

Sending and Receiving Strings via MiWi Wireless Network

Paul Krutty

Abstract

This Application Note will be expanding on the Microchip Simple MiWi Example. In the Microchip application note, the boards are set up to send “dummy” messages between two different nodes. I refer to the messages as “dummy” because they have no payload. In this note I will be explaining how to actually send and receive messages with actual content.

This Application Note will be expanding on the Microchip Simple MiWi Example. In the Microchip application note, the boards are set up to send “dummy” messages between two different nodes. I refer to the messages as “dummy” because they have no payload. In this note I will be explaining how to actually send and receive messages with actual content.

Some of the issues one faces when trying to send messages with a payload. First, the function for creating messages builds the messages one BYTE at a time. Fortunately, the solution to this problem is easy, in C the data type char can just be cast to BYTE directly.

```
char c = 'c';
```

```
BYTE a = (BYTE)c;
```

This however, raises another problem because we need characters not strings. Again, there is a pretty simple solution. As C programmers know character pointers or char* are collection of accessible chars. They can also be assigned just like strings.

```
char* stng = "This is a test";
```

```
char c = stng[0];
```

With this information and the compose message function, MiApp_WriteData(BYTE), we can send a message over the MiWi wireless network. See appendix one for the sample code.

Receiving messages poses its own set of problems. The messages are sent in the form of the rxMessage struct which is listed in appendix two. The first thought is to cast Payload back to the char* it was before it was sent. This however, just leaves a jumbled unreadable message. The problem is the receiving node doesn't know that byte was a char to begin with so casting it to a pointer is futile. The

solution is to use a char array. The struct even provides you with the PayloadSize so the array is easy to allocate.

```
char receive[rxMessage.PayloadSize +1];
```

Direct assignment does not work but iterating through the payload on can direct assign to the elements of the char array. The extra plus one seen above is for the addition of the null character, '\0', at the end of the array. The sample code for receiving messages can be seen in appendix three.

Appendix

1.

```
char* strng = "Message I want to send"; //define message to be sent
```

```
for(int n=0; n<strlen(strng); n++) //iterate through string, strlen() returns string length
```

```
    MiApp_WriteData((BYTE)strng[n]);
```

```
//Message ready to be sent
```

2.

```
typedef struct
{
    union
    {
        BYTE Val;
        struct
        {
            BYTE broadcast: 1;
            BYTE ackReq: 1;
            BYTE secEn: 1;
            BYTE repeat: 1;
            BYTE command: 1;
            BYTE srcPrnt: 1;
            BYTE dstPrnt: 1;
            BYTE altSrcAddr: 1;
        } bits
    } flags;

    BYTE *SourceAddress;
    BYTE *Payload;
    BYTE PayloadSize;
    BYTE RSSI;
    BYTE LQI;
} RECEIVED_MESSAGE;
```

3.

```
//message recived
```

```
char data[rxMessage.PayloadSize +1];           //define char array
```

```
for(int n=0; n<rxMessage.PayloadSize; n++)      //iterate through payload
```

```
    data[n] = rxMessage.Payload[n];             //input data to char array
```

```
data[rxMessage.PayloadSize] = '\0'             //append null character
```