

Serial Communication

TEP: 113
Group: Core Working Group
Type: Documentary
Status: Draft
TinyOS-Version: 2.x
Author: Ben Greenstein and Philip Levis
Draft-Created: 11-Jul-2005
Draft-Version: 1.9
Draft-Modified: 2007-02-06
Draft-Discuss: TinyOS Developer List <tinyos-devel at mail.millennium.berkeley.edu>

Note

This memo documents a part of TinyOS for the TinyOS Community, and requests discussion and suggestions for improvements. Distribution of this memo is unlimited. This memo is in full compliance with TEP 1.

Abstract

This memo describes the structure and standard implementation of the TinyOS 2.x serial communication system for mote-to-PC data exchange. The system is broken into three levels (encoding, framing, and dispatch) to allow easy experimentation and replacement. It can also handle multiple packet formats: unlike 1.x, 2.x serial packets are not bound to the mote's radio packet format. Additionally, one of the supported packet formats is platform independent, so PC-side applications can communicate with arbitrary motes.

1. Introduction

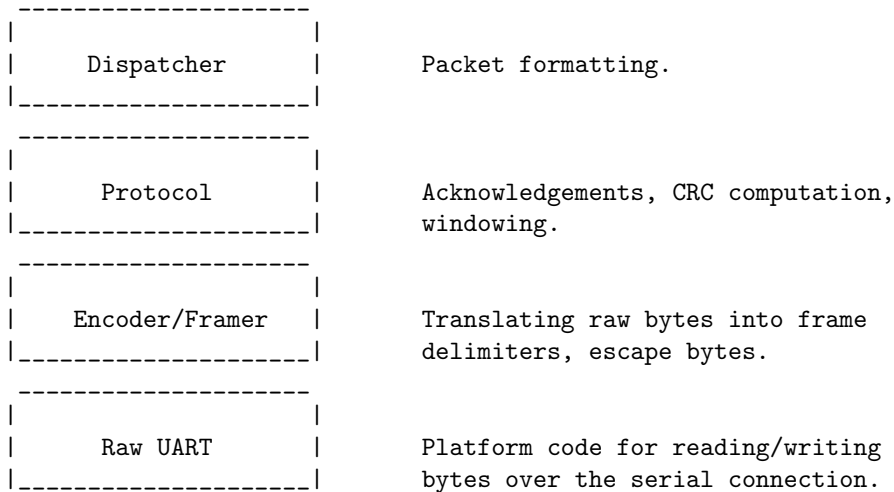
Users need to read data out of a TinyOS network. The most common approach is to attach a mote to a PC or laptop with a wired connection. While the interface on the PC side can vary from a serial cable to a USB device to IP, the mote generally talks to a serial port (UART). In TinyOS 1.x, the UART packet format is platform-specific, pushing a good deal of complexity into the protocol and PC-side tools in order to discover and properly handle platform diversity. TinyOS 2.0 introduces the notion of packet format dispatch, so a mote can support multiple UART packet formats simultaneously. This allows transparent bridging (e.g., an 802.15.4 base station) to exist in parallel with platform-independent communication, which simplifies the PC toolchain. This memo documents the protocols and structure of the TinyOS 2.x serial communication stack.

2. Serial Stack Structure

The TinyOS 2.x serial communication stack is broken up into four functional components. From bottom to top, they are

- o the raw UART,
- o the encoder/framer,
- o the protocol,
- o and the dispatcher.

Structurally, they look like this:



The bottom three provide a byte-level interface: only the Dispatcher provides a packet-level interface. The top three are all platform-independent: only the UART is platform-specific code.

The lowest level of the stack is the raw UART. This HIL component provides functionality for configuring the UART (speed, stop bytes, etc.), sending/receiving bytes, and flushing the UART.

The Encoder/Framer sits above the raw UART. This component translates raw data bytes into packet bytes using a serial protocol's encoding. The Encoder/Framer assumes that a protocol's encoding has two kinds of bytes: delimiters and data bytes, and signals each in separate events to the component above.

The Protocol component handles data and delimiter byte events. It is responsible for reading in and sending all protocol control packets. If the Protocol component starts receiving a data packet, it signals to the Dispatcher that a packet has started and signals the data bytes. When the data packet completes, the Protocol signals to the Dispatcher that the packet is complete and whether it passed the protocol-level CRC.

The Dispatcher component handles data packet bytes and delimiters. It is responsible for reading data bytes into a `message_t` and signaling packet reception to components above it. The dispatcher can support multiple packet formats. Based on how `message_t` works (see TEP 111[[tep111](#)]), this boils down to knowing where in a `message_t` a particular packet format begins (based on its header size). Section 3.4 describes how the default TinyOS 2.x implementation, `SerialDispatcherC` does this.

3. The 2.x Serial Stack Implementation

Section 2 describes the basic structure of the TinyOS 2.x serial stack. This section describes its actual implementation, including `SerialActiveMessageC`, which sits on top of the Dispatcher. All of the components except for `UartC` are part of the serial library that lives in `tos/lib/serial`.

3.1 Raw UART: UartC

The UART HIL[TEP2] is `UartC`, which provides a byte-level interface to the underlying serial communication. It provides the `SerialByteComm` interface:

```
interface SerialByteComm {
    async command error_t put(uint8_t data);
    async event void get(uint8_t data);
    async event void putDone();
}
```

Alternatively, `UartC` may provide the `UartStream` multi-byte level interface. See the Low-Level I/O TEP [TEP117] for details.

Additionally, `UartC` provides a split-phase interface to signal when the UART is idle. There are situations (such as when powering down the usart, when switching from TX to RX on a radio with a UART data line, etc.) when we need explicit information that the data sent over the UART has actually been transmitted in full. The problem is that on MCUs that double-buffer UART communication (such as the msp430), a `putDone` event signifies that the UART is ready to accept another byte, but NOT that the UART is idle.

```
interface SerialFlush {
    command void flush();
    event void flushDone();
}
```

It may provide additional interfaces for configuring the serial port. This TEP does not consider this topic.

3.2 Encoder/Framer: HdlcTranslateC

`HdlcTranslateC` is the serial encoder/framer. It uses the `SerialByteComm` interface and provides the `SerialFrameComm` interface:

```
interface SerialFrameComm {
    async command error_t putDelimiter();
    async command error_t putData(uint8_t data);
    async command void resetSend();
    async command void resetReceive();
    async event void delimiterReceived();
    async event void dataReceived(uint8_t data);
    async event void putDone();
}
```

As its name suggests, it uses the same encoding as the HDLC[HDLC] protocol. `0x7e` is reserved as a frame delimiter byte, and `0x7d` is reserved as an escape byte. `HdlcTranslateC` maintains ten bits of state. The receive and send paths each have one bit to store whether they are using an escape byte, and the transmit path has a byte for when it sends an escaped byte.

When `HdlcTranslateC` receives a delimiter byte, it signals `delimiterReceived()`. When `HdlcTranslateC` receives an escape byte, it sets the `receiveEscape` flag to true. When it receives any other byte, it tests to see if the `receiveEscape` flag is set; if so, it XORs the data byte with `0x20` and clears the flag. It signals `dataReceived()` with the byte. The most common use of escape byte is to transmit data bytes corresponding to the delimiter byte or escape byte. For example, `0x7e` becomes `0x7d 0x5e`.

`HdlcTranslateC` performs similar actions on the transmit side. When told to transmit the delimiter or escape byte as a data byte, it sets the `transmitEscape` flag to true, stores the data byte XOR `0x20`, and sends an escape byte. When the escape byte is sent, it sends the stored data byte.

3.3 Protocol: SerialP

The SerialP component implements the serial protocol using PPP/HDLC- like framing (See RFC 1662[RFC1662]). Type dispatch and buffer management are left to higher layers in the serial stack. The protocol is currently stop-and-wait in the host-to-mote direction and best effort in the mote-to-host direction.

SerialP provides two byte-level interfaces to the upper layer for sending and receiving packets, respectively called `SendBytePacket` and `ReceiveBytePacket`.

On the sending side, SerialP is responsible for encapsulation of upper layer packets. An upper layer component such as `SerialDispatcherC` initiates the sending of a packet by calling `startSend()`, passing the first byte to send. SerialP collects subsequent bytes by signalling `nextByte()`. Within the `nextByte` handler or between calls to `nextByte()`, the upper layer should indicate the end-of-packet by calling `completeSend()`. If `completeSend` is called from within a `nextByte()` handler, SerialP will ignore the return of the call to `nextByte()`.

```
interface SendBytePacket {
    async command error_t startSend(uint8_t first_byte);
    async command error_t completeSend();
    async event uint8_t nextByte();
    async event void sendCompleted(error_t error);
}
```

SerialP maintains a small window of bytes that have been received by the upper layer and not yet sent to the UART. Depending on the timing requirements of the underlying UART, the size of this window can be changed. SerialP uses repeated calls to `nextByte()` to keep this window filled.

SerialP uses `SerialFrameComm` to send a delimiter between frames, a serial-level type field, the bytes of the packet, and a two-byte frame CRC. For mote-to-host gap detection and link reliability, a sequence number may also be sent (not activated in the default implementation).

After sending an entire frame and receiving the last `putDone()` event from below, SerialP signals `sendCompleted()` to indicate the success or failure of a requested transmission.

Packet reception is also managed by SerialP and the interface provided to the upper layer is `ReceiveBytePacket`:

```
interface ReceiveBytePacket {
    async event error_t startPacket();
    async event void byteReceived(uint8_t b);
    async event void endPacket(error_t result);
}
```

Upon receiving an interframe delimiter and a new frame's header, SerialP signals the upper layer indicating that a packet is arriving. For each byte received, SerialP signals `byteReceived()`. Once SerialP receives the complete frame it signals `endPacket` with a value of `SUCCESS`. If instead it loses sync during reception it signals `endPacket` with `FAIL`.

SerialP acknowledges frames it receives. Acknowledgements have a higher priority than data transmissions and consequently, data frames may be slightly delayed. However, acknowledgement information is stored in a queue separate from the data buffer, so a data packet to be transmitted may begin spooling into SerialP while SerialP is actively sending an acknowledgement.

3.4 Dispatcher: SerialDispatcherC

`SerialDispatcherC` handles the data packets that the Protocol component receives. It uses the `SendBytePacket` and `ReceiveBytePacket` interfaces, and provides parameterized `Send` and `Receive` interfaces. The parameter in the `Send` and `Receive` interfaces (`uart_id_t`) determines the packet format contained in the `message_t`.

SerialDispatcherC places a one-byte header, the packet format identifier, on the packets sent and received through SerialP. SerialDispatcherC uses a parameterized SerialPacketInfo interface to be able to handle various packet formats:

```
interface SerialPacketInfo {
    async command uint8_t offset();
    async command uint8_t dataLinkLength(message_t* msg, uint8_t upperLen);
    async command uint8_t upperLength(message_t* msg, uint8_t dataLinkLen);
}
```

When SerialDispatcherC receives the first data byte of a packet from SerialP, it stores it as the packet type and calls `offset()` to determine where in a `message_t` that packet format begins. It then spools data bytes in, filling them into its `message_t` buffer. Similarly, on the send side, it first sends the type byte and spools out data bytes starting from the index denoted by the call to `offset()`. SerialDispatcherC uses the two length commands, `dataLinkLength()` and `upperLength()`, to translate between the two notions of packet length: above, length refers to the payload excluding header, while below it refers to the payload plus header.

A component that provides communication over the serial port with `uart_id_t U` MUST wire a component implementing `SerialPacketInfo` to `SerialDispatcherC` with `uart_id_t U`. The file `Serial.h` contains reserved `uart_id_t`'s for supported packet formats. Currently, only platform independent active messages (`TOS_SERIAL_ACTIVE_MESSAGE_ID`, described in Section 3.5), 802.15.4 active messages (`TOS_SERIAL_802_15_4_ID`), mica2 CC1000 packets (`TOS_SERIAL_CC1000_ID`) and the error code `TOS_SERIAL_UNKNOWN_ID` are reserved. New packet formats MUST NOT reuse any reserved identifiers.

3.5 SerialActiveMessageC

`SerialActiveMessageC` is a platform-independent active message layer that operates on top of the serial communication stack. `SerialActiveMessageC` is a configuration that wires `SerialActiveMessageP` to `SerialDispatcherC` with `uart_id_t TOS_SERIAL_ACTIVE_MESSAGE_ID` and wires `SerialPacketInfoActiveMessageP` to `SerialDispatcherC` with `uart_id_t TOS_SERIAL_ACTIVE_MESSAGE_ID`:

```
includes Serial;
configuration SerialActiveMessageC {
    provides {
        interface Init;
        interface AMSend[am_id_t id];
        interface Receive[am_id_t id];
        interface Packet;
        interface AMPacket;
    }
    uses interface Leds;
}
implementation {
    components new SerialActiveMessageP() as AM, SerialDispatcherC;
    components SerialPacketInfoActiveMessageP as Info;

    Init = SerialDispatcherC;
    Leds = SerialDispatcherC;

    AMSend = AM;
    Receive = AM;
    Packet = AM;
    AMPacket = AM;
}
```


of fair-share queueing. SerialAMReceiverC provides the virtualized abstraction for reception. These abstractions are very similar to TinyOS's radio abstractions, namely, AMSenderC and AMReceiverC. See Section 4 of TEP 116[TEP116] for more information. Unlike the services in the TEP 116, the serial component virtualizations provide no snooping capabilities.

5. Author's Address

Philip Levis
358 Gates
Computer Science Laboratory
Stanford University
Stanford, CA 94305

phone - +1 650 725 9046
email - pal@cs.stanford.edu

Ben Greenstein
Intel Research Seattle
1100 NE 45th Street, 6th Floor
Seattle, WA 98105

phone - +1 206 206 545 2501
email - benjamin.m.greenstein@intel.com

6. Citations

[TEP2] TEP 2: Hardware Abstraction Architecture. [tinyos-2.x/doc/txt/tep2.txt](#)

[TEP111] TEP 111: message_t. [tinyos-2.x/doc/txt/tep111.txt](#)

[TEP116] TEP 116: Packet Protocols. [tinyos-2.x/doc/txt/tep116.txt](#)

[TEP117] TEP 117: Low-Level I/O. [tinyos-2.x/doc/txt/tep117.txt](#)

[HDLC] International Organization For Standardization, ISO Standard 3309-1979, "Data communication - High-level data link control procedures - Frame structure", 1979.

[RFC1662] PPP in HDLC-like Framing, Internet Engineering Task Force (IETF), 1994