```
/* This is an investigative variabnt of the Raspberry Pi, receive, side of my
 * 'step-5' milestone: * Transmit / Receive capacitance measurement. In this
 * code I am trying to work out what is going on with the receipt and rendering
 * of the non-string and non-8-bit intteger data values. Floats and unsigned long
 * are not coming across correctly. One theory is this is due to differing
 * endian encoding on the tiny84 vs the RPi. So I am herein trying to work this
 * out.
 *
 * 06-19-2022: First iteration, using code that has started as a copy of CapDataReceive_02a.cpp
 * 06-20-2022: Inserting HEX output to investigate endian difference
 */
#define VERSION "06-26-2022 rel 04"

/*
 * For nRF24 radio chip documentation see https://nRF24.github.io/RF24
 * For the approach on capacitance measurement on the ATTiny84 side --:
 *   RCTiming_capacitance_meter || Paul Badger 2008
 *   @ https://www.arduino.cc/en/Tutorial/Foundations/CapacitanceMeter

 * This source based on "manualAcknowledgements.cpp" in the RPi nRF24 library
 * noted above.
 */
#include <ctime>       // time()
#include <iostream>    // cin, cout, endl
#include <iomanip>     // format manipulators for use with cout
#include <string>      // string, getline()
#include <time.h>      // CLOCK_MONOTONIC_RAW, timespec, clock_gettime()
#include <RF24/RF24.h> // RF24, RF24_PA_LOW, delay()


/* Macro uised to reverse endian byte-order. See @ https://stackoverflow.com/a/41196306 */
#define REVERSE_BYTES(...) do for(size_t REVERSE_BYTES=0; REVERSE_BYTES<sizeof(__VA_ARGS__)>>1; +
+REVERSE_BYTES)\
    ((unsigned char*)&(__VA_ARGS__))[REVERSE_BYTES] ^= ((unsigned char*)&(__VA_ARGS__))
    [sizeof(__VA_ARGS__)-1-REVERSE_BYTES],\
    ((unsigned char*)&(__VA_ARGS__))[sizeof(__VA_ARGS__)-1-REVERSE_BYTES] ^= ((unsigned
    char*)&(__VA_ARGS__))[REVERSE_BYTES],\
    ((unsigned char*)&(__VA_ARGS__))[REVERSE_BYTES] ^= ((unsigned char*)&(__VA_ARGS__))
    [sizeof(__VA_ARGS__)-1-REVERSE_BYTES];\
while(0)

using namespace std;


void getHexOfFloat(float f) {
    unsigned char* pFloat = (unsigned char*) & f;
    for (size_t k=0; k<sizeof(f); k++) {
        cout << setfill('0') << setw(2) << hex << (unsigned int)pFloat[k] << " ";
    }
        cout << dec;
}

void getHexOfString(char* s, int iLen) {
    for (int k=0; k<iLen; k++) {
        cout << setfill('0') << setw(2) << hex << (unsigned int)s[k] << " ";
    }
    cout << dec;
}
```

```
void showHexOfBytes(unsigned char* b, int iLen) {
    for (int k=0; k<iLen; k++) {
        cout << setfill('0') << setw(2) << hex << (unsigned int)b[k] << " ";
    }
    cout << dec;
}




/****************** Linux ***********************/
// Radio CE Pin, CSN Pin, SPI Speed
// CE Pin uses GPIO number with BCM and SPIDEV drivers, other platforms use their own pin
numbering
// CS Pin addresses the SPI bus number at /dev/spidev<a>.<b>
// ie: RF24 radio(<ce_pin>, <a>*10+<b>); spidev1.0 is 10, spidev1.1 is 11 etc..

// Generic:
RF24 radio(22, 0);
/****************** Linux (BBB,x86,etc) ***********************/
// See http://nRF24.github.io/RF24/pages.html for more information on usage
// See http://iotdk.intel.com/docs/master/mraa/ for more information on MRAA
// See https://www.kernel.org/doc/Documentation/spi/spidev for more information on SPIDEV

// For this milestone I am using a payload suitable suitable for
// characterizing a capacitance value.
// Make data structures to store the receive and ACK payloads.
struct RxPayloadStruct {
  char statusText[11];            // For use in debugging.
  float testFloat1 = 0;
  uint32_t chargeTime = 0;
  float testFloat2 = 0;
  char units[4];                  // nFD, mFD, FD
  float capacitance = 0;
};
RxPayloadStruct rxPayload;

// Make a data structure to store the ACK payload
struct AckPayloadStruct {
  char message[11];               // Outgoing message, up to 10 chrs+Null.
  uint8_t counter;
};
AckPayloadStruct ackPayload;

void setRole(); // prototype to set the node's role
void master();  // prototype of the TX node's behavior
void slave();   // prototype of the RX node's behavior

// custom defined timer for evaluating transmission time in microseconds
struct timespec startTimer, endTimer;
uint32_t getMicros(); // prototype to get ellapsed time in microseconds


int main(int argc, char** argv) {
    // perform hardware check
    if (!radio.begin()) {
        cout << "radio hardware is not responding!!" << endl;
```

```
        return 0; // quit now
    }

    // Let these addresses be used for the pair
    uint8_t address[2][6] = {"1Node", "2Node"};
    // It is very helpful to think of an address as a path instead of as
    // an identifying device destination

    // to use different addresses on a pair of radios, we need a variable to
    // uniquely identify which address this radio will use to transmit
    bool radioNumber = 1; // 0 uses address[0] to transmit, 1 uses address[1] to transmit

    // print example's name
    cout << endl<< argv[0] << " [" << VERSION << "]" << endl;

    // Set the radioNumber via the terminal on startup
    cout << "Which radio is this? Enter '0' or '1'. Defaults to '0' ";
    string input;
    getline(cin, input);
    radioNumber = input.length() > 0 && (uint8_t)input[0] == 49;

    // to use ACK payloads, we need to enable dynamic payload lengths
    radio.enableDynamicPayloads();    // ACK payloads are dynamically sized

    // Acknowledgement packets have no payloads by default. We need to enable
    // this feature for all nodes (TX & RX) to use ACK payloads.
    radio.enableAckPayload();

    // Set the PA Level low to try preventing power supply related problems
    // because these examples are likely run with nodes in close proximity to
    // each other.
    radio.setPALevel(RF24_PA_LOW);  // RF24_PA_MAX is default.

    // set the TX address of the RX node into the TX pipe
    radio.openWritingPipe(address[radioNumber]);     // always uses pipe 0

    // set the RX address of the TX node into a RX pipe
    radio.openReadingPipe(1, address[!radioNumber]); // using pipe 1

    // For debugging info
    // radio.printDetails();        // (smaller) function that prints raw register values
    radio.printPrettyDetails();     // (larger) function that prints human readable data

    // ready to execute program now
    setRole(); // calls master() or slave() based on user input
    return 0;
}


/**
 * set this node's role from stdin stream.
 * this only considers the first char as input.
 */
void setRole() {
    string input = "";
    while (!input.length()) {
        cout << "*** PRESS 't' to begin transmitting to the other node\n";
        cout << "*** PRESS 'r' to begin receiving from the other node\n";
```

```
        cout << "*** PRESS 'q' to exit" << endl;
        getline(cin, input);
        if (input.length() >= 1) {
            if (input[0] == 'T' || input[0] == 't')
                master();
            else if (input[0] == 'R' || input[0] == 'r')
                slave();
            else if (input[0] == 'Q' || input[0] == 'q')
                break;
            else
                cout << input[0] << " is an invalid input. Please try again." << endl;
        }
        input = ""; // stay in the while loop
    } // while
} // setRole()


/**
 * make this node act as the transmitter - but not in this iteration.
 */
void master() {
    cout << "You selected TRANSMIT mode. ";
    cout << endl;
    cout << "For Milestone #5 the Raspberry Pi can only ";
    cout << "act as a receiver. ";
    cout << endl;
    cout << "Quit or press r to try again.";
    cout << endl;
} // master


/**
 * make this node act as the receiver
 */
void slave() {
    memcpy(ackPayload.message, "Pkt Count ", 10);                    // set the ackPayload message
    ackPayload.counter = 0;                                         // set the ackPayload counter

    // load the ackPayload for first received transmission on pipe 0
    radio.writeAckPayload(1, &ackPayload, sizeof(ackPayload));

    radio.startListening();                                         // put radio in RX mode
    time_t startTimer = time(nullptr);                              // start a timer
    while (time(nullptr) - startTimer < 12) {                        // use 12 second timeout
        uint8_t pipe;
        if (radio.available(&pipe)) {                               // is there a received payload? get
the pipe number that recieved it
            uint8_t bytes = radio.getDynamicPayloadSize();          // yes, get it's size
            radio.read(&rxPayload, sizeof(rxPayload));              // fetch payload from RX FIFO

            unsigned char* pPayloadAsChar = (unsigned char*) & rxPayload;

            unsigned int wdthVarName = 14;
            unsigned int wdthValue = 14;

            cout << endl;
            cout << setw(0) << setfill(' ');
```

```
        cout << "Recieved " << (unsigned int)bytes;
        cout << " bytes on pipe " << (unsigned int)pipe << " | ";

        cout << "Total Size Of rxPayload: " << sizeof(rxPayload) << " || ";
        cout << "Payload bytes as HEX --: ";
        showHexOfBytes(pPayloadAsChar,sizeof(rxPayload));
        cout << endl;
        cout << setw(60) << setfill('-') << "-" << endl;

        cout << setw(wdthVarName) << setfill(' ') << " statusText: ";
        cout << setw(2) << (unsigned int)sizeof(rxPayload.statusText);
        cout << " | " << setw(wdthValue) << rxPayload.statusText << " | 0x ";
        getHexOfString(rxPayload.statusText, sizeof(rxPayload.statusText));
        cout << endl;

        cout << setw(wdthVarName) << setfill(' ') << " testFloat1: ";
        cout << setw(2) << (unsigned int)sizeof(rxPayload.testFloat1);
        cout << " | " << setw(wdthValue) << setprecision(2) << rxPayload.testFloat1 << " | 0x
";
        getHexOfFloat(rxPayload.testFloat1);
        cout << endl;

        cout << setw(wdthVarName) << setfill(' ') << " chargeTime: ";
        cout << setw(2) << (unsigned int)sizeof(rxPayload.chargeTime);
        cout << " | " << setw(wdthValue) << rxPayload.chargeTime << " | 0x ";
        showHexOfBytes((unsigned char*)&rxPayload.chargeTime,sizeof(rxPayload.chargeTime));
        //getHexOfLong(rxPayload.testFloat1);
        cout << endl;

        cout << setw(wdthVarName) << setfill(' ') << " testFloat2: ";
        cout << setw(2) << (unsigned int)sizeof(rxPayload.testFloat2);
        cout << " | " << setw(wdthValue) << rxPayload.testFloat2 << " | 0x ";
        getHexOfFloat(rxPayload.testFloat2);
        cout << endl;

        cout << setw(wdthVarName) << setfill(' ') << " units: ";
        cout << setw(2) << (unsigned int)sizeof(rxPayload.units);
        cout << " | " << setw(wdthValue) << rxPayload.units << " | 0x ";
        getHexOfString(rxPayload.units, sizeof(rxPayload.units));
        cout << endl;

        cout << setw(wdthVarName) << setfill(' ') << " capacitance: ";
        cout << setw(2) << (unsigned int)sizeof(rxPayload.capacitance);
        cout << " | " << setw(wdthValue) << rxPayload.capacitance << " | 0x ";
        getHexOfFloat(rxPayload.capacitance);
        cout << endl << endl;

        // Print what we are going to respond back with
        cout << " Sent in Response: ";
        cout << ackPayload.message;
        cout << (unsigned int)ackPayload.counter << endl;   // print ACK payload sent

        startTimer = time(nullptr);                         // reset timer

        // save incoming counter & increment
        ackPayload.counter = ackPayload.counter + 1;
        // load the payload for the next received transmission on pipe 0
        radio.writeAckPayload(1, &ackPayload, sizeof(ackPayload));
```

```
        } // if received something
    } // while
    cout << "Timeout While Waiting to Receive Data. Leaving RX role." << endl;
    cout << "You may quite or press r to try again." << endl;
    radio.stopListening(); // recommended idle behavior is TX mode
    // This then ends this routine, returning back to the setRole() while loop,
    // allowing the user to quit or initiate another try.
} // slave


/**
 * Calculate the ellapsed time in microseconds
 */
uint32_t getMicros() {
    // this function assumes that the timer was started using
    // `clock_gettime(CLOCK_MONOTONIC_RAW, &startTimer);`

    clock_gettime(CLOCK_MONOTONIC_RAW, &endTimer);
    uint32_t seconds = endTimer.tv_sec - startTimer.tv_sec;
    uint32_t useconds = (endTimer.tv_nsec - startTimer.tv_nsec) / 1000;

    return ((seconds) * 1000 + useconds) + 0.5;
}
```