Radiocrafts

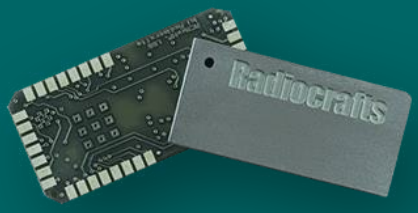
Embedded Wireless Solutions

Fast to Market. Proven Quality.

AN029: C-Programming of ICI

By: Omar Khalil

2019-02-01



C programming of ICI

By Omar Khalil

Background

The purpose of this application note is to demonstrate how easy it is to create your own application using Radiocraft's ICI (intelligent C-programmable I/O). ICI enables users to program and customize their own applications inside the RIIoT module without the use of an external application microcontroller. To read more about the ICI interface please refer to *SPR SDK User Manual*

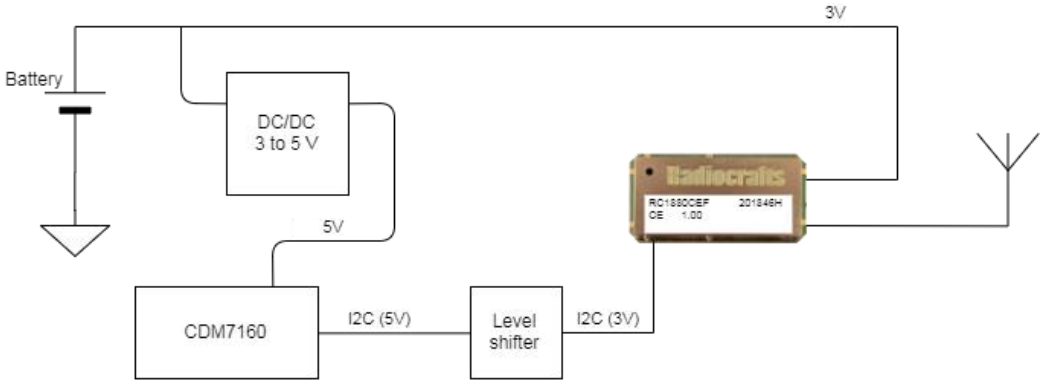


Figure 1. Block schematic of CO2 sensor solution

The example used in this demonstration is to collect data from the CDM7160 CO2 sensor, by Figaro. Above is shown the ICI block schematic that is the HW foundation for this application note. The basic blocks needed are a RC1880CEF-SPR module, a battery, an antenna and a sensor. Since the sensor in this example is s CO2 sensor operating at 5V, extra DC/DC converter and level shifter is needed.

This application note covers a sensor node in RIIoT, but by modifying the network setup part, the code can be used also on RIIM (Radiocraft's Industrial IP Mesh).

At the beginning, we present the application requirements, then an overview of the code architecture where different C-functions and their various events are introduced. Afterwards, a flow chart is presented where the code steps are explained chronologically. Lastly, each block of the code is explained individually. You can find the complete code at the bottom of this document.

By the end of this application note, the user should be able to understand how to modify the code, based on the knowledge he gained in this document, to fit into his own application with minimal C-programming knowledge. Though, this tutorial assumes the reader has basic C-knowledge.

Requirements for application

Each application has its own requirements. The example covered in this document has only a few requirements. The requirements are numbered to be referred to later in the text.

1. Auto join a network at power on.
2. Remember/store old network at power down to make e.g. battery change easy.
3. If old network is not found after 1 minute, search for new network.
4. Read C02 Sensor periodically every 2 seconds and send result via RF to the gateway.

Understanding an event-based code

A key to understanding a code on a conceptual level is to understand, not only what each line of code means, but to also have a system-level view of why functions are arranged in this order, what each function means, in addition to when is it called. To gain this perspective, have a look a figure 2 below. Figure 2 represents a cause-and-effect diagram of the code where it shows every event possible and what are its consequences in terms of functions called and actions performed by these functions.

The main function called at the event of power-up is the setup function, where all initialization and setup take place. If the readSensorTimer is activated, it calls the readSensor function which read the designated registers. In the case when the sensor is trying to re-join a network, the function networkStateChangedHandler is called to start the resetNetwork timer. When this timer times out, the sensor stops trying to re-join and the network settings are reset.

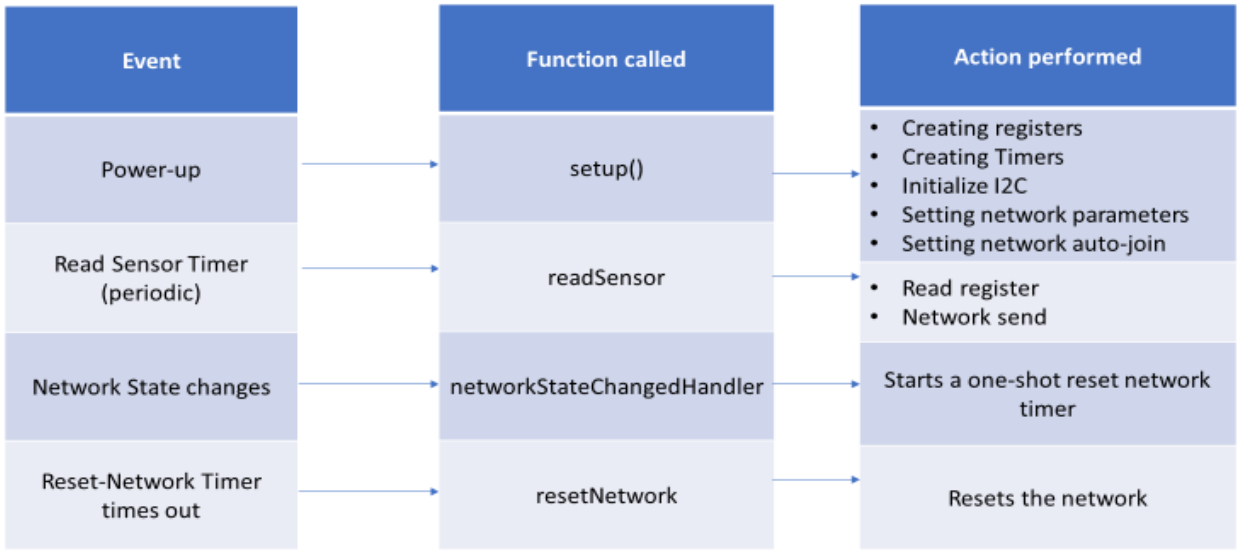


Figure 2. Events and functions

Code flowchart

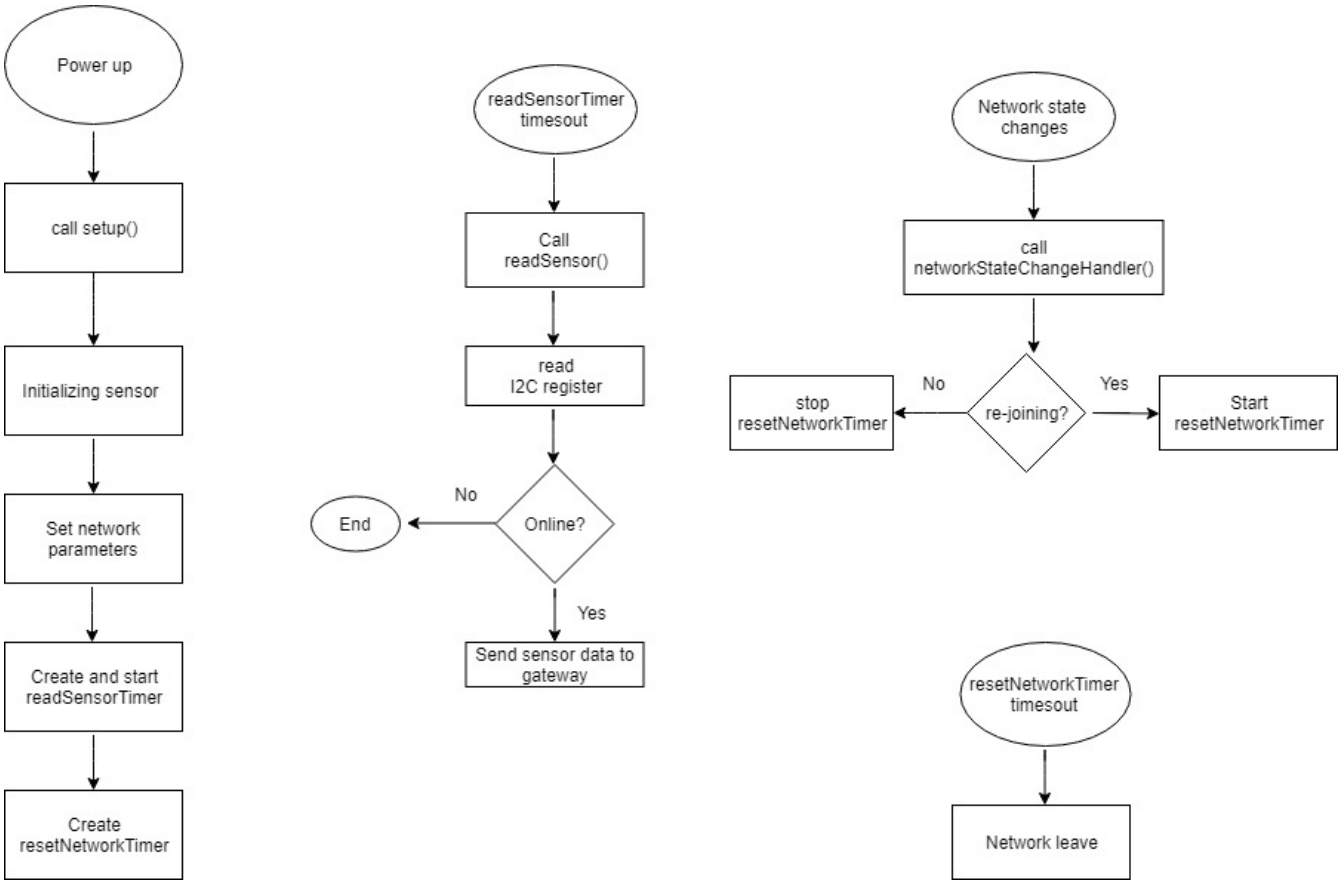


Figure 3. Flowchart of the code

To add a new perspective to understanding the code, a flow chart is presented in figure 3. A flow chart portrays the steps, logical statements, and decision in a code, but in a graphical way, to give the code-reader an idea of the possible steps and their resulting outcomes before having to read the actual code.

The flowchart above shows the steps taking place after start-up. Such steps include, sensor initialization and the setup of the data rate and channel to be used by the sensor board. In addition to creating timers which will be used later. In the middle and right hand-side parts of the flowchart, we can see 3 possible scenarios and how the code is designed to deal with them.

Understanding each block of code and its function

1) Including the header file:

The code starts by including Radiocraft's header file which contains the declarations for the APIs used in the code.

```
#include "spr_app.h"
```

2) Defining the constants which will be used in the code:

```
/********** Constants ***********/
#define CDM7160_I2C_ADDRESS 0x69
#define REG_CTL 0x01
#define REG_ST1 0x02
#define REG_DAL 0x03
#define REG_DAH 0x04

#define POWER_DOWN_MODE 0x00
#define CONTINUOUS_OPERATION_MODE 0x06

#define SENSOR_ID_CDM7160 0x02
```

Second part of the code is to define the constants which will be referred to in following parts of the code. From the sensor's data sheet, you can find the registers' addresses in addition to the I2C address needed to carry out specific tasks. By assigning these addresses names, you can later refer to the registers by the names you give them, such as "REG_DAL" (from the sensor's data sheet) instead of having to type its hexadecimal address, thus, increasing the code's readability.

3) Declaring variables and functions which will be used in the code:

```
/********** App Variables ***************/
// Variables shared between functions in your app.
// Their values must be initialized in Setup(), and not here.
static TimerId readSensorTimer;
static TimerId resetNetworkTimer;
static bool online;

/********** Handler Declarations ***************/
static uint8_t readRegister(uint8_t address);
static SPR_Status writeRegister(uint8_t address, uint8_t value);
static void readSensor(void);
static void networkStateChangedHandler(NetworkState state);
static void receivedMessageHandler(uint8_t len, uint8_t *message);
static void resetNetwork(void);
```

The 3 "app variables" are variables used in the main setup function, thus they must be declared at the top. They are basically the 2 timers used and a Boolean variable. In the "Handler Declaration" part, the functions referred to in the main setup function must be declared here. The "readRegister" and "writeRegister" functions are declared first.

The “networkStateChangeHandler” is an event handler which sets the steps to be carried out by your application in case the network state changes. The “receivedMessageHandler” is also another event handler which set the steps to be carried out in the case of receiving a message. “ResetNetwork” function is used in case you need to reset your network connection.

4) The main setup function:

```
void Setup()
{
    // initialize global variables
    online = false;

    I2C.init(I2C_400KHZ);

    writeRegister(REG_CTL, CONTINUOUS_OPERATION_MODE);

    // Configure the network
    Network.setFreqBand(FREQ_868_MHZ);
    Network.setDataRate(DATA_RATE_50_KBPS);
    uint8_t channelMask[CHANNEL_BITMAP_SIZE];

    channelMask[0] = 0x01; //just scan the first channel
    Network.setChannelMask(channelMask);
    Network.setNetworkStateChangeHandler(networkStateChangedHandler);
    Network.setReceivedMessageHandler(receivedMessageHandler);
    Network.setAutoJoin(true);
    Node.setBatteryPowered(true);
    Network.setPollRate(5 * SECOND);

    readSensorTimer = Timer.create(PERIODIC, 2*SECOND, readSensor);
    Timer.start(readSensorTimer);

    // timer to reset the network if rejoin is not successful within the duration
    resetNetworkTimer = Timer.create(ONE_SHOT, 1*MINUTE, resetNetwork);
}
```

This is the core function of the code, where all the necessary operations are carried out. You do not need to implicitly call this function as it is automatically called by the framework at start-up phase. The variable called “online” was declared at the top of the code as a “Boolean” variable, but it can’t be given a value at the top in the declaration part, thus here in the main function it is given an initial value of “false”.

Afterwards, the I2C connection is set, so is the operation mode which is set to “continuous operation mode”. Following are network configuration steps. Where the channel, data rate, and the frequency band are set. Which basically corresponds to requirements 1 and 2 from the requirements’ list mentioned earlier in this document.

Finally, the “readSensorTimer” is created to read sensor data every 2 seconds, then it is started. The resetNetworkTimer is also created to reset the network in case network joining is not successful. Which basically corresponds to requirements3 and 4 from the requirements’ list mentioned earlier in this document.

5) Implementing other functions:

5.1 readRegister

```
static uint8_t readRegister(uint8_t address)
{
    uint8_t writeBuffer[1];
    uint8_t readBuffer[1];
    writeBuffer[0] = address;
    I2C.transfer(CDM7160_I2C_ADDRESS, writeBuffer, sizeof(writeBuffer),readBuffer, sizeof(readBuffer));
    return readBuffer[0];
}
```

The readRegister function takes in the address of the register you want to read from, and returns the value read. Thus, in the first two lines of the function implementation, two arrays, of size one-byte, are created. The writeBuffer array is to write the address of the register you need to read from. While the readBuffer array is to store what you read from that register.

Reading from a register is not a one-step procedure as in writing in a register. You have to first write the address of the register, then you can read. Thus, the 3rd line of the function entails that after entering the I2C address of the sensor, you write the address of the register, then you can read.

5.2 writeRegister

```
static SPR_Status writeRegister(uint8_t address, uint8_t value)
{
    uint8_t writeBuffer[2];
    writeBuffer[0] = address;
    writeBuffer[1] = value;
    SPR_Status status = I2C.write(CDM7160_I2C_ADDRESS, writeBuffer, sizeof(writeBuffer));

    return status;
}
```

Contrary to the “read” operation, writing into a register is straight-forward. First you create the array where you store what you want to write and where to write it (address of the register).

5.3 readSensor

```
static void readSensor(void)
{
    uint8_t status = readRegister(REG_ST1);
    uint8_t data_l = readRegister(REG_DAL);
    uint8_t data_h = readRegister(REG_DAH);
    uint16_t data = ((uint16_t)data_h << 8) + data_l;
    Debug.printline("ST1=%x CO2 concentration = %dppm", status, data);

    if (online) {
        uint8_t message[4];
        message[0] = SENSOR_ID_CDM7160;
        message[1] = status;
        message[2] = data_l;
        message[3] = data_h;
        Debug.printline("send st=%x", Network.send(sizeof(message), message));
    }
}
```

To fulfil requirement 4, the function “readSensor” performs 3 tasks. Firstly, it reads the data in the registers (where readings are stored) then rearranges the string of data to make the highest significant bits on the left and least significant bits on the right. Afterwards, it forms the message that will be sent to the gateway and sends it. As seen by the second part of the function, the message is an array of 4 bytes, where the first byte represents the sensor address, the second represents the connection status, and the third and fourth carry the sensor reading.

5.4 networkStateChangedHandler

```
static void networkStateChangedHandler(NetworkState state)
{
    Debug.printf("Network State: %s\r\n", Network.getNetworkStateString());

    online = (ONLINE == state);

    if (ONLINE == state || REJOINING == state)
        Debug.printline("PanId=%2x Ch=%d ShortAddr=%2x", Network.getPanId(), Network.getChannel(), Network.getShortAddress());

    if (REJOINING == state)
        Timer.start(resetNetworkTimer); // reset network after a time if rejoining doesn't work
    else
        Timer.stop(resetNetworkTimer);
}
```

This event handler dictates certain actions to be taken according to changes in the network state. If the sensor is trying to re-join the network, the “resetNetworkTimer” starts, and if it times out before a network is joined, it resets the network, as dictated by requirement 3.

Summary

In this application not we have shown how easy it is to setup an ICI application to read and report a CO2 sensor using less than a hundred lines of code. Please find below the actual code used in this tutorial.

```
#include "spr_app.h"

/********* Constants **********/
#define CDM7160_I2C_ADDRESS 0x69
#define REG_CTL 0x01
#define REG_ST1 0x02
#define REG_DAL 0x03
#define REG_DAH 0x04

#define POWER_DOWN_MODE 0x00
#define CONTINUOUS_OPERATION_MODE 0x06

#define SENSOR_ID_CDM7160 0x02

/********* App Variables ****************/
// Variables shared between functions in your app.
// Their values must be initialized in Setup(), and not here.
static TimerId readSensorTimer;
static TimerId resetNetworkTimer;
static bool online;

/********* Handler Declarations ****************/
static uint8_t readRegister(uint8_t address);
static SPR_Status writeRegister(uint8_t address, uint8_t value);
static void readSensor(void);
static void networkStateChangedHandler(NetworkState state);
static void receivedMessageHandler(uint8_t len, uint8_t *message);
static void resetNetwork(void);

/**
* Setup() is called by the framework on startup
*/
void Setup()
{
   Debug.printf("My Setup\r\n");

   // initialize global variables
   online = false;

   I2C.init(I2C_400KHZ);
```

```
    writeRegister(REG_CTL, CONTINUOUS_OPERATION_MODE);

    // Configure the network
    Network.setFreqBand(FREQ_868_MHZ);
    Network.setDataRate(DATA_RATE_50_KBPS);
    uint8_t channelMask[CHANNEL_BITMAP_SIZE];

    channelMask[0] = 0x01; //just scan the first channel
    Network.setChannelMask(channelMask);
    Network.setNetworkStateChangeHandler(networkStateChangedHandler);
    Network.setReceivedMessageHandler(receivedMessageHandler);
    Network.setAutoJoin(true);
    Node.setBatteryPowered(true);
    Network.setPollRate(5 * SECOND);

    readSensorTimer = Timer.create(PERIODIC, 2*SECOND, readSensor);
    Timer.start(readSensorTimer);

    // timer to reset the network if rejoin is not successful within the duration
    resetNetworkTimer = Timer.create(ONE_SHOT, 1*MINUTE, resetNetwork);
}

/********* Handler Functions ***************/
static uint8_t readRegister(uint8_t address)
{
    uint8_t writeBuffer[1];
    uint8_t readBuffer[1];
        writeBuffer[0] = address;
    I2C.transfer(CDM7160_I2C_ADDRESS, writeBuffer, sizeof(writeBuffer),readBuffer, sizeof(readBuffer));
    return readBuffer[0];
}

static SPR_Status writeRegister(uint8_t address, uint8_t value)
{
    uint8_t writeBuffer[2];
        writeBuffer[0] = address;
        writeBuffer[1] = value;
    SPR_Status status = I2C.write(CDM7160_I2C_ADDRESS, writeBuffer, sizeof(writeBuffer));

    return status;
}

static void readSensor(void)
{
    uint8_t status = readRegister(REG_ST1);
    uint8_t data_l = readRegister(REG_DAL);
    uint8_t data_h = readRegister(REG_DAH);
    uint16_t data = ((uint16_t)data_h << 8) + data_l;
    Debug.printline("ST1=%x CO2 concentration = %dppm", status, data);
```

```
    if (online) {
        uint8_t message[4];
        message[0] = SENSOR_ID_CDM7160;
        message[1] = status;
        message[2] = data_l;
        message[3] = data_h;
        Debug.printline("send st=%x", Network.send(sizeof(message), message));
    }
}

static void networkStateChangedHandler(NetworkState state)
{
    Debug.printf("Network State: %s\r\n", Network.getNetworkStateString());

    online = (ONLINE == state);

    if (ONLINE == state || REJOINING == state)
        Debug.printline("PanId=%2x Ch=%d ShortAddr=%2x", Network.getPanId(), Network.getChannel(),
Network.getShortAddress());

    if (REJOINING == state)
        Timer.start(resetNetworkTimer); // reset network after a time if rejoining doesn't work
    else
        Timer.stop(resetNetworkTimer);
}

static void receivedMessageHandler(uint8_t len, uint8_t *message)
{
    Debug.printf("Received message: ");
    Debug.printArray(len, message);
}

static void resetNetwork(void)
{
    Debug.printline("Reset network");
    Network.leave();
}
```

Document Revision History

Document Revision	Changes
1.0	First release

Disclaimer

Radiocrafts AS believes the information contained herein is correct and accurate at the time of this printing. However, Radiocrafts AS reserves the right to make changes to this product without notice. Radiocrafts AS does not assume any responsibility for the use of the described product; neither does it convey any license under its patent rights, or the rights of others. The latest updates are available at the Radiocrafts website or by contacting Radiocrafts directly.

As far as possible, major changes of product specifications and functionality will be stated in product specific Errata Notes published at the Radiocrafts website. Customers are encouraged to check regularly for the most recent updates on products and support tools.

Trademarks

RC232™ is a trademark of Radiocrafts AS. The RC232™ Embedded RF Protocol is used in a range of products from Radiocrafts. The protocol handles host communication, data buffering, error check, addressing and broadcasting. It supports point-to-point, point-to-multipoint and peer-to-peer network topologies.

All other trademarks, registered trademarks and product names are the sole property of their respective owners.

Life Support Policy

This Radiocrafts product is not designed for use in life support appliances, devices, or other systems where malfunction can reasonably be expected to result in significant personal injury to the user, or as a critical component in any life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness. Radiocrafts AS customers using or selling these products for use in such applications do so at their own risk and agree to fully indemnify Radiocrafts AS for any damages resulting from any improper use or sale.