OpenCPU Quick Start Application Note

GPS/GPRS Module Series

Rev. OpenCPU_Quick_Start_Application_Note_V1.1

Date: 2017-07-20

APPLICATIVE PRODUCT

MODULE TYPE
M85
M66

About the Document

History

Revision	Date	Author	Description
1.0	2017-07-01	Chunmao Li	Initial
1.1	2017-07-10	Chunmao Li	1. Added M66 and M85 R2.0 as supported modules, and modified the description about download tool. 2. Added the description about how to work with Eclipse.

Contents

Table Index

Figure Index

1 Introduction

This document describes how to quickly start to program with OpenCPU Software Development Kit. Besides, this document tells the dos and don’ts for reference to design and program the application.

2 OpenCPU Documentation

Table 1: OpenCPU Documentation

Document	Description
ZF_ OpenCPU_ Quick_ Start _ Application_ Note	Description of the first step to use OpenCPU SDK to develop application.
ZF_ OpenCPU_User_Guide	Complete description of OpenCPU platform. This document includes the description of the OpenCPU user APIs.
ZF_ OpenCPU_RIL_Application_Note	This document describes how to program AT command with RIL API, and get the response of AT command when the API returns.
ZF_ OpenCPU_FOTA_Application_Note	This document describes how to program FOTA in application.
ZF_ OpenCPU_Security_Data_Application_Note	This document introduces OpenCPU Security Data Solution, and shows how to program the critical data in OpenCPU platform.
ZF_OpenCPU_Watchdog_Application_Note	This document introduces OpenCPU watchdog solution.
ZF_ OpenCPU_GCC_Installation_Guide	Guide for the installation steps of GCC compiler.
ZF_ OpenCPU_GCC_Eclipse_User_Guide	This document tells how to set up the Eclipse IDE development environment.
Firmware_Upgrade_Tool_Lite_GS2_User_Guide	This document describes how to download firmware using the download tool, which is applicable to M85 R1.0 OpenCPU module.
ZF_QFlash_User_Guide	This document describes how to download firmware using the download tool, which is applicable to M66 and M85 R2.0 OpenCPU module.

3 Necessaries

To work with OpenCPU, you need to confirm if you have had the software and hardware components listed below.

3.1. Host System

The following host operating systems and architectures are supported.

- Microsoft Windows XP (SP1 or later)
- Windows Vista
- Windows 7 systems using IA32, AMD64, and Intel 64 processors

3.2. Compiler and IDE

- GCC Compiler (Sorcery CodeBench Lite for ARM EABI). Please refer to the document [1].
- Command-line complication. Please refer to the document [2].
- IDE: Eclipse (Optional). Please refer to the document [4].

3.3. Programming Language

Basic C-language programming knowledge is a must.

3.4. Module Hardware

- ZFI GSM/GPRS Module with OpenCPU features
- ZF EVB
- Other accessories (power adapter, COM cable)

If you need these parts, please contact Quectel technical support.

3.5. OpenCPU SDK

- OpenCPU SDK software package
- Firmware Download Tool (included in SDK).

4 Set up Development Environment

ZF OpenCPU supports two ways to develop and compile application program: command-line and Eclipse IDE. Two different software development kit packages are issued for the different compiling tools. For example, the “OpenCPU_GS3_SDK_V1.0” package is applicable to command line, and “OpenCPU_GS3_SDK_V1.0_**Eclipse**” (_Eclipse suffix is appended) is applicable to Eclipse.

Whatever compiling way it is, the compiler is GCC (Sourcery CodeBench Lite for ARM). Please refer to the Chapter 2 in document [1] to install GCC compiler and verify the validity before you choose a compiling way.

4.1. Command Line

Since the compiling commands are very simple in OpenCPU (only make clean/new are used), so this compiling way is the default compiling way. And developer may mange the application codes with some code management tool, such as Source Insight.

Developer just needs to refer to the following two steps to set up the development environment.

- **Install Sourcery CodeBench Lite on your host computer.** Please refer to the Chapter 2 in document [1] to install GCC compiler and verify the validity.
- **Set up the environment after installation.** Please refer to the Chapter 3 in document [1] to configure the environment.

4.2. Work with IDE (Eclipse)

Please refer to the document [4] for how to work with Eclipse.

5 Compilation

This section only introduces how to compile the program in command line. Please refer to document [4] for how to compile with Eclipse.

5.1. Compiling

In OpenCPU, compiling commands are executed in command line. The clean and compiling commands are defined as below.

```
make clean
make new
```

5.2. Compiling Output

In command-line, some compiler processing information will be output during compiling. All WARNINGs and ERRORs are recorded in “\SDK\build\gcc\build.log”.

If there’s any compiling error during compiling, please check the “build.log” for the error line number and the error hints.

6 Download

6.1. For TE-A

If you are using TE-A module and EVB, please download firmware through the MAIN port on EVB.

6.2. For Raw Module

If the module has been pasted on your own board, please download firmware through the UART port 1.

6.3. For Mass Production

In order to improve the production efficiency, ZF provides the special fixture and download tool which can download firmware to several pieces of modules at one time. Mass production customers can consult ZF technical support for that if needed.

6.4. Download

The document [5] (for M85 R1.0) and [6] (for M85 R2.0 and M66) introduces the download tool and how to use it to download application bin.

7 Debugging

In OpenCPU, printing trace messages through serial port is the main debugging method.

OpenCPU module provides three serial ports, MAIN UART (UART PORT 1), DEBUG UART (UART PORT 2) and AUX UART (UART PORT 3).

In program, you can call Ql_UART_Open() to open any UART port, and call Ql_UART_Write() to output debugging messages. Especially, if the DEBUG port is used as debugging port, the API function Ql_Debug_Trace() is available to output messages through DEBUG port. The Ql_Debug_Trace function can format and print a series of characters and values through DEBUG port.

8 About OpenCPU SDK

You can decompress the SDK package to get the directory structure of OpenCPU SDK. The typical directory hierarchy is as below.

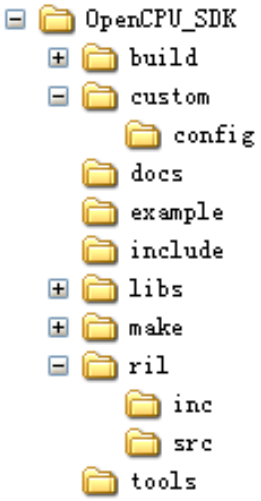

Figure 1: Directory Hierarchy

Table 2: Directory Description

Directory	Description
OpenCPU_SDK	The root directory of OpenCPU SDK.
build	All compiling results are output to this directory.
custom	This directory is designed as the root directory of your project. In the subdirectory “custom\config\”, you can reconfigure the application according to requirements, such as heap memory size, multitasks and the stack size of tasks, GPIO initial status. All configuration files for you are named with prerfix “custom_”.
docs	Store all OpenCPU related documents.
example	All example codes are here. Each example file implements an application of independent function. And each example file can be compiled to an executable image bin.

include	All APIs head files are stored here.
libs	Dependent libraries for compiling.
make	All compiling scripts and makefile are placed here.
ril	Place the open source codes of OpenCPU RIL. You can also easily add a new API to implement a standard AT command based on the open source of RIL.
tools	Some tools for application development, such as download tool and packaging tool for FOTA.

Please refer to the Section 2.4 in document [2] for more details about SDK development environment, such as compiling method and download.

Please refer to Chapter 4 in document [2] for system configuration of application, such as heap memory size, tasks, stack size of tasks and GPIO initial status.

9 Create Your Project

By default, the directory SDK\custom\ is designed as the root directory of your project. In this directory, a program file “main.c” is placed, which demonstrates the primary program framework of OpenCPU program.

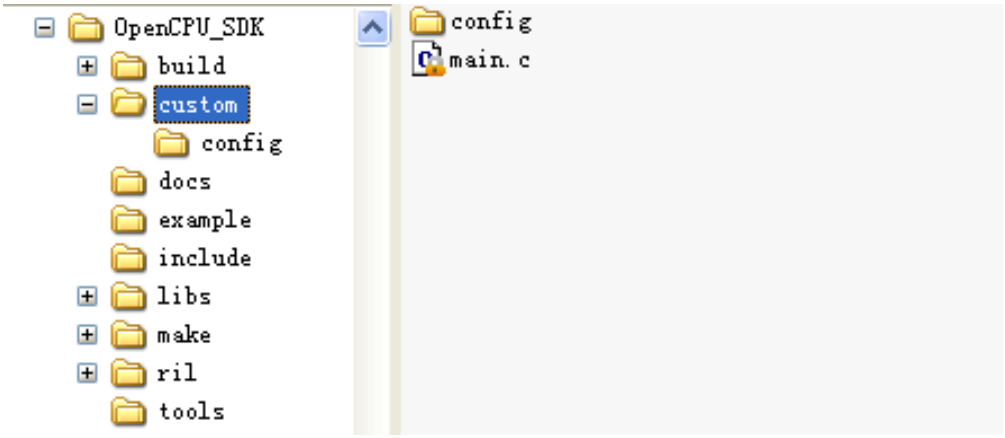

Figure 2: Custom Directory

In the directory SDK\custom\, you can add other module files and subdirectories. Please refer to the Section 2.4.5 in document [2] for extended information.

All source code files are managed by the makefile \SDK\make\gcc\gcc_makefile. You can decide which directories, and which source code files need to be compiled in this makefile. Please refer to the Section 2.4.5 in document [2] for extended information.

Up to now, you may notice that there is a default project. What you need to do is to add codes to main.c, or change the existing codes in main.c. Besides, you can add other .c files. And all newly-added .c files in SDK\custom\ will be compiled automatically.

10 Quick Start with Program

In this section, two example applications are given to guide how to start to program over OpenCPU SDK. The first example application implements an LED blinking by periodically pulling down/up a GPIO. The other example application demonstrates how to program GPRS, and send a package of data to server.

All example codes will cover SDK\custom\main.c. Or you can delete the main.c and create a new .c file.

10.1. How to Program GPIO

- **Include Head Files**

To know which head files are needed, you should understand the requirements of this application. For this example application, the requirement is: implementing an LED blinking by periodically pulling down/up a GPIO.

First of all, you need to control a GPIO, the APIs and related definitions are in ql_gpio.h.

Secondly, “periodically” means a timer is needed. The related definitions are in ql_timer.h.

Finally, any application has a message loop procedure, so the ql_system.h is must. Besides, you need to print some log information to debug the program, the related head files are ql_stdlib.h and ql_trace.h. All return values for API functions are defined in ql_error.h.

Conclusions as a result, the head files that you need to include are as follows:

```
#include "ql_stdlib.h"
#include "ql_trace.h"
#include "ql_error.h"
#include "ql_system.h"
#include "ql_gpio.h"
#include "ql_timer.h"
```

- **Program GPIO**

On ZF EVB, the NETLIGHT pin and STATUS pin have been respectively connected to an LED, which means you can control NETLIGHT pin or STATUS pin to implement LED blinking.

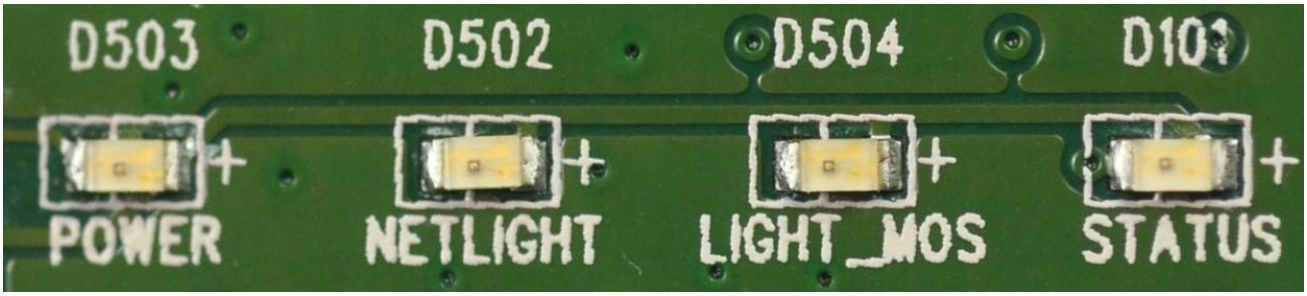

Figure 3: EVB LED Indicator

Here, the program chooses NETLIGHT pin as GPIO pin.

```
// Define GPIO pin
static Enum_PinName m_gpioPin = PINNAME_NETLIGHT;
```

At the beginning of this application, the GPIO is initialized like this:

- Directory – output
- Initial level – low
- Pull state – pull enable and pull up

```
// Initialize GPIO
ret = Ql_GPIO_Init(m_gpioPin, PINDIRECTION_OUT, PINLEVEL_LOW, PINPULLSEL_PULLUP);
if (QL_RET_OK == ret)
{
    Ql_Debug_Trace("<-- Initialize GPIO successfully -->\r\n");
}else{
    Ql_Debug_Trace("<-- Fail to initialize GPIO pin, cause=%d -->\r\n", ret);
}
```

Next, you need to start a timer, and pull up/down the GPIO periodically to implement LED blinking.

- **Timer and LED Blinking**

In this case, the program defines a timer with the timeout of 500ms. It means the LED will light for 500ms and becomes dark for 500ms.

Firstly, define a timer and the timer interrupt handler.

```
// Define a timer and the handler
static u32 m_myTimerId = 2014;
static u32 m_nInterval = 500;     // 500ms
static void Callback_OnTimer(u32 timerId, void* param);
```

Secondly, register and start the timer.

```
// Register and start timer
Ql_Timer_Register(m_myTimerId, Callback_OnTimer, NULL);
Ql_Timer_Start(m_myTimerId, m_nInterval, TRUE);
```

Finally, implement the timer interrupt handler.

```
static void Callback_OnTimer(u32 timerId, void* param)
{
    s32 gpioLvl = Ql_GPIO_GetLevel(m_gpioPin);
    if (PINLEVEL_LOW == gpioLvl)
    {
        // Set GPIO to high level, then LED is light
        Ql_GPIO_SetLevel(m_gpioPin, PINLEVEL_HIGH);
        Ql_Debug_Trace("<-- Set GPIO to high level -->\r\n");
    }else{
        // Set GPIO to low level, then LED is dark
        Ql_GPIO_SetLevel(m_gpioPin, PINLEVEL_LOW);
        Ql_Debug_Trace("<-- Set GPIO to low level -->\r\n");
    }
}
```

Now, all programming work are finished. The complete codes are as below.

```
#include "ql_stdlib.h"
#include "ql_trace.h"
#include "ql_error.h"
#include "ql_system.h"
#include "ql_gpio.h"
#include "ql_timer.h"

// Define GPIO pin
static Enum_PinName m_gpioPin = PINNAME_NETLIGHT;

// Define a timer and the handler
static u32 m_myTimerId = 2014;
static u32 m_nInterval = 500;     // 500ms
static void Callback_OnTimer(u32 timerId, void* param);

/*****************************************************************/
/* The entrance procedure for this example application                */
/*****************************************************************/
void proc_main_task(s32 taskId)
```

```
{
    s32 ret;
    ST_MSG msg;

    Ql_Debug_Trace("OpenCPU: LED Blinking by NETLIGH\r\n");

    // Initialize GPIO
    ret = Ql_GPIO_Init(m_gpioPin, PINDIRECTION_OUT, PINLEVEL_LOW,
PINPULLSEL_PULLUP);
    if (QL_RET_OK == ret)
    {
        Ql_Debug_Trace("<-- Initialize GPIO successfully -->\r\n");
    }else{
        Ql_Debug_Trace("<-- Fail to initialize GPIO pin, cause=%d -->\r\n", ret);
    }

    // Register and start timer
    Ql_Timer_Register(m_myTimerId, Callback_OnTimer, NULL);
    Ql_Timer_Start(m_myTimerId, m_nInterval, TRUE);

    // START MESSAGE LOOP OF THIS TASK
    while(TRUE)
    {
        Ql_OS_GetMessage(&msg);
        switch(msg.message)
        {
        default:
            break;
        }
    }
}

static void Callback_OnTimer(u32 timerId, void* param)
{
    s32 gpioLvl = Ql_GPIO_GetLevel(m_gpioPin);
    if (PINLEVEL_LOW == gpioLvl)
    {
        // Set GPIO to high level, then LED is light
        Ql_GPIO_SetLevel(m_gpioPin, PINLEVEL_HIGH);
        Ql_Debug_Trace("<-- Set GPIO to high level -->\r\n");
    }else{
        // Set GPIO to low level, then LED is dark
        Ql_GPIO_SetLevel(m_gpioPin, PINLEVEL_LOW);
        Ql_Debug_Trace("<-- Set GPIO to low level -->\r\n");
```

```
    }
}
```

- **Run This Application**

You can copy the complete codes to SDK\custom\main.c to cover the existing codes, and compile and download the app bin to module. Please refer to the Section 2.4.3 and 2.4.4 in document [2] for compiling and download information.

When this application is running, you can watch the “D502” LED on EVB blinking at the period of 500ms. Meanwhile, you can watch the following output from DEBUG port.

```
11/6/2013 16:54:05.666 [RX] - OpenCPU_GS2_SDK_V5.2<CR><LF>
OpenCPU: LED Blinking by NETLIGH<CR><LF>
<-- Initialize GPIO successfully --><CR><LF>
<-- Set GPIO to high level --><CR><LF>
<-- Set GPIO to low level --><CR><LF>
<-- Set GPIO to high level --><CR><LF>
<-- Set GPIO to low level --><CR><LF>
<-- Set GPIO to high level --><CR><LF>
<-- Set GPIO to low level --><CR><LF>
<-- Set GPIO to high level --><CR><LF>
```

10.2. How to Program GPRS

- **Include Head Files**

To know which head files are needed, you should understand the requirements of this application.

First of all, you need to use OpenCPU RIL feature, so custom_feature_def.h and ril.h are required.

Secondly, GPRS-related API functions are defined in ql_gprs.h and ql_socket.h.

Finally, any application has a message loop procedure, so the ql_system.h is must. Besides, you need to print some log information to debug the program, the related head files are ql_stdlib.h and ql_trace.h. All return values for API are defined in ql_error.h.

Conclusions as a result, the head files that you need to include are as follows:

```
#include "custom_feature_def.h"
#include "ril.h"
#include "ql_stdlib.h"
#include "ql_trace.h"
#include "ql_error.h"
#include "ql_system.h"
```

```
#include "ql_gprs.h"
#include "ql_socket.h"
```

- **Define PDP Context and GPRS Configurations**

```
#define PDP_CONTEXT_ID    0
static ST_GprsConfig m_GprsConfig = {
    "CMNET",      // APN name
    "",           // User name for APN
    "",           // Password for APN
    0,
    NULL,
    NULL,
};
```

Here, the APN is set to China Mobile's APN name – "CMNET" and account. You can change APN name and account accordingly.

- **Define Server IP and Socket Port**

```
static u8    m_SrvADDR[20] = "116.247.104.27\0";
static u32 m_SrvPort = 6003;
```

Here, the same codes define a public server and socket port of Quectel. You can use your own server.

- **Define Receive Buffer**

When the socket connection is established, the program needs a buffer to accept data from socket.

```
#define    SOC_RECV_BUFFER_LEN    1460
static u8    m_SocketRcvBuf[SOC_RECV_BUFFER_LEN];
```

- **Declare Callback for GPRS and Socket**

```
static void Callback_GPRS_Deactived(u8 contextId, s32 errCode, void* customParam);
static void Callback_Socket_Close(s32 socketId, s32 errCode, void* customParam );
static void Callback_Socket_Read(s32 socketId, s32 errCode, void* customParam );
static void Callback_Socket_Write(s32 socketId, s32 errCode, void* customParam );
```

Callback_GPRS_Deactivated: when GPRS network drops down, this callback will be invoked.
Callback_Socket_Close: when socket connection is disconnected, this callback will be invoked.
Callback_Socket_Read: when socket data comes, this callback will be invoked.
Callback_Socket_Write: when call Ql_SOC_Write to send data to socket, but socket is busy. Later, this callback will be invoked to inform App that socket is available.

- **Program OpenCPU RIL**

Since you need to use OpenCPU RIL feature, application needs to call Ql_RIL_Initialize() to initialize RIL-related functions when the main task receives the message *MSG_ID_RIL_READY*.

```
// START MESSAGE LOOP OF THIS TASK
while(TRUE)
{
    Ql_OS_GetMessage(&msg);
    switch(msg.message)
    {
    case MSG_ID_RIL_READY:
        Ql_Debug_Trace("<-- RIL is ready -->\r\n");
        Ql_RIL_Initialize();
        break;
```

- **Program System URC Messages**

Before accessing GPRS network, you have to wait till the module has registered to GPRS network. When the module registers to GPRS network, application will receive the URC message URC_GPRS_NW_STATE_IND. Before this, application program receives some other URC messages which indicates the initializing status of module during booting, such as CFUN status, SIM card status and GSM network status changing. You can properly program these URC messages after receiving them.

The following codes are the complete system messages and URC messages programming.

```
/*****************************************************************/
/* The entrance procedure for this example application           */
/*****************************************************************/
void proc_main_task(s32 taskId)
{
    ST_MSG msg;

    Ql_Debug_Trace("OpenCPU: Simple GPRS-TCP Example\r\n");

    // START MESSAGE LOOP OF THIS TASK
    while(TRUE)
    {
        Ql_OS_GetMessage(&msg);
        switch(msg.message)
        {
        case MSG_ID_RIL_READY:
            Ql_Debug_Trace("<-- RIL is ready -->\r\n");
            Ql_RIL_Initialize();
```

```
            break;
        case MSG_ID_URC_INDICATION:
            // Ql_Debug_Trace("<-- Received URC: type: %d, -->\r\n", msg.param1);
            switch (msg.param1)
            {
            case URC_SYS_INIT_STATE_IND:
                Ql_Debug_Trace("<-- Sys Init Status %d -->\r\n", msg.param2);
                break;
            case URC_SIM_CARD_STATE_IND:
                Ql_Debug_Trace("<-- SIM Card Status:%d -->\r\n", msg.param2);
                break;
            case URC_GSM_NW_STATE_IND:
                Ql_Debug_Trace("<-- GSM Network Status:%d -->\r\n", msg.param2);
                break;
            case URC_GPRS_NW_STATE_IND:
                Ql_Debug_Trace("<-- GPRS Network Status:%d -->\r\n", msg.param2);
                if (NW_STAT_REGISTERED == msg.param2)
                {
                    GPRS_Program();
                }
                break;
            case URC_CFUN_STATE_IND:
                Ql_Debug_Trace("<-- CFUN Status:%d -->\r\n", msg.param2);
                break;
            default:
                Ql_Debug_Trace("<-- Other URC: type=%d\r\n", msg.param1);
                break;
            }
            break;
        default:
            break;
        }
    }
}
```

- **Program GPRS**

After module registers to GPRS network, you can start to program GPRS. GPRS programming mostly contains the several steps below. You can also refer to the Section 5.8 in document [2] for the detailed information of GPRS related APIs and usage.

First of all, register GPRS-related callback functions.

```
ST_PDPContxt_Callback callback_gprs_func = {
    NULL,
    Callback_GPRS_Deactived
};
ST_SOC_Callback callback_soc_func = {
    NULL,
    Callback_Socket_Close,
    // callback_socket_accept,
    NULL,
    Callback_Socket_Read,
    Callback_Socket_Write
};

// Register GPRS callback
ret = Ql_GPRS_Register(PDP_CONTEXT_ID, &callback_gprs_func, NULL);
if (GPRS_PDP_SUCCESS == ret)
{
    Ql_Debug_Trace("<-- Register GPRS callback function -->\r\n");
}else{
    Ql_Debug_Trace("<-- Fail to register GPRS, ret=%d. -->\r\n", ret);
    return;
}
```

Secondly, configure PDP context.

```
ret = Ql_GPRS_Config(PDP_CONTEXT_ID, &m_GprsConfig);
if (GPRS_PDP_SUCCESS == ret)
{
    Ql_Debug_Trace("<-- Configure GPRS PDP -->\r\n");
}else{
    Ql_Debug_Trace("<-- Fail to configure GPRS PDP, ret=%d. -->\r\n", ret);
    return;
}
```

Thirdly, activate PDP.

```
ret = Ql_GPRS_ActivateEx(PDP_CONTEXT_ID, TRUE);
if (ret == GPRS_PDP_SUCCESS)
{
    m_GprsActState = 1;
    Ql_Debug_Trace("<-- Activate GPRS successfully. -->\r\n\r\n");
}else{
    Ql_Debug_Trace("<-- Fail to activate GPRS, ret=%d. -->\r\n\r\n", ret);
    return;
}
```

Finally, deactivate PDP (if not needed).

```
ret = Ql_GPRS_DeactivateEx(PDP_CONTEXT_ID, TRUE);
Ql_Debug_Trace("<-- Deactivate GPRS, ret=%d -->\r\n\r\n", ret);
```

- **Program Socket**

After GPRS PDP is activated, you can start to program TCP/UDP socket. Socket programming mostly contains the several steps below. Please refer to the Section 5.9 in document [2] for the detailed information of GPRS related APIs and usage.

First, register socket related callback functions.

```
ret = Ql_SOC_Register(callback_soc_func, NULL);
if (SOC_SUCCESS == ret)
{
    Ql_Debug_Trace("<-- Register socket callback function -->\r\n");
}else{
    Ql_Debug_Trace("<-- Fail to register socket callback, ret=%d. -->\r\n", ret);
    return;
}
```

Secondly, create socket.

```
m_SocketId = Ql_SOC_Create(PDP_CONTEXT_ID, SOC_TYPE_TCP);
if (m_SocketId >= 0)
{
    Ql_Debug_Trace("<-- Create socket successfully, socket id=%d. -->\r\n", m_SocketId);
}else{
    Ql_Debug_Trace("<-- Fail to create socket, ret=%d. -->\r\n", m_SocketId);
    return;
}
```

Thirdly, connect to socket server.

```
ret = Ql_SOC_ConnectEx(m_SocketId,(u32)m_ipAddress, m_SrvPort, TRUE);
if (SOC_SUCCESS == ret)
{
    m_SocketConnState = 1;
    Ql_Debug_Trace("<-- Connect to server successfully -->\r\n");
}else{
    Ql_Debug_Trace("<-- Fail to connect to server, ret=%d -->\r\n", ret);
    Ql_Debug_Trace("<-- Close socket.-->\r\n");
    Ql_SOC_Close(m_SocketId);
    m_SocketId = -1;
    return;
}
```

After the socket is connected with server, you can send data to server or receive data from server.

Fourthly, send socket data.

```
char pchData[200];
s32   dataLen = 0;
u64   ackNum = 0;
Ql_memset(pchData, 0x0, sizeof(pchData));
dataLen += Ql_sprintf(pchData + dataLen, "%s", "A B C D E F G");
ret = Ql_SOC_Send(m_SocketId, (u8*)pchData, dataLen);
if (ret == dataLen)
{
    Ql_Debug_Trace("<-- Send socket data successfully. --> \r\n");
}else{
    Ql_Debug_Trace("<-- Fail to send socket data. --> \r\n");
}
```

Here, the codes demonstrate sending data “A B C D E F G” to server.

After sending data, you can call Ql_SOC_GetAckNumber() to check if the server has received the data.

Besides, you can call Ql_SOC_Close() to close socket connection, and call Ql_GPRS_DeactivateEx() to deactivate GPRS PDP.

You can refer to the example_tcp_demo.c in SDK for the complete codes. And you can compile and run this example. The usage for this example is included in the example.

11 Reference Design and Programming Notes

11.1. External Watchdog

In actual product, if no external MCU, please add external watchdog chip to prevent application from exception. When the external watchdog overflows, please reset the VBAT pin of module to power off/on the module, so that the module can reset status thoroughly.

In program, you can specify the GPIO pin to feed the extern watchdog in \SDK\custom\custom_sys_cfg.c.

For multitask application, OpenCPU has designed the special watchdog solution that can monitor all tasks with an external watchdog chip. Please refer to the document [7] for extended information.

11.2. Resetting Module Solution

In order to keep a stable running state, it is recommended to add resetting module mechanism. For example, reset the module when the module is idle or has low load. The 24-hour of resetting period is recommended.

When network trouble happens to module, such as GSM/GPRS registration failure, you can use this resetting solution to restore it.

There are software method (API function) and hardware method to reset the module (refer to the previous section - external watchdog). It is recommended to use hardware method, so that the module can reset thoroughly.

11.3. Critical User Data Protection

Please adopt OpenCPU Security Data Solution to store the critical configurations for the application, such as APN, server IP address and socket port number. The safe storing region can contain 1700 bytes data.

The data is directly accessed on raw flash. For the sake of flash life, please control the frequency of writing. Please refer to the document [3] for the extended information of OpenCPU Security Data Solution.

11.4. Power Saving Mode

You can call Ql_SleepEnable to enable power saving mode (low power consumption mode, or sleep mode), the system will switch to 32K clock to work when CPU is idle. Under sleep mode, UART port cannot receive data. QL_SleepDisable can disable the power saving mode.

Please note that when application launches the GPT timer (fast timer), the module cannot enter into power saving mode.

11.5. UART Port

OpenCPU module provides three serial ports. The default configuration parameters are: 115200 baud rate, and 8N1. The data buffer size of UART port is 2KB.

When received the event “EVENT_UART_READY_TO_READ” in UART callback, application should call Ql_UART_Read to read all data out of the UART buffer. Or application cannot receive this event when new data comes and cause UART port “dead”.

When you call Ql_UART_Write to send data, if the returned bytes number is less than the bytes number to write, which means the UART buffer is full, the program should stop sending data and wait for the EVENT_UART_READY_TO_WRITE event in UART callback to send the remained data.

11.6. Timer

In OpenCPU, there are two kinds of timers, one is common timer, and the other is fast timer. There’re 10 common timers available for each task, and there’s only one fast timer for whole application. The common timer is probably delayed because of task scheduling, so a common timer is a task timer, while the fast timer doesn’t belong to any task, and it’s triggered by interruption. So the fast timer has good real-time. However, please don’t put too much work load in the interruption handler, which can cause system exception.

11.7. Dynamic Memory

You can call Ql_MEM_Alloc() to apply for the specified size of dynamic memory, and call Ql_MEM_Free() to free the memory. The size of the dynamic memory of application is maximum 500KB available.

11.8. GPRS and TCP

For simplifying GPRS and TCP socket programming, the GPRS/TCP related API functions have been designed for synchronous APIs, which means the API functions return the final result. The maximum time for the synchronous API to return is 180s.

Synchronous APIs about GPRS:

- s32 Ql_GPRS_ActivateEx(u8 contextId, bool isBlocking); the timeout is 180s.
- s32 Ql_GPRS_DeactivateEx(u8 contextId, bool isBlocking); the timeout is 90s.

Synchronous APIs about TCP socket:

- s32 Ql_SOC_ConnectEx(s32 socketId, u32 remoteIP, u16 remotePort, bool isBlocking); the timeout is 75s.
- s32 Ql_IpHelper_GetIPByHostNameEx (u8 contextId, u8 requestId, u8 *hostName, u32* ipCount,u32* ipAddress); the timeout is 60s.

12 Appendix

12.1. Reference

Table 3: Reference Document

SN	Document Name
[1]	ZF_ OpenCPU_GCC_Installation_Guide
[2]	ZF_ OpenCPU_User Guide
[3]	ZF_OpenCPU_Security_Data_Application_Note
[4]	ZF_OpenCPU_GCC_Eclipse_User_Guide
[5]	ZF_ Firmware_Upgrade_Tool_Lite_GS2_User_Guide
[6]	ZF_QFlash_User_Guide
[7]	ZF_OpenCPU_Watchdog_Application_Note