message_t

TEP:	111
Group:	Core Working Group
Type:	Documentary
Status:	Final
TinyOS-Version:	2.x
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

This memo covers the TinyOS 2.x message buffer abstraction, `message_t`. It describes the message buffer design considerations, how and where `message_t` is specified, and how data link layers should access it. The major goal of `message_t` is to allow datagrams to be passed between different link layers as a contiguous region of memory with zero copies.

1. Introduction

In TinyOS 1.x, a message buffer is a `TOS_Msg`. A buffer contains an active message (AM) packet as well as packet metadata, such as timestamps, acknowledgement bits, and signal strength if the packet was received. `TOS_Msg` is a fixed size structure whose size is defined by the maximum AM payload length (the default is 29 bytes). Fixed sized buffers allows TinyOS 1.x to have zero-copy semantics: when a component receives a buffer, rather than copy out the contents it can return a pointer to a new buffer for the underlying layer to use for the next received packet.

One issue that arises is what defines the `TOS_Msg` structure, as different link layers may require different layouts. For example, 802.15.4 radio hardware (such as the CC2420, used in the Telos and micaZ platforms) may require 802.15.4 headers, while a software stack built on top of byte radios (such as the CC1000, used in the mica2 platform) can specify its own packet format. This means that `TOS_Msg` may be different on different platforms.

The solution to this problem in TinyOS 1.x is for there to be a standard definition of `TOS_Msg`, which a platform (e.g., the micaZ) can redefine to match its radio. For example, a mica2 mote uses the standard definition, which is:

```
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;
```

while on a mote with a CC2420 radio (e.g., micaZ), `TOS_Msg` is defined as:

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

  // 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.

  uint8_t strength;
  uint8_t lqi;
  bool crc;
  uint8_t ack;
  uint16_t time;
 } TOS_Msg;
```

There are two basic problems with this approach. First, exposing all of the link layer fields leads components to directly access the packet structure. This introduces dependencies between higher level components and the structure layout. For example,

many network services built on top of data link layers care whether sent packets are acknowledged. They therefore check the `ack` field of `TOS_Msg`. If a link layer does not provide acknowledgements, it must still include the `ack` field and always set it to 0, wasting a byte of RAM per buffer.

Second, this model does not easily support multiple data link layers. Radio chip implementations assume that the fields they require are defined in the structure and directly access them. If a platform has two different link layers (e.g., a CC1000 *and* a CC2420 radio), then a `TOS_Msg` needs to allocate the right amount of space for both of their headers while allowing implementations to directly access header fields. This is very difficult to do in C.

The `data` payload is especially problematic. Many components refer to this field, so it must be at a fixed offset from the beginning of the structure. Depending on the underlying link layer, the header fields preceding it might have different lengths, and packet-level radios often require packets to be contiguous memory regions. Overall, these complexities make specifying the format of `TOS_Msg` very difficult.

TinyOS has traditionally used statically sized packet buffers, rather than more dynamic approaches, such as scatter-gather I/O in UNIX sockets (see the man page for `recv(2)` for details). TinyOS 2.x continues this approach.

2. message_t

In TinyOS 2.x, the standard message buffer is `message_t`. The message_t structure is defined in `tos/types/message.h`:

```
typedef nx_struct message_t {
  nx_uint8_t header[sizeof(message_header_t)];
  nx_uint8_t data[TOSH_DATA_LENGTH];
  nx_uint8_t footer[sizeof(message_footer_t)];
  nx_uint8_t metadata[sizeof(message_metadata_t)];
} message_t;
```

This format keeps data at a fixed offset on a platform, which is important when passing a message buffer between two different link layers. If the data payload were at different offsets for different link layers, then passing a packet between two link layers would require a `memmove(3)` operation (essentially, a copy). Unlike in TinyOS 1.x, where TOS_Msg as explicitly an active messaging packet, message_t is a more general data-link buffer. In practice, most data-link layers in TinyOS 2.x provide active messaging, but it is possible for a non-AM stack to share message_t with AM stacks.

The header, footer, and metadata formats are all opaque. Source code cannot access fields directly. Instead, data-link layers provide access to fields through nesC interfaces. Section 3 discusses this in greater depth.

Every link layer defines its header, footer, and metadata structures. These structures MUST be external structs (`nx_struct`), and all of their fields MUST be external types (`nx_*`), for two reasons. First, external types ensure cross-platform compatibility[1]. Second, it forces structures to be aligned on byte boundaries, circumventing issues with the alignment of packet buffers and field offsets within them. Metadata fields must be nx_structs for when complete packets are forwarded to the serial port in order to log traffic. For example, the CC1000 radio implementation defines its structures in `CC1000Msg.h`:

```
typedef nx_struct cc1000_header {
  nx_am_addr_t addr;
  nx_uint8_t length;
  nx_am_group_t group;
  nx_am_id_t type;
} cc1000_header_t;

typedef nx_struct cc1000_footer {
  nxle_uint16_t crc;
} cc1000_footer_t;

typedef nx_struct cc1000_metadata {
  nx_uint16_t strength;
  nx_uint8_t ack;
  nx_uint16_t time;
  nx_uint8_t sendSecurityMode;
  nx_uint8_t receiveSecurityMode;
} cc1000_metadata_t;
```

Each link layer defines its structures, but a **platform** is responsible for defining `message_header_t`, `message_footer_t`, and `message_metadata_t`. This is because a platform may have multiple link layers, and so only it can resolve which structures are needed. These definitions MUST be in a file in a platform directory named `platform_message.h`. The mica2 platform, for example, has two data link layers: the CC1000 radio and the TinyOS serial stack[2]. The serial packet format does not have a footer or metadata section. The `platform_message.h` of the mica2 looks like this:

```
typedef union message_header {
  cc1000_header_t cc1k;
  serial_header_t serial;
} message_header_t;

typedef union message_footer {
  cc1000_footer_t cc1k;
} message_footer_t;

typedef union message_metadata {
  cc1000_metadata cc1k;
} message_metadata_t;
```

For a more complex example, consider a fictional platform named 'megamica' that has both a CC1000 and a CC2420 radio. Its `platform_message.h` would look like this:

```
typedef union mega_mica_header {
  cc1000_header_t cc1k;
  cc2420_header_t cc2420;
  serial_header_t serial;
} message_header_t;
```

```
typedef union mega_mica_footer {
  cc1000_footer_t cc1k;
  cc2420_footer_t cc2420;
} message_footer_t;

typedef union mega_mica_metadata {
  cc1000_metadata_t cc1k;
  cc2420_metadata_t cc2420;
} message__metadata_t;
```

If a platform has more than one link layer, it SHOULD define each of the message_t fields to be a union of the underlying link layer structures. This ensures that enough space is allocated for all underlying link layers.

3. The message_t fields

TinyOS 2.x components treat packets as abstract data types (ADTs), rather than C structures, obtaining all of the traditional benefits of this approach. First and foremost, clients of a packet layer do not depend on particular field names or locations, allowing the implementations to choose packet formats and make a variety of optimizations.

Components above the basic data-link layer MUST always access packet fields through interfaces. A component that introduces new packet fields SHOULD provide an interface to those that are of interest to other components. These interfaces SHOULD take the form of get/set operations that take and return values, rather than offsts into the structure.

For example, active messages have an interface named `AMPacket` which provides access commands to AM fields. In TinyOS 1.x, a component would directly access `TOS_Msg.addr`; in TinyOS 2.x, a component calls `AMPacket.getAddress(msg)`. The most basic of these interfaces is Packet, which provides access to a packet payload. TEP 116 describes common TinyOS packet ADT interfaces[3].

Link layer components MAY access packet fields differently than other components, as they are aware of the actual packet format. They can therefore implement the interfaces that provide access to the fields for other components.

3.1 Headers

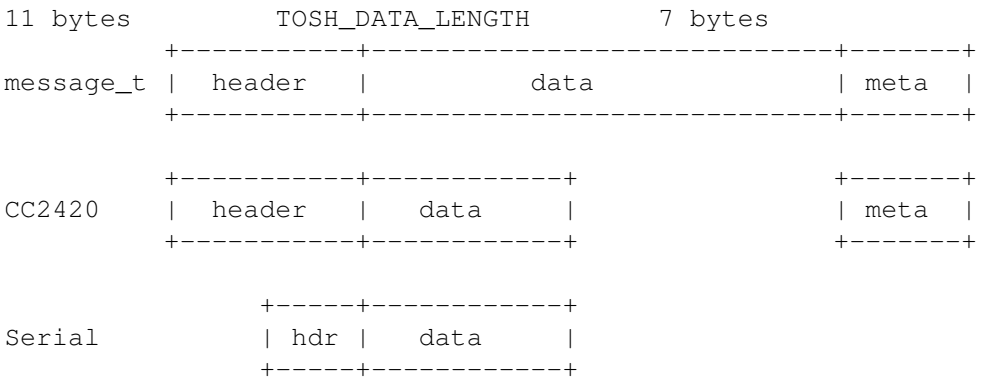
The message_t header field is an array of bytes whose size is the size of a platform's union of data-link headers. Because radio stacks often prefer packets to be stored contiguously, the layout of a packet in memory does not necessarily reflect the layout of its nesC structure.

A packet header MAY start somewhere besides the beginning of the message_t. For example, consider the Telos platform:

```
typedef union message_header {
  cc2420_header_t cc2420;
  serial_header_t serial;
} message_header_t;
```

The CC2420 header is 11 bytes long, while the serial header is 5 bytes long. The serial header ends at the beginning of the data payload, and so six padding bytes pre-

cede it. This figure shows the layout of message_t, a 12-byte CC2420 packet, and a 12-byte serial packet on the Telos platform:



Neither the CC2420 nor the serial stack has packet footers, and the serial stack does not have any metadata.

The packet for a link layer does not necessarily start at the beginning of the message_t. Instead, it starts at a negative offset from the data field. When a link layer component needs to read or write protocol header fields, it MUST compute the location of the header as a negative offset from the data field. For example, the serial stack header has active message fields, such as the AM type. The command that returns the AM type, `AMPacket.type()`, looks like this:

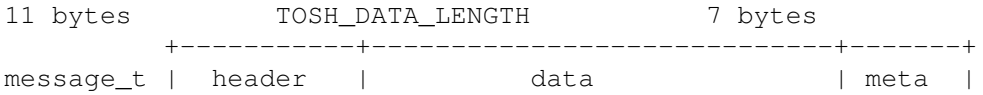
```
serial_header_t* getHeader(message_t* msg) {
  return (serial_header_t*)(msg->data - sizeof(serial_header_t));
}
command am_id_t AMPacket.type(message_t* msg) {
  serial_header_t* hdr = getheader(msg);
  return hdr->type;
}
```

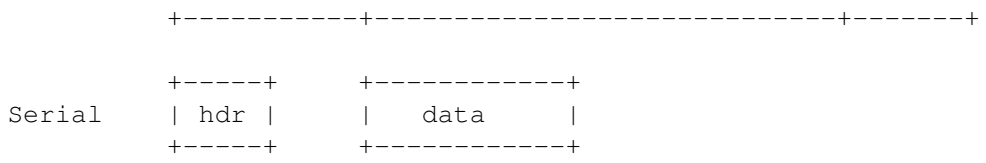
Because calculating the negative offset is a little bit unwieldy, the serial stack uses the internal helper function getHeader(). Many single-hop stacks follow this approach, as it is very likely that nesC will inline the function, eliminating the possible overhead. In most cases, the C compiler also compiles the call into a simple memory offset load.

The following code is incorrect, as it directly casts the header field. It is an example of what components MUST NOT do:

```
serial_header_t* getHeader(message_t* msg) {
  return (serial_header_t*)(msg->header);
}
```

In the case of Telos, for example, this would result in a packet layout that looks like this:





3.2 Data

The data field of message_t stores the single-hop packet payload. It is TOSH_DATA_LENGTH bytes long. The default size is 28 bytes. A TinyOS application can redefine TOSH_DATA_LENGTH at compile time with a command-line option to ncc: `-DTOSH_DATA_LENGTH=x`. Because this value can be reconfigured, it is possible that two different versions of an application can have different MTU sizes. If a packet layer receives a packet whose payload size is longer than TOSH_DATA_LENGTH, it MUST discard the packet. As headers are right justified to the beginning of the data payload, the data payloads of all link layers on a platform start at the same fixed offset from the beginning of the message buffer.

3.3 Footer

The message_footer_t field ensures that message_t has enough space to store the footers for all underlying link layers when there are MTU-sized packets. Like headers, footers are not necessarily stored where the C structs indicate they are: instead, their placement is implementation dependent. A single-hop layer MAY store the footer contiguously with the data region. For short packets, this can mean that the footer will actually be stored in the data field.

3.4 Metadata

The metadata field of message_t stores data that a single-hop stack uses or collects does not transmit. This mechanism allows packet layers to store per-packet information such as RSSI or timestamps. For example, this is the CC2420 metadata structure:

```
typedef nx_struct cc2420_metadata_t {
  nx_uint8_t tx_power;
  nx_uint8_t rssi;
  nx_uint8_t lqi;
  nx_bool crc;
  nx_bool ack;
  nx_uint16_t time;
} cc2420_metadata_t;
```

3.5 Variable Sized Structures

The message_t structure is optimized for packets with fixed-size headers and footers. Variable-sized footers are generally easy to implement. Variable-sized headers are a bit more difficult. There are three general approaches that can be used.

If the underlying link hardware is byte-based, the header can just be stored at the beginning of the message_t, giving it a known offset. There may be padding between

the header and the data region, but assuming a byte-based send path this merely requires adjusting the index.

If the underlying link hardware is packet-based, then the protocol stack can either include metadata (e.g., in the metadata structure) stating where the header begins or it can place the header at a fixed position and use `memmove(3)` on reception and transmit. In this latter case, on reception the packet is continguously read into the message_t beginning at the offset of the header structure. Once the packet is completely received, the header can be decoded, its length calculated, and the data region of the packet can be moved to the `data` field. On transmission, the opposite occurs: the data region (and footer if need be) are moved to be contiguous with the header. Note that on completion of transmission, they need to be moved back. Alternatively, the radio stack can institute a single copy at the botttom layer.

4. Implementation

The definition of message_t can be found in `tinyos-2.x/tos/types/message.h`.

The definition of the CC2420 message format can be found in `tinyos-2.x/tos/chips/cc2420/CC2420.h`

The defintion of the CC1000 message format can be found in `tinyos-2.x/tos/chips/cc1000/CC1000Ms`

The definition of the standard serial stack packet format can be found in `tinyos-2.x/tos/lib/serial/Ser`

The definition of the telos family packet format can be found in `tinyos-2.x/tos/platform/telosa/plat` and the micaz format can be found in `tinyos-2.x/tos/platforms/micaz/platform_message.h`.

5. Author's Address

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

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

6. Citations

[1] nesC: A Programming Language for Deeply Embedded Networks.

[2] TEP 113: Serial Communication.

[3] TEP 116: Packet Protocols.