Packet Protocols

TEP:	116
Group:	Core Working Group
Type:	Documentary
Status:	Final
TinyOS-Version:	> 2.1
Author:	Philip Levis

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

The memo documents the interfaces used by packet protocol components in TinyOS 2.x as well as the structure and implementation of ActiveMessageC, the basic data-link HIL component. It also documents the virtualized active message interfaces AMSenderC and AMReceiverC.

1. Introduction

Sensor nodes are network-centric devices. Much of their software complexity comes from network protocols and their interactions. In TinyOS, the basic network abstraction is an *active message*, a single-hop, unreliable packet. Active messages have a destination address, provide synchronous acknowledgements, and can be of variable length up to a fixed maximum size. They also have a type field, which is essentially a protocol identifier for components built on top of this abstraction.

In TinyOS 1.x, the component GenericComm provides interfaces for transmitting and receiving active messages:

```
configuration GenericComm {
  provides {
    interface StdControl as Control;
    interface SendMsg[uint8_t id];
    interface ReceiveMsg[uint8_t id];
    command uint16_t activity();
  }
  uses {
    event result_t sendDone();
  }
```

```
}
```

This component, while simple, has several issues. First, it has the activity() command, which does not have a single caller in the entire TinyOS tree. This command requires GenericComm to allocate a timer, wasting CPU cycles and RAM.

Second, it does not allow a node to receive packets besides those destined to it. Several network protocols (e.g., MintRoute[1], TAG[2]) take advantage of snooping on these packets for a variety of improvements in efficiency or performance. This has led to the creation of GenericCommPromiscuous, whose Receive interface does not distinguish between packets received that were addressed to the node and packets received that were addressed to other nodes. Choosing one of the two implementations is a global decision across an application. There is a way to enable both reception semantics at the same time for a different protocols, but they require a creative use of default event handlers.

Third, it assumes that components will directly access the packet structure, the accepted approach in TinyOS 1.x. However, directly accessing packet structures introduces unforseen dependencies: a component that names a header field, for example, binds itself to data link layers that have a field with that name. Similarly, components on top of GenericComm directly access the data payload of a packet.

TEP 111 documents the structure of a TinyOS 2.x packet buffer[3]. This TEP documents the interfaces used to access packet buffers, as well as ActiveMessageC, the basic data-link packet communication HIL.

2. Communication interfaces

Packet-level communication has three basic classes of interfaces. *Packet* interfaces are for accessing message fields and payloads. *Send* interfaces are for transmitting packets, and are distinguished by their addressing scheme. The *Receive* interface is for handling packet reception events. Finally, depending on whether the protocol has a dispatch identifier field, the Receive and Send interfaces may be parameterized in order to support multiple higher-level clients.

2.1 Packet interfaces

The basic TinyOS 2.x message buffer type is message_t, which is described in TEP 111. message_t right-justifies data-link headers to the data payload so that higher-level components can pass buffers between different data link layers without having to move data payloads. This means that the data payload of a data link frame is always at a fixed offset of a message_t.

Once protocols layer on top of each other, the data payload for components on top of the data link layer are no longer at a fixed offset. Where a component can put its header or data depends on what headers underlying components introduce. Therefore, in order to be able to find out where it can put its data, it must query the components below it. The Packet interface defines this mechanism:

```
interface Packet {
  command void clear(message_t* msg);
  command uint8_t payloadLength(message_t* msg);
  command void setPayLoadLength(message_t* msg, uint8_t len);
  command uint8_t maxPayloadLength();
```

```
  command void* getPayload(message_t* msg, uint8_t len);
}
```

A component can obtain a pointer to its data region within a packet by calling `getPayload()`. A call to this command includes the length the caller requires. The command `maxPayloadLength` returns the maximum length the payload can be: if the `len` parameter to `getPayload` is greater than the value `maxPayloadLength` would return, `getPayload` MUST return NULL.

A component can set the payload length with `setPayLoadLength`. A component can obtain the size of the data region of packet in use with a call to `payloadLength`. As Send interfaces always include a length parameter in their send call, `setPayLoadLength` is not required for sending, and so is never called in common use cases. Instead, it is a way for queues and other packet buffering components to store the full state of a packet without requiring additional memory allocation.

The distinction between `payloadLength` and `maxPayloadLength` comes from whether the packet is being received or sent. In the receive case, determining the size of the existing data payload is needed; in the send case, a component needs to know how much data it can put in the packet. By definition, the return value of `payloadLength` must be less than or equal to the return value of `maxPayloadLength`.

The Packet interface assumes that headers have a fixed size. It is difficult to return a pointer into the data region when its position will only be known once the header values are bound.

The `clear` command clears out all headers, footers, and metadata for lower layers. For example, calling `clear` on a routing component, such as CollectionSenderC[4]_, will clear out the collection headers and footers. Furthermore, CollectionSenderC will recursively call `clear` on the layer below it, clearing out the link layer headers and footers. Calling `clear` is typically necessary when moving a packet across two link layers. Otherwise, the destination link layer may incorrectly interpret metadata from the source link layer, and, for example, transmit the packet on the wrong RF channel. Because `clear` prepares a packet for a particular link layer, in this example correct code would call the command on the destination link layer, not the source link layer.

Typically, an incoming call to the Packet interface of a protocol has an accompanying outgoing call to the Packet interface of the component below it. The one exception to this is the data link layer. For example, if there is a network that introduces 16-bit sequence numbers to packets, it might look like this:

```
generic module SequenceNumber {
  provides interface Packet;
  uses interface Packet as SubPacket;
}
implementation {
  typedef nx_struct seq_header {
    nx_uint16_t seqNo;
  } seq_header_t;

  enum {
    SEQNO_OFFSET = sizeof(seq_header_t),
  };

  command void Packet.clear(message_t* msg) {
    void* payload = call SubPacket.getPayload(msg, call SubPacket.maxPayloa
```

```
    call SubPacket.clear();
    if (payload != NULL) {
      memset(payload, sizeof(seq_header_t), 0);
    }
  }

  command uint8_t Packet.payloadLength(message_t* msg) {
    return SubPacket.payloadLength(msg) - SEQNO_OFFSET;
  }

  command void Packet.setPayloadLength(message_t* msg, uint8_t len) {
    SubPacket.setPayloadLength(msg, len + SEQNO_OFFSET);
  }

  command uint8_t Packet.maxPayloadLength() {
    return SubPacket.maxPayloadLength(msg) - SEQNO_OFFSET;
  }

  command void* Packet.getPayload(message_t* msg, uint8_t len) {
    uint8_t* payload = call SubPacket.getPayload(msg, len + SEQNO_OFFSET);
    if (payload != NULL) {
      payload += SEQNO_OFFSET;
    }
    return payload;
  }
}
```

The above example is incomplete: it does not include the code for the send path that increments sequence numbers.

In practice, calls to Packet are very efficient even if they pass through many components before reaching the data link layer. nesC's inlining means that in almost all cases there will not actually be any function calls, and since payload position and length calculations all use constant offsets, the compiler generally uses constant folding to generate a fixed offset.

The Packet interface provides access to the one field all packet layers have, the data payload. Communication layers can add additional header and footer fields, and may need to provide access to these fields. If a packet communication component provides access to header and/or footer fields, it MUST do so through an interface. The interface SHOULD have a name of the form *XPacket*, where *X* is a name that describes the communication layer. For example, active message components provide both the Packet interface and the AMPacket interface. The latter has this signature:

```
interface AMPacket {
  command am_addr_t address();
  command am_addr_t destination(message_t* amsg);
  command am_addr_t source(message_t* amsg);
  command void setDestination(message_t* amsg, am_addr_t addr);
  command void setSource(message_t* amsg, am_addr_t addr);
  command bool isForMe(message_t* amsg);
  command am_id_t type(message_t* amsg);
  command void setType(message_t* amsg, am_id_t t);
```

```
  command am_group_t group(message_t* amsg);
  command void setGroup(message_t* amsg, am_group_t grp);
  command am_group_t localGroup();
}
```

The command address() returns the local AM address of the node. AMPacket provides accessors for its four fields, destination, source, type and group. It also provides commands to set these fields, for the same reason that Packet allows a caller to set the payload length. Packet interfaces SHOULD provide accessors and mutators for all of their fields to enable queues and other buffering to store values in a packet buffer. Typically, a component stores these values in the packet buffer itself (where the field is), but when necessary it may use the metadata region of message_t or other locations.

The group field refers to the AM group, a logical network identifier. Link layers will typically only signal reception for packets whose AM group matches the node's, which `localGroup` returns.

2.2 Sending interfaces

There are multiple sending interfaces, corresponding to different addressing modes. For example, address-free protocols, such as collection routing, provide the basic `Send` interface. Active message communication has a destination of an AM address, so it provides the `AMSend` interface. This, for example, is the basic, address-free Send interface:

```
interface Send {
  command error_t send(message_t* msg, uint8_t len);
  command error_t cancel(message_t* msg);
  event void sendDone(message_t* msg, error_t error);

  command uint8_t maxPayloadLength();
  command void* getPayload(message_t* msg, uint8_t len);
}
```

while this is the AMSend interface:

```
interface AMSend {
  command error_t send(am_addr_t addr, message_t* msg, uint8_t len);
  command error_t cancel(message_t* msg);
  event void sendDone(message_t* msg, error_t error);

  command uint8_t maxPayloadLength();
  command void* getPayload(message_t* msg, uint8_t len);
}
```

Sending interfaces MUST include these four commands and one event. The duplication of some of the commands in Packet is solely for ease of use: `maxPayloadLength` and `getPayload` MUST behave identically as `Packet.maxPayloadLength` and `Packet.getPayload`. Their inclusion is so that components do not have to wire to both Packet and the sending interface for basic use cases.

When called with a length that is too long for the underlying maximum transfer unit (MTU), the send command MUST return ESIZE.

The Send and AMSend interfaces have an explicit queue of depth one. A call to send on either of these interfaces MUST return EBUSY if a prior call to send returned SUCCESS but no sendDone event has been signaled yet. More explicitly:

```
if (call Send.send(...) == SUCCESS &&
    call Send.send(...) == SUCCESS) {
   // This block is unreachable.
}
```

Systems that need send queues have two options. They can use a QueueC (found in tos/system) to store pending packet pointers and serialize them onto sending interface, or they can introduce a new sending interface that supports multiple pending transmissions.

The cancel command allows a sender to cancel the current transmission. A call to cancel when there is no pending sendDone event MUST return FAIL. If there is a pending sendDone event and the cancel returns SUCCESS, then the packet layer MUST NOT transmit the packet and MUST signal sendDone with ECANCEL as its error code. If there is a pending sendDone event and cancel returns FAIL, then sendDone MUST occur as if the cancel was not called.

2.3 Receive interface

Receive is the interface for receiving packets. It has this signature:

```
interface Receive {
  event message_t* receive(message_t* msg, void* payload, uint8_t len);
}
```

The receive() event's payload parameter MUST be identical to what a call to the corresponding Packet.getPayload() would return, and the len parameter MUST be identical to the length that a call to Packet.getPayload would return. These parameters are for convenience, as they are commonly used by receive handlers, and their presence removes the need for a call to getPayload(). Unlike Send, Receive does not have a convenience getPayload call, because doing so prevents fan-in. As Receive has only a single event, users of Receive can be wired multiple times.

Receive has a *buffer-swap* policy. The handler of the event MUST return a pointer to a valid message buffer for the signaler to use. This approach enforces an equilibrium between upper and lower packet layers. If an upper layer cannot handle packets as quickly as they are arriving, it still has to return a valid buffer to the lower layer. This buffer could be the msg parameter passed to it: it just returns the buffer it was given without looking at it. Following this policy means that a data-rate mismatch in an upper-level component will be isolated to that component. It will drop packets, but it will not prevent other components from receiving packets. If an upper layer did not have to return a buffer immediately, then when an upper layer cannot handle packets quickly enough it will end up holding all of them, starving lower layers and possibly preventing packet reception.

A *user* of the Receive interface has three basic options when it handles a receive event:

1) Return msg without touching it.
2) Copy some data out of payload and return msg.

3) Store `msg` in its local frame and return a different `message_t*` for the lower layer to use.

These are simple code examples of the three cases:

```
// Case 1
message_t* Receive.receive(message_t* msg, void* payload, uint8_t len) {
  return msg;
}

// Case 2
uint16_t value;
message_t* Receive.receive(message_t* msg, void* payload, uint8_t len) {
  if (len >= sizeof(uint16_t)) {
    nx_uint16_t* nval = (nx_uint16_t*)payload;
    value = *nval;
  }
  return msg;
}

//Case 3
message_t buf;
message_t* ptr = &buf;
message_t* Receive.receive(message_t* msg, void* payload, uint8_t len) {
  message_t* tmp = ptr;
  ptr = msg;
  post processTask();
  return tmp;
}
```

Because of case 3), a lower layer MUST respect the buffer swap semantics and use the pointer returned from `receive`. The pointer passed as a parameter to `receive` MUST NOT be touched, used, or stored after the signaling of `receive`.

2.4 Dispatch

A packet protocol MAY have a dispatch identifier. This generally manifests as the protocol component providing parameterized interfaces (rather than a single interface instance). A dispatch identifier allows multiple services to use a protocol independently. If a protocol provides a dispatch mechanism, then each dispatch identifier SHOULD correspond to a single packet format: if an identifier corresponds to multiple packet formats, then there is no way to disambiguate them. Packets whose internal structure depends on their fields (for example, a packet that has a control field which indicates which optional fields are present) do not pose such problems.

3. HIL: ActiveMessageC

A platform MUST provide ActiveMessageC as a basic HIL to packet-level communication. ActiveMessageC provides a best-effort, single-hop communication abstraction.

Every active message has a 16-bit destination address and an 8-bit type. There is one reserved destination address, AM_BROADCAST_ADDR, which has the value of 0xffff. ActiveMessageC has the following signature:

```
configuration ActiveMessageC {
  provides {
    interface Init;
    interface SplitControl;

    interface AMSend[uint8_t id];
    interface Receive[uint8_t id];
    interface Receive as Snoop[uint8_t id];

    interface Packet;
    interface AMPacket;
    interface PacketAcknowledgements;
  }
}
```

The Receive interface is for packets destined to the node, while the Snoop interface is for packets destined to other nodes. A packet is destined for a node if its destination AM address is either the AM broadcast address or an address associated with the AM stack. Different link layers have different snooping capabilities. The Snoop interface does not assume always-on listening, for example, in the case of a TDMA or RTS/CTS data link layer. By separating out these two interfaces, ActiveMessageC avoids the complications encountered in 1.x with regards to GenericComm vs. GenericCommPromiscuous.

ActiveMessageC is usually just a configuration that has pass-through wiring to a chip-specific HAL active message implementation. The definition of ActiveMessageC is left to the platform for when a node has more than one radio. In this case, the platform decides how to map the basic packet abstraction to the hardware underneath. Approaches include choosing one radio or having some form of address-based dispatch.

4. AM Services: AMSenderC, AMReceiverC, AMSnooperC, AMSnoopingReceiverC

TinyOS 2.x provides four component single-hop communication virtualizations to applications: AMReceiverC, AMSnooperC, AMSnoopingReceiverC, and AMSenderC. Each is a generic component that takes an active message ID as a parameter. These components assume the existence of ActiveMessageC.

4.1 Dispatch: `am_id_t`

Active messages have an 8-bit type field, which allows multiple protocols to all use AM communication without conflicting. Following the guidelines for protocol dispatch identifiers, each am_id_t used in a network SHOULD have a single packet format, so that the am_id_t, combined with the packet contents, are sufficient to determine the exact packet format.

4.2 AMReceiverC

AMReceiverC has the following signature:

```
generic configuration AMReceiverC(am_id_t t) {
  provides{
    interface Receive;
    interface Packet;
    interface AMPacket;
  }
}
```

AMReceiver.Receive.receive is signalled whenever the packet layer receives an active message of the corresponding AM type whose destination address is the local address or the broadcast address. Note that since Receive.receive swaps buffers, a program MUST NOT instantiate two AMReceivers with the same am_id_t and MUST NOT instantiate an AMReceiver and an AMSnoopingReceiver with the same am_id_t.

4.3 AMSnooperC

AMSnooper has an identical signature to AMReceiver:

```
generic configuration AMSnooperC(am_id_t t) {
  provides{
    interface Receive;
    interface Packet;
    interface AMPacket;
  }
}
```

AMSnooper.Receive.receive is signalled whenever the packet layer receives an active message of the corresponding AM type whose destination address is neither to the local address nor the broadcast address. Note that since Receive.receive swaps buffers, a program MUST NOT instantiate two AMSnoopers with the same am_id_t and MUST NOT instantiate an AMSnooper and an AMSnoopingReceiver with the same am_id_t.

4.4 AMSnoopingReceiverC

AMSnoopingReceiverC has an identical signature to AMReceiverC:

```
generic configuration AMSnoopingReceiverC(am_id_t t) {
  provides{
    interface Receive;
    interface Packet;
    interface AMPacket;
  }
}
```

AMSnoopingReceiverC.Receive.receive is signalled whenever the packet layer receives an active message of the corresponding AM type, regardless of destination address. Note that since Receive.receive swaps buffers, a program that instantiates an AMSnoopingReceiverC with a certain am_id_t MUST NOT instantiate another AMSnoopingReceiverC, AMSnooperC, or AMReceiverC with the same am_id_t.

4.5 AMSenderC

AMSenderC has the following signature:

```
generic configuration AMSenderC(am_id_t AMId) {
  provides {
    interface AMSend;
    interface Packet;
    interface AMPacket;
    interface PacketAcknowledgements as Acks;
  }
}
```

Because this is a send virtualization, AMSenderC.AMSend.send returns EBUSY only if there is a send request outstanding on this particular AMSenderC. That is, each AMSenderC has a queue of depth one. The exact order in which pending AMSenderC requests are serviced is undefined, but it MUST be fair, where fair means that each client with outstanding packets receives a reasonable approximation of an equal share of the available transmission bandwidth.

5. Power Management and Local Address

In addition to standard datapath interfaces for sending and receiving packets, an active message layer also has control interfaces.

5.1 Power Management

The communication virtualizations do not support power management. ActiveMessageC provides SplitControl for explicit power control. For packet communication to operate properly, a component in an application has to call ActiveMessageC.SplitControl.start(). The HAL underneath ActiveMessageC MAY employ power management techniques, such as TDMA scheduling or low power listening, when "on."

5.2 Local Active Message Address

An application can change ActiveMessageC's local AM address at runtime. This will change which packets a node receives and the source address it embeds in packets. To change the local AM address at runtime, a component can wire to the component `ActiveMessageAddressC`. This component only changes the AM address of the default radio stack (AMSenderC, etc.); if a radio has multiple stacks those may have other components for changing their addresses in a stack-specific fashion.

5. HAL Requirements

A radio chip *X* MUST have a packet abstraction with the following signature:

```
provides interface Init;
provides interface SplitControl;
provides interface AMSend[am_id_t type];
provides interface Receive[am_id_t type];
```

```
provides interface Receive as Snoop[am_id_t type];
provides interface Packet;
provides interface AMPacket;
provides interface PacketAcknowledgments;
```

The component SHOULD be named *XActiveMessageC*, where *X* is the name of the radio chip. The component MAY have additional interfaces. These interfaces can either be chip-specific or chip-independent.

6. message_t

Active messages are a basic single-hop packet abstraction. Therefore, following TEP 111[3], all data link and active message headers MUST be in the `message_header_t` structure of message_t. This ensures that an active message received from one data link layer (e.g., the radio) can be passed to another data link layer (e.g., the UART) without shifting the data payload. This means that the `message_header_t` must include all data needed for AM fields, which might introduce headers in addition to those of the data link. For example, this is an example structure for a CC2420 (802.15.4) header:

```
typedef nx_struct cc2420_header_t {
  nx_uint8_t length;
  nx_uint16_t fcf;
  nx_uint8_t dsn;
  nx_uint16_t destpan;
  nx_uint16_t dest;
  nx_uint16_t src;
  nx_uint8_t type;
} cc2420_header_t;
```

The first six fields (length through src) are all 802.15.4 headers. The type field, however, has been added to the header structure in order to support AM dispatch.

7. Implementation

The following files in `tinyos-2.x/tos/system` provide reference implementations of the abstractions described in this TEP.

- `AMSenderC.nc`, `AMReceiverC.nc`, `AMSnooperC.nc`, and `AMSnoopingReceiverC.nc` are implementations of virtualized AM services.
- `AMQueueP` provides a send queue of *n* entries for *n* AMSenderC clients, such that each client has a dedicated entry.
- `AMQueueImplP` is the underlying queue implementation, which is reusable for different clients (it is also used in the serial stack[4]).
- `AMQueueEntryP` sits on top of `AMQueueP` and stores the parameters to `AMSend.send` in an outstanding packet with the `AMPacket` interface.

The following files in `tinyos-2.x/tos/interfaces` contain example implementations of packet protocol interfaces:

- `Packet.nc` is the basic interface that almost all packet protocols provide.
- **`Send.nc` is the transmission interface for address-free** protocols.
- **`AMSend.nc` is the transmission interface for AM address** send protocols.
- `AMPacket.nc` is the packet interface for AM-specific fields.

An active messaging implementation for the CC2420 radio chip can be found in `tos/chips/CC2420/CC2420ActiveMessageC.nc`. The micaz platform and telos family have an `ActiveMessageC.nc` which exports the interfaces of `CC2420ActiveMessageC`.

8. Author's Address

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

phone - +1 650 725 9046

9. Citations

[1]The MintRoute protocol. `tinyos-1.x/tos/lib/MintRoute`. Also, A. Woo, T. Tong, and D. Culler. "Taming the Underlying Challenges of Reliable Multihop Routing in Sensor Networks." SenSys 2003.

[2]Tiny AGgregation, one protocol of the TinyDB system. `tinyos-1.x/tos/lib/TinyDB`. Also, S. Madden and M. Franklin and J. Hellerstein and W. Hong. "TinyDB: An Acquisitional Query Processing System for Sensor Networks." Transactions on Database Systems (TODS) 2005.

[3]TEP 111: message_t.

[4]TEP 113: Serial Communication.