```
/* 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_02b.cpp
 * 06-20-2022: Inserting HEX output to investigate endian difference
 * 06-26-2022: Manually load struct from raw bytes received
 * 06-27-2022: Continue manual loading work, based on endianTest_05.cpp success on my desktop.
 */
#define VERSION "06-27-2022 rel 05"

/*
 * 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()

using namespace std;


/****************** 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



/* ================================================================================
   Gloabal Variable Declarations
   ================================================================================
*/

    /* Using a struct to directly load the received payload seems to fail
```

```
       due to boundary-alignment issues. Whole bytes seem to get dropped
       or shifted as compared to the originating struct on the transmit
       side. NOTE: At first I thought this was an endian mismatch issue,
       but after much debugging have determined that is not the issue, the
       issue is struct boundary/alignment mismatch.
    */
    /* SO - below declare is a variable to read the received bytes into.
       And I will see if I can work out how best to then take the raw
       bytes and assign them into the intended rxPayload struct.
    */
uint8_t rxBytes[40];

    /* Struct to hold the data received in
       from the ATTiny's nRF24
    */
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;

    /* Structure to store the outgoing ACK payload
    */
struct AckPayloadStruct {
  char message[11];               // Outgoing message, up to 10 chrs+Null.
  uint8_t counter;
};
AckPayloadStruct ackPayload;

    /* Custom defined timer for evaluating transmission
       time in microseconds.
    */
struct timespec startTimer, endTimer;



/* ================================================================================
   Function prototypes
   ================================================================================
*/
void setRole();                                          // prototype to set the node's
role
void slave();                                            // prototype of the RX node's
behavior
void loadRxStruct(RxPayloadStruct* pStruct, uint8_t* pBytes);   // prototype for struct load
function
void showHexOfBytes(unsigned char* b, int iLen);                // display hex value of variables



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;

} // Main()



/* ================================================================================
   Local Functions
   ================================================================================
*/

/* 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 '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] == '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()


/* Performs receiver-role tasks */
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(&rxBytes[0], sizeof(rxBytes));      // fetch payload from RX FIFO

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

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

                /* Show the sizes of things.
                */
            cout << "Recieved " << (unsigned int)bytes;
            cout << " bytes on pipe " << (unsigned int)pipe << " | ";
            cout << "Size of rxBytes[]: " << sizeof(rxBytes) << " || ";
            cout << "Size Of rxPayload struct: " << sizeof(rxPayload) << " | " << endl;

                /* Inspect the received data received from ATTiny.
                */
            cout << setfill('-') << setw(sizeof(rxBytes)*3) << "-" << setfill(' ') << endl;
            cout << "rxBytes in HEX / rxBytes[] index offset ---: " << endl;
            showHexOfBytes((unsigned char *)&rxBytes,sizeof(rxBytes));
```

```
        cout << endl;
        for (unsigned int k=0; k<sizeof(rxBytes); k++) {
            cout << setw(2) << setfill('0') << k << " ";
        }
        cout << endl;
        cout << setfill('-') << setw(sizeof(rxBytes)*3) << "-" << setfill(' ') << endl;

            /* 'Manually' load rxPayload structure from the received bytes array.
            */
        loadRxStruct(&rxPayload, rxBytes);

            /* Display received data, as loaded into the struct, on the console.
             */
        cout << setw(wdthVarName) << setfill(' ') << " statusText: ";
        cout << setw(2) << (unsigned int)sizeof(rxPayload.statusText);
        cout << " | " << setw(wdthValue) << rxPayload.statusText << " | 0x ";
        showHexOfBytes((unsigned char*)&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
        ";
        showHexOfBytes((unsigned char*)&rxPayload.testFloat1,sizeof(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));
        cout << endl;

        cout << setw(wdthVarName) << setfill(' ') << " testFloat2: ";
        cout << setw(2) << (unsigned int)sizeof(rxPayload.testFloat2);
        cout << " | " << setw(wdthValue) << rxPayload.testFloat2 << " | 0x ";
        showHexOfBytes((unsigned char*)&rxPayload.testFloat2,sizeof(rxPayload.testFloat2));
        cout << endl;

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

            /* Display onto the console 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

            /* Reset the timer.
            */
```

```
            startTimer = time(nullptr);

                /* Increment the 'payloads received' counter.
                */
            ackPayload.counter = ackPayload.counter + 1;

                /* Load the ACK payload for use on the next received
                   transmission on pipe 0
                */
            radio.writeAckPayload(1, &ackPayload, sizeof(ackPayload));
        } // if received something
    } // while

        /* Handle radio listening timout case. Which, other than a Control-C by
           the user, is the only way the slave() function ends. And upon ending
           we return back to the setRole() function's while loop, allowing the
           user to quite or initiate another try.
        */
    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
} // slave




/* Manually load the incoming data into a structure.
   ------------------------------------------------------------------------
   REQUIRES: The RxPayloadStruct defintion on this node exactly match the
             definition created on the transmit node.
 */
void loadRxStruct(RxPayloadStruct* pStruct, uint8_t* pBytes) {
    size_t offset = 0;

        /* Define columns for the display output.
        */
    unsigned int wdthCol1, wdthCol2, wdthCol3, wdthCol4;
    wdthCol1 = 23; wdthCol2 = 3; wdthCol3 = 25; wdthCol4 = 6;

    cout << setfill(' ');
    cout << fixed;

    cout << setw(wdthCol1) << "offset to statusText:" << setw(wdthCol2) << offset << endl;
    memcpy(pStruct->statusText, &pBytes[offset], sizeof(pStruct->statusText));
    offset = offset + sizeof(pStruct->statusText);

    cout << setw(wdthCol1) << "offset to testFloat1:" << setw(wdthCol2) << offset;
    pStruct->testFloat1 = *(float *)&pBytes[offset];
    cout << setw(wdthCol3) << left << " | Value of testFloat1= " << setw(wdthCol4) << right <<
    pStruct->testFloat1 << endl;
    offset = offset + sizeof(pStruct->testFloat1);

    cout << setw(wdthCol1) << "offset to chargeTime:" << setw(wdthCol2) << offset;
    pStruct->chargeTime = *(uint32_t *)&pBytes[offset];
    cout << setw(wdthCol3) << left << " | Value of chargeTime= " << setw(wdthCol4) << right <<
    pStruct->chargeTime << endl;
    offset = offset + sizeof(pStruct->chargeTime);
```

```
    cout << setw(wdthCol1) << "offset to testFloat2:" << setw(wdthCol2) << offset;
    pStruct->testFloat2 = *(float *)&pBytes[offset];
    cout << setw(wdthCol3) << left << " | Value of testFloat2= " << setw(wdthCol4) << right <<
    pStruct->testFloat2 << endl;
    offset = offset + sizeof(pStruct->testFloat2);

    cout << setw(wdthCol1) << "offset to units:" << setw(wdthCol2) << offset << endl;
    memcpy((unsigned char *)&pStruct->units, &pBytes[offset], sizeof(pStruct->units));
    offset = offset + sizeof(pStruct->units);

    cout << setw(wdthCol1) << "offset to capacitance:" << setw(wdthCol2) << offset;
    pStruct->capacitance = *(float *)&pBytes[offset];
    cout << setw(wdthCol3) << left << " | Value of capacitance= " << setw(wdthCol4) << right <<
    pStruct->capacitance << endl;

    cout << endl;
}


/* Display the HEX value of the bytes that store a variable.
   ------------------------------------------------------------------------
   PARMS:      1. The first byte of the variable to show the HEX for is passed in
            as a pointer to an unsigned char.
               2. The 2nd param is the length of the variable, i.e., the length
            of it's data type (or length of the string array if a string).
            E.g., use the sizeof() function on the variable to obtain this
            param.
 */
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;
}
```