
ret = Ql_SOC_Create(pdpCtxId, SOC_TYPE_TCP);
if (ret < 0)
{
    Ql_Debug_Trace("Failed to create socket! [%d]\r\n", ret);
    return FALSE;
}
else
    socketId = ret;

// NOTE: ConnectEx takes pointer to 4 byte IP-address casted to u32.
// It's strange but it works.
ret = Ql_SOC_ConnectEx(socketId, (u32)&ip, port, TRUE);
if (ret != SOC_SUCCESS)
{
    Ql_Debug_Trace("Failed to connect! [%d]\r\n", ret);
    return FALSE;
}
return TRUE;

bool Sf_TCP_Disconnect(void)
{
    // Close socket.
s32 ret = Ql_SOC_Close(socketId);
    if (ret != SOC_SUCCESS)
    {
        Ql_Debug_Trace("Failed to close socket! [%d]\r\n", ret);
        return FALSE;
    }
    return TRUE;
s32 Sf_TCP_Send(u8 *data, s32 length) {
    return Ql_SOC_Send(socketId, data, length);
}

s32 Sf_TCP_Recv(u8 *buffer, s32 length) {
    return Ql_SOC_Recv(socketId, buffer, length);
}

---

sf_http.h

/*
 * sf_http.h
 */

#ifndef SF_HTTP_H_
#define SF_HTTP_H_

#include "ql_type.h"

/**
 * HTTP Port
 */

const u16 HTTP_PORT;

/**
 * Sends a HTTP request to preconfigured server.
 * @param url       Url.
 * @param type      Request type.
 * @param dataLength Length of data.
 * @param callback  Callback function for writing data.
 * @return TRUE if successful.
 */
bool Sf_HTTP_SendRequest(const char *url, const char *type, s32 dataLength, void (*callabck)(s32 error));

/**
 * Sends a HTTP request to preconfigured server.
 * @param callback Callback function for reading response.
 */
void Sf_HTTP_GetResponse(void (*callback)(s32 error));

#endif /* SF_HTTP_H_ */

sf_http.c

/*
 * sf_http.c
```c
#include "ql_stdlib.h"
#include "ql_trace.h"
#include "ql_error.h"
#include "ql_socket.h"
#include "sf_config.h"
#include "sf_http.h"
#include "sf_tcp.h"

const u16 HTTP_PORT = 80;

static char header[128];
static s32 headerLength;
static s32 sent;
static u8 receivedHeaderEnd;

/**
* Callbacks
*/
static void (*callback_read)(s32 error);
static void (*callback_write)(s32 error);

static void http_write(s32 error)
{
    if(error)
    {
        callback_write(error);
        return;
    }

    // Send header
    while(sent < headerLength)
    {
        s32 ret = Sf_TCP_Send((u8*)header + sent, headerLength - sent);
        // Blocking
        if(ret == SOC_WOULDBLOCK)
            return;
        // Error
        else if(ret < 0)
        {
            callback_write(ret);
            return;
        }
```
sent += ret;
}

// Send data
callback_write(error);
}

static void http_read(s32 error)
{
    if(error)
    {
        callback_read(error);
        return;
    }
    // Read header
    while(receivedHeaderEnd != 4)
    {
        u8 c;
        s32 ret = Sf_TCP_Recv(&c, 1);
        // Blocking
        if(ret == SOC_WOULDBLOCK)
            return;
        // Error
        else if(ret < 0)
        {
            callback_read(ret);
            return;
        }
        // Check for end (\r\n\r\n)
        if(ret == 1)
        {
            if(c == '\r' || c == '\n')
                receivedHeaderEnd++;
            else
                receivedHeaderEnd = 0;
        }
        // Receive data
        callback_read(error);
    }

void makeHeader(const char *host, const char *path, const char *type, s32 contentLength)
{
    headerLength = Ql_sprintf(
Remote measuring and logging - Evaluation of the Quectel OpenCPU technology
Simon Nilsson 2015-06-11

```
// Buffer
header,

// Format
"%s %s HTTP/1.1\r\n"
"Host: %s\r\n"
"Content-Length: %d\r\n"
"Connection: Close\r\n"
"\r\n",

// Request type.
type,

// Path
path,

// Host
host,

// Content-Length
contentLength
);

bool Sf_HTTP_SendRequest(const char *url, const char *type, s32 dataLength, void (*callback)(s32))
{
    callback_write = callback;
    char host[100];
    char path[100];
    sent = 0;

    if(Ql_sscanf(url, "%99[\/%99[\n]", host, path) == 2)
    {
        // Prepare header
        makeHeader(host, path, type, dataLength);

        // Get IP
        u32 ip;
        if(Sf_GetIP(host, &ip))
            {
                u8 *ip8 = (u8*)&ip;
                Ql_Debug_Trace("Remote Address: %d.%d.%d.%d\r\n", 
ip8[0], ip8[1], ip8[2], ip8[3], HTTP_PORT 
    );

        // Connect
        if(Sf_TCP_Connect(ip, HTTP_PORT, http_read, http_write))
            {
```

65
Ql_Debug_Trace("<- Connection successful -->\r\n");
http_write(0);
return TRUE;
}
else Ql_Debug_Trace("Unable to get IP for "\"%s\"\r\n", host);
}
else Ql_Debug_Trace("Unable to parse URL "\"%s\"\r\n", url);
return FALSE;

/**
 * Sends a HTTP request to preconfigured server.
 * @param callback Callback function for reading response.
 */
void Sf_HTTP_GetResponse(void (*callback)(s32 error))
{
    callback_read = callback;
    receivedHeaderEnd = 0;
    http_read(0);