Deciphering TinyOS Serial Packets

Octave Tech Brief #5-01

Jeff Thorn, Director of Product Development

March 10, 2005

Introduction

One of the first things a developer wants to do once they get their new mote kit up and running is figure out how to use it with their own applications. There is ample documentation on the TinyOS website about programming the motes, but figuring out how to get that data to your back end applications and what it means can be rather difficult. This document serves as a beginners guide to deciphering TinyOS serial packets. It assumes the reader has been successful in reading the raw data packet from their mote base station (though their serial port connected to a MIB510 Serial Interface board with a mic2 node 0, for example). It begins with some general information about the raw data packet. It then discusses the detailed packet information for a TinyOS message and its payload data. Sample code that shows conversion to engineering units is provided in the Appendix.

The equipment used in the preparation of this document was a Crossbow MICA2 mote kit preinstalled with the Surge / Surge_Reliable TinyOS application. The development environment used in testing was C# .NET, but the information herein is primarily platform independent. The detail within this Tech Brief is the result of extensive review of source code, particularly the Surge-View java application and xlisten C application. Wherever possible, references to source code files are made. In addition, the TinyOS user community and Crossbow engineering and technical support have been, and continue to be, a valuable resource.

General Information

Some high level points that are critical to understanding the makeup of the serial data packet are listed below.

- A TinyOS data packet has a maximum length of 255 bytes.
- The raw data packet is wrapped on both ends by a frame synchronization byte of **`0x7E`**. This is used to detect the start and end of a packet from the stream.
- The raw data packet uses an escape byte of **`0x7D`**. This is needed in case a byte of payload data is the same as a reserved byte code, such as the frame synch byte **`0x7E`**. In such a case, the payload data will be preceded by the escape byte and the payload data itself will be exclusively OR'ed with **`0x20`**. For example, a payload data byte of **`0x7E`** would appear in the data packet as **`0x7D 0x5E`**.
- On an XP machine, multiple byte values are byte-swapped in the data stream. For example, the 2 byte UART Address field (`0x007E`) will appear as `7E 00` in the byte stream. (Note: the .NET `BitConverter.ToUInt16()` handles this conversion correctly).

Raw Data Packet

The following diagram and table describes the raw data packet (see \tools\java\net\tinyos\packet\Packetizer.java):

SYNC_BYTE	Packet Type	Payload Data	SYNC_BYTE
0	1	2...n-1	n

Byte #	Field	Description
0	Packet frame synch byte	Always **0x7E**
1	Packet Type	There are 5 known packet types: • P_PACKET_NO_ACK (0x42) - User packet with no ACK required. • P_PACKET_ACK (0x41) – User packet. ACK required. Includes a prefix byte. Receiver must send a P_ACK response with prefix byte as contents. • P_ACK (0x40) – The ACK response to a P_PACKET_ACK packet. Includes the prefix byte as its contents. • P_UNKNOWN (0xFF) – An unknown packet type.
2…n-1	Payload Data	In most cases will be a TinyOS Message of varying length, which is described below.
n	SYNC_BYTE	Always 0x7E

TinyOS Message

The payload data will typically be a type of TinyOS message, as defined by the struct TOS_Msg in the file \tos\types\AM.h. This data structure is defined as follows:

```
typedef struct TOS_Msg
{
  /* The following fields are transmitted/received on the radio. */
  uint16_t addr;
  uint8_t type;
  uint8_t group;
  uint8_t length;
  int8_t data[TOSH_DATA_LENGTH];
  uint16_t crc;

  /* The following fields are not actually transmitted or received
   * on the radio! They are used for internal accounting only.
   * The reason they are in this structure is that the AM interface
   * requires them to be part of the TOS_Msg that is passed to
   * send/receive operations.
   */
  uint16_t strength;
  uint8_t ack;
  uint16_t time;
  uint8_t sendSecurityMode;
  uint8_t receiveSecurityMode;
} TOS_Msg;
```

The TOS_Msg data packet is described in the following diagram and table:

Address		Message Type	Group ID	Data Length	Data	CRC	
0	1	2	3	4	5...n-2	n-1	n

Byte #	Field	Description
0 - 1	Message Address	One of 3 possible value types: • Broadcast Address (0xFFFF) – message to all nodes. • UART Address (0x007e)– message from a node to the gateway serial port. All incoming messages will have this address. • Node Address – the unique ID of a node to receive message.
2	Message Type	Active Message (AM) unique identifier for the type of message it is. Typically each application will have its own message type. Examples include: • AMTYPE_XUART = 0x00 • AMTYPE_MHOP_DEBUG = 0x03 • AMTYPE_SURGE_MSG = 0x11 • AMTYPE_XSENSOR = 0x32 • AMTYPE_XMULTIHOP = 0x33 • AMTYPE_MHOP_MSG = 0xFA
3	Group ID	Unique identified for the group of motes participating in the network. The default value is 125 (0x7d). Only motes with the same group id will talk to each other.
4	Data Length	The length (*l*) in bytes of the data payload. This does not include the CRC or frame synch bytes.
5…n-2	Payload data	The actual message content. The data resides at byte 5 through byte 5 plus the length of the data (*l* from above). The data will be specific to the message type. Specific message types are discussed in the next section.
n-1, n	CRC	Two byte code that ensures the integrity of the message. The CRC includes the Packet Type plus the entire unescaped TinyOS message. A discussion on how the CRC is computed is included in the Appendix.

Multihop Message

The payload data inside a TinyOS message is raw data specific to an application embedded with the mote device. In many cases, particularly applications that utilize ad-hoc mesh networking, the application will utilize the Multihop message protocol. The definition of the Multihop message is shown below. The format of the Multihop message is shown in the following diagram and table.

```
/* From /contrib/xbow/toslib/ReliableRoute/MultiHop.h – (one of a few implementations) */
typedef struct MultihopMsg {
  uint16_t sourceaddr;
  uint16_t originaddr;
  int16_t seqno;
  uint8_t hopcount;
  uint8_t data[(TOSH_DATA_LENGTH - 7)];
}
```

Source Address		Origin Address		Sequence Number		Hop Count	Application Data
0	1	2	3	4	5	6	7...n

Byte #	Field	Description
0,1	Source Address	The address of the forwarding node.
2,3	Origin Address	The address of the node that originated the packet.
4,5	Sequence Number	Used for determining route and calculating missed packets
6	Hop Count	Used for calculating route. Number of nodes traversed.
7…n	Application Data	The specific application data. Such as SurgeMsg (see below).

Surge Message

The Application Data inside a Multihop message is raw data specific to an actual end user application. The format of the data is determined by the Message Type field in byte 2 of the TinyOS message. The application that comes pre-installed on the motes from Crossbow is called Surge_Reliable, which has a message type of AMTYPE_SURGE_MSG (0x11). The data format for the Surge_Reliable application is defined in the SurgeMsg struct declared in the Surge.h header file. The format of that message is given below.

```
/* From Surge.h  in contrib/xbow/apps/Surge_Reliable   */
typedef struct SurgeMsg {
  uint8_t type;
  uint16_t reading;
  uint16_t parentaddr;
  uint32_t seq_no;
  uint8_t light;
  uint8_t temp;
  uint8_t magx;
  uint8_t magy;
  uint8_t accelx;
  uint8_t accely;
}
```

The SurgeMsg fields are described in the following diagram and table. See the appendix for information on converting to engineering units.

Type	Reading		Parent Addr		Sequence Number				Light	Temp	Mag X	Mag Y	Accel X	Accel Y
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Byte #	Field	Description
0	Type	The type of message that indicates the action. Known values are: • SURGE_TYPE_SENSORREADING (0x00) – The message contains sensor data. • SURGE_TYPE_ROOTBEACON (0x01) - • SURGE_TYPE_SETRATE (0x02) – Changes the rate a mote will send data. • SURGE_TYPE_SLEEP (0x03) – Puts the mote to sleep.

		• SURGE_TYPE_WAKEUP (0x04) – Wakes mote. • SURGE_TYPE_FOCUS (0x05) – Causes mote to chirp. • SURGE_TYPE_UNFOCUS (0x06) – Returns mote to normal (unfocused mode).
1-2	Reading	Does not appear to be used.
3-4	Parent Address	The address of the Parent Node.
5-8	Sequence Number	The upper 9 bits represent the battery voltage. The remaining bits count the number of packets sent since the application was last reset.
9	Light	The raw light sensor reading.
10	Temp	The raw thermistor reading.
11	Mag X	The raw sensor reading for the x-axis magnetometer.
12	Mag Y	The raw sensor reading for the y-axis magnetometer.
13	Accel X	The raw sensor reading for the x-axis accelerometer.
14	Accel Y	The raw sensor reading for the x-axis accelerometer.

Sample Data Packet

Tying it all together, a sample Surge data packet is deconstructed below.

```
Raw Data Packet:
7E 42 7D 5E 00 11 7D 5D 16 00 00 02 00 9D 00 00 00 00 00 00 00 6F 00 80 DB F9 7D 5E FF FF
01 C8 EC D9 7E
```

The raw data packet from the serial port is shown above. Below is the TOS Message after stripping the frame sync bytes and un-escaping the data.

```
TOS Message:
7E 00 11 7D 16 00 00 02 00 9D 00 00 00 00 00 00 00 6F 00 80 DB F9 7E FF FF 01 C8 EC D9
```

The table below shows the meaning of this byte sequence.

Bytes	Description
7E 00	The UART Serial address – 126 (0x00 7E)
11	The message type. Surge message – 17 (0x11)
7D	Group ID. Crossbow mote default is 125 (0x7D)
16	Data length – 22 (0x16)
00 00	The address of the last forwarding node. (node 0)
02 00	The origin address. (node 2)
9D 00	The sequence number – 157 (0x00 9D)
00	The hop count
00	Surge message type (00 – sensor reading)
00 00	Reading – not used
00 00	Parent Node address
6F 00 80 DB	Battery voltage (<< 9 bits), surge sequence no (>>23 bits)
F9	Raw light value
7E	Raw temp value
FF	Raw mag x value
FF	Raw mag y value
01	Raw accel x value
C8	Raw accel y value
EC D9	CRC

Additional Resources

Be sure to visit the Octave Technology website (www.octavetech.com) regularly to check for new Tech Briefs. The list below contains additional links for more information on TinyOS and wireless sensor networks.

- TinyOS Website – http://www.tinyos.net
- TinyOS User Group - http://www.tinyos.net/support.html#lists
- TinyOS Documentation - http://www.tinyos.net/tinyos-1.x/doc/
- TinyOS Sourceforge Project - http://sourceforge.net/projects/tinyos/
- Crossbow Website - http://www.xbow.com/
- Crossbow FAQ - http://xbow.custhelp.com/cgi-bin/xbow.cfg/php/enduser/std_alp.php
- Crossbow MPR/MIB User Manual - http://www.xbow.com/Support/Support_pdf_files/MPR-MIB_Series_User_Manual_7430-0021-06_A.pdf

About Octave Technology

Octave Technology develops innovative software solutions utilizing wireless sensor networks and Auto-ID technologies. Octave specializes in bringing emerging technologies to market by putting new products in the hands of the people who will benefit from them most – the end user. Using sensor and Auto-ID technologies, Octave streamlines inefficient manual and paper-based business processes. Through the research, development and deployment of enterprise software and middleware solutions that extend important data beyond the desktop, Octave Technology allows virtually any industry or organization - maintenance, manufacturing, supply chain, consumer products - to quickly deploy and benefit from the integration of Auto-ID technologies.

Octave Technology is available for consultation regarding wireless sensor network and Auto ID technology pilot programs. For more information, contact info@octavetech.com.

For comments related to this document please contact Octave Technology's Director of Product Development, Jeff Thorn (jthorn@octavetech.com).

Appendix

Sample code for converting raw sensor values to engineering units is shown below. The code is primary C#, but can easily be applied to Java or straight C/C++. Additional resources for finding more calculations are:

- Xlisten: /xlisten/xconvert.c
- Xlisten: /amtypes/surge.c (Available in version 1.16 or higher. Note that version 1.13 is shipped w/ XBow mote kits).

Calculating CRC

The following code is used to compute the CRC of a data packet:

```
public static int calcByte(int crc, int b)
{
    crc = crc ^ (int)b << 8;

    for (int i = 0; i < 8; i++)
    {
        if ((crc & 0x8000) == 0x8000)
            crc = crc << 1 ^ 0x1021;
        else
            crc = crc << 1;
    }

    return crc & 0xffff;
}

public static int calc(byte[] packet, int index, int count)
{
    int crc = 0;
    int i;

    while (count > 0)
    {
        crc = calcByte(crc, packet[index++]);
        count--;
    }

    return crc;
}
```

Calculating Battery Voltage in Engineering Units

```
/// <summary>
/// Calculates the voltage. First the Vref value must be
/// computed using the upper 9 bits of the _surgeSeqno
/// field. The voltage is calculated using the following
/// formula (courtesy of Martin Turon & Crossbow):
///
///   BV = RV*ADC_FS/data
///         where:
///           BV = Battery Voltage
///               ADC_FS = 1023
///               RV = Voltage Reference for mica2 (1.223 volts)
///               data = data from the adc measurement of channel 1
///               BV (volts) = 1252.352/data
///               BV (mv) = 1252352/data
///
/// See also:  /xlisten/xconvert.c  line: 52
///
/// The value for _surgeSeqno is bytes 5-8 from the SurgeMsg.
///
/// </summary>
/// <returns>The voltage of the battery in millivolts as a float.</returns>
float CalculateVoltage()
{
        float vref = (_surgeSeqno & 0xFF800000) >> 23;

        return 1252352 / vref;
}
```

Calculating Temperature in Engineering Units

```
/// <summary>
/// Calculates the temperature, in degrees Celsius of the:
///
///            Panasonic ERT-J1VR103J Thermistor
///
///     The formula in the MTS Series User Manual is:
///            1/T(K) = a + b × ln(Rthr) + c × [ln(Rthr)]^3
///
///     where:
///                    Rthr = R1(ADC_FS-ADC)/ADC
///                    a = 0.00130705
///                    b = 0.000214381
///                    c = 0.000000093
///                    R1 = 10 kOhm
///                    ADC_FS = 1023
///                    ADC = output value from mote's ADC measurement.
///
///     Note that an adjusted temp (adjTemp) is used because the
///     actual temp reading is a 10 bit value that has been compressed
///     into a single byte. As such it must be shifted left 2 bit places.
///
///     The value for _temp is the raw sensor reading from the SurgeMsg.
///
/// </summary>
/// <returns>Temperature in Degrees Celsius as a float.</returns>
float CalculateTempC()
{
        int adjTemp = _temp << 2;

        // prevent divide by zero.
        if(adjTemp == 0)
                return 0F;

        float temperature, a, b, c, Rthr;
        a  = 0.001307050F;
        b  = 0.000214381F;
        c  = 0.000000093F;
        Rthr = 10000 * (1023 - adjTemp) / adjTemp;

        temperature = 1 / (float)(a + b * Math.Log(Rthr) + c *
                                  Math.Pow(Math.Log(Rthr),3));
        temperature -= 273.15F;   // Convert from Kelvin to Celcius

        return temperature;
}

/// <summary>
/// Uses normal Celsius to Fahrenheit conversion formula:
///
///            Fahrenheit = Celsius * 9/5 + 32
///
/// </summary>
/// <returns>Temperature in degree F as a float.</returns>
float CalculateTempF()
{
        return (CalculateTempC() * 9 / 5) + 32;
}
```

Calculating Light in Engineering Units

```
/// <summary>
/// Calculates the light value for the:
///
///            Clairex CL94L light sensor
///
/// Performs simple ADC conversion with the light value
/// as the reference.
///
///    See also:
///            /xlisten/amtypes/surge.c      line: 54
///            /xlisten/xconvert.c           line: 163
/// </summary>
/// <returns>Voltage of ADC channel as an unsigned integer in mV</returns>
UInt16 CalculateLight()
{
       return (UInt16)( CalculateVoltage() * _light / 1024);
}
```