Embedded Communications

an example of an xbee–computer wireless link

Jay Kickliter
jay.kickliter@gmail.com

2009

I Introduction

When working with embedded electronics, in particular microcontroller based systems, it's often necessary to monitor their state remotely. Due to their nature, microcontroller can be difficult to debug and monitor. They are basically computers without peripherals. It is possible to permanently attach a display and input device to a microcontroller, but it is usually impractical. They are often embedded in hard to reach dynamic systems. In cases where it is necessary to have the ability to monitor or reconfigure microcontroller based systems, doing so through a wireless link can greatly simplify the process. In the case of objects that move, like robots or unmanned-vehicles, a wireless link might be the only option.

There are several ways a wireless link be established between two systems. In this example, we'll be establishing a link between a microcontroller on the remote end, and monitoring the data it sends on a local computer. Although almost any combination of microcontroller and computer could be used, this example will use a Propeller Chip, an eight core microcontroller made my Parallax, and an Apple laptop computer.

For a wireless link, we'll use an XBee 900 module from Digi International. XBee modules are a good choice for small systems, because they are relatively cheap, and interchangeable. If for some reason you decide to replace a 900 MHz module with a 2.4 GHz module, minimal configurations changes are needed. The reason for using a 900 MHz module in this example is range. Digi makes 2.4 GHz modules that are a bit faster than the 900 MHz, but do not have the range that comes with using 900 MHz.

XBee modules can be operated in two different modes: transparent and API. Transparent mode is very convenient since the modules require no configuration; every byte pushed into one module exits the other as though there's a serial cable connecting them. Despite the convince of using transparent mode, we'll be using API mode. It's a bit more difficult to implement, but the benefits outweigh the added complexity. API mode allows greater control of the link, with the ability to send packets to an explicit address. In contrast, when data in sent in transparent mode, it is sent to all modules within range.

II Protocol

In transparent mode, all that is required to transmit a byte is to send that byte to the local XBee. After a certain timeout period, that XBee module packetizes any bytes in its buffer and sends them to the remote XBee(s), which output they bytes to whatever hardware is attached to them. API mode is a bit different. Much of the data formatting that is done automatically by the XBee in transparent mode needs to be performed before the user can transmit. For instance, the procedure to transmit the byte value 0x7E in transparent may look like this (in pseudo code):

```
serialTransmit(0x7E)
```

Sending data in API mode is not so simple. Before further discussion, it must added that API mode has two sub-modes: escaped and unescaped. Escaped mode requires that any data that can conflict with API control-characters must be preceded with 0x7D and XOR'd with 0x20. That procedure ensures that the XBee module won't be confused with any user data that happens to be same as any of the XBee control characters. This example will use escaped mode. It has the added benefit of making it easier to program receive routines.

Let's revisit our earlier example of sending the byte 0x7E. All that was required to send a byte was a single function call. In API mode several packet formatting steps must first take place:

1. Create a temporary buffer guaranteed to be bigger than needed

2. Insert 0x7E into position 0, this is the XBee frame delimiter, indicating the start of a new API packet

0
0x7E

3. Calculate the length of the packet, which in our case will be several bytes longer than the single 0x7E we want to send. Put the upper 8 bits of that length into buffer position 1, and the lower 8 bits into position 2

1	2
0x00	0x0F

4. Add the XBee API identifier to position 3. There are several ID's for different purposes, but we are trying to send a packet to another XBee, so we will use 0x10, which is the *TX Request* id

3
0x10

5. Insert a unique frame ID byte into position 4. This is not very important unless you want the module to respond with an acknowledge packet that it received the transmit request, however a byte needs to be here, so we'll use 0x01

4
0x01

6. Insert the 64 bit address of the remote module into positions 5–12, send most significant byte first, let's assume the remote XBee's address is 0x13A200404B2277

5	6	7	8	9	10	11	12
0x00	0x13	0xA2	0x00	0x40	0x4B	0x22	0x77

7. Insert the 16 bit address of the network you are operating on into positions 13–14, MSB first. We'll use the address 0xFFFE, since it is the default

13	14
0xFF	0xFE

8. Insert 0x00 into both position 15 and 16, these fields have definitions, but are not flexible, and serve no real purpose

15	16
0x00	0x00

9. Finally, staring at position 17, insert however many bytes you need to send, in our case, just insert 0x7E into position 17

17
0x7E

10. Calculate the checksum of the packet and insert it into position n, which in our case is 18. To calculate checksum, we add all the bytes from positions 3 to n-1, discarding any carry and keeping only the bottom 8 bits. Subtract this number from 0xFF. What is left over is our checksum.

    Checksum = 0xFF - ((0x00 + 0x01 + 0x00 + 0x13 + 0xA2 + 0x00 + 0x40 + 0x4B + 0x22 + 0x77 + 0xFF + 0xFE + 0x00 + 0x00 + 0x7E) & 0xFF)
    Checksum = 0xAA

11. Our (almost) completed packet looks like:

0	1	2	3	4	5	6	7	8	9
0x7E	0x00	0x0F	0x10	0x01	0x00	0x13	0xA2	0x00	0x40

10	11	12	13	14	15	16	17	18
0x4B	0x22	0x77	0xFF	0xFE	0x00	0x00	0x7E	0xAA

12. Starting at byte 1, we must enumerate through the whole packet and escape any control characters. Any byte with the vale 0x7E, 0x7D, 0x11, 0x13 must be replaced with 0x7D, and be followed by that byte's value XOR'd with 0x20. Since our packet has a 0x13 at index 6, and a 0x7E at index 17, we will have to do two escapes. After escaping those two bytes, our packet will look like:

0	1	2	3	4	5	6	7
0x7E	0x00	0x0F	0x10	0x01	0x00	0x7D	0x33

8	9	10	11	12	13	14
0xA2	0x00	0x40	0x4B	0x22	0x77	0xFF

15	16	17	18	19	20
0xFE	0x00	0x00	0x7D	0x5E	0xAA

13. Lastly, all that is left to send the packet is to transmit bytes to the XBee module over the serial line

III Implementation

Since a link is no good unless there is something on both ends to process the data it carries, we'll have to implement the XBee packet processing on both the remote and local ends. In this case, the remote end is a Propeller Chip, and the local end is an Apple MacBook computer. Since they both use different programming languages, and have different functionality, we'll have to create send an receive functionality twice. The Propeller Chip is a 32-bit 8 core microcontroller, has no operating system, and is programmed in a quasi object-orientated language called Spin. The MacBook is programmed with the NeXTstep API, which was acquired by Apple Computer and renamed Cocoa. Cocoa is programmed in Objectective-C, and plain ANSI C.

III.1 Propeller Chip

Because the Propeller Chip is muli-cored, it is actually quite easy to implement XBee send and receive functionality.

Receive Function

```
PUB receivePacket(_escaped) | char, index, length, checksum
  'clear lenght and checksum to 0
  length~
  checksum~
  'receive bytes until we get a 0x7E
  repeat until (char := uarts.rx(port)) == $7E
  'get a byte, escape it if needed, and make it the first length
  'word by shifting it 8 bits to the left
  char := uarts.rxtime(port, 1)
  if char == $7D AND _escaped
    length |= (uarts.rx(port) ^ $20) << 8
  else
    length |= char << 8
  'do the same, but don't shift this time, since the next byte
  'is the lowsest signifigant byte of the length word
  char := uarts.rxtime(port, 1)
  if char == $7D AND _escaped
    length |= (uarts.rx(port) ^ $20)
```

```
else
  length |= char
'clear index to 0
index~
'repeat length times
repeat while index < length + 1
  char := uarts.rx(port)
  'need to un-escape it
  '0x7D indicates that the next byte is the actual byte
  'we ignore the 0x7D and use the next byte
  if char == $7D
    'XOR the char with 0x20
    receiveBuffer[index] := uarts.rx(port) ^ $20
  'not an escaped byte, take it as is
  else
    receiveBuffer[index] := char
  'calculate checksun with every iteration of the loop
  'advance the index variable by 1
  checksum += receiveBuffer[index++]
'AND checksum with 0xFF do keep only the lower 8 bits
checksum &= $FF
'if we have a valid packet, all the bytes after lenght, including
'the last byte, checksum, added together equal 0xFF
if checksum <> $FF
  'checksum didn't check out, return -1 to the caller
  return -1
'so far, this receive method only accounts for packets
'sent from other XBee modules, not acknowlege packets
'or various status packets
case receiveBuffer[0]
  'if the first byte after length is 0x90, we are dealing
  'with received packet from another XBee module
  'subtract 12 from length, since the user is only concerned with
  'the length of the meaningful data in the packet, not addresses
  'and such, since they are static, and always the same length
  $90:  rxLength := length-12
        'shift then AND bytes 1..4 to get the upper 32 bit address
        'of the sending module
```

```
          rxRemoteAddressUpper32 := receiveBuffer[1] << 24
                                    | receiveBuffer[2] << 16
                                    | receiveBuffer[3] << 8
                                    | receiveBuffer[4]
          'shift then AND bytes 5..8 to get the lower 32 bit address
          'of the sending module
          rxRemoteAddressLower32 := receiveBuffer[5] << 24
                                    | receiveBuffer[6] << 16
                                    | receiveBuffer[7] << 8
                                    | receiveBuffer[8]
          'shift then AND bytes 10..11 to get the 16 bit network address
          rxNetworkAddress16 := receiveBuffer[10] << 8
                                | receiveBuffer[11]
          'move the important data from receiveBuffer[] to rxData[]
          'so the caller can access it
          bytemove(@rxData, @receiveBuffer[12], rxLength)
          'return 0x90 to the caller, so it knows an rx packet was
          'received
          return $90
```

Send Function

```
pub apiArray(_64BitDestinationAddressUpper, _64BitDestinationAddressLower, _16BitN
  'clear the index
  ptr := 0
  'add 0x7E to byte 0 of array
  dataSet[ptr++] := $7E
  '_arraySize is set by caller, we have to add 14 to it
  'to account for the added bytes API mode requires
  Length := 14 + _arraySize
  'add MSB of length to array
  'add LSB of length to array
  dataSet[ptr++] := Length >> 8
  dataSet[ptr++] := Length
  'add 0x10 to indicade a TX request API packet
  dataSet[ptr++] := $10
  'add frame id, value passed in but isn't important
  dataSet[ptr++] := _FrameID
```

```
'the remote 64 bit address is passed by caller
'in two longs, we need to split up those 2 longs
'into 8 bytes and add them to the array
dataSet[ptr++] := _64BitDestinationAddressUpper >>24
dataSet[ptr++] := _64BitDestinationAddressUpper >>16
dataSet[ptr++] := _64BitDestinationAddressUpper >>8
dataSet[ptr++] := _64BitDestinationAddressUpper
dataSet[ptr++] := _64BitDestinationAddressLower >>24
dataSet[ptr++] := _64BitDestinationAddressLower >>16
dataSet[ptr++] := _64BitDestinationAddressLower >>8
dataSet[ptr++] := _64BitDestinationAddressLower
'the 16 bit network address is passed by caller
'as a word, we need to split it into 2 bytes
dataSet[ptr++] := _16BitNetworkAddress          >>8
dataSet[ptr++] := _16BitNetworkAddress
'unimportant, but the XBee expects these two bytes
dataSet[ptr++] := $00
dataSet[ptr++] := $00
'the caller passed us the address to a byte array
'that is in HUB memory, the caller also passed
'in _arraySize, so we know how many bytes from that
'array to read. Jut loop until we have read all the
'bytes and written them into our array
repeat sourceArrayPtr from 0 to _arraySize - 1
  dataSet[ptr++] := byte[_arrayAddress++]
'start with checksum equals 0xFF, then we subtract all
'the bytes we encounter in our outgoing packet until
'we reach the end
checkSum := $FF
Repeat chars from 3 to ptr-1
  checkSum := checkSum - dataSet[chars]
'add our calculated checksum to the end off the array
dataSet[ptr] := checkSum
'if in escaped mode, loop through our outgoing
'array and escape any characters as necessary
'send them on the fly to the XBEee module
'there's no need to store them in memory first
if (_escaped)
```

```
    tx(dataSet[0])
    Repeat chars from 1 to ptr
      if (dataSet[chars] == $7E OR
          dataSet[chars] == $7D OR
          dataSet[chars] == $11 or
          dataSet[chars] == $13)
        tx($7D)
        tx(dataSet[chars] ^ $20)
      else
        tx(dataSet[chars])
  'if not escaped mode, send the bytes as they are
  else
    Repeat chars from 0 to ptr
      tx(dataSet[chars])
```

III.2 Computer

Due to the nature of event driven Objective-c, the computer code is a bit fragmented and spans many more lines and several files. It's a bit out of scope to print it inline in this paper, but source will be available for download.

IV Conclusions

Implementing a protocol can seem like a daunting challenge at first. The trick is it take it step by step. Add functionality incrementally. The code in this paper stared off very simple, if perhaps a little too hard-coded. This code is actually not even complete, and probably never will be. To implement all the features XBee modules would take a lot of time. But, due to the modular nature of the base code shown here, the added effort is logarithmic, not linear. The basic send and receive functions documented here would be the basis for any added features, such as reading and setting parameters of the XBee module on the fly.

One important lesson learned during the process of building this code is to *write the functions in natural language before writing any actual code.* It's one of those things you always hear you should do, but it takes experience to learn it is something you *need* to do.