

The Case for Buffer Allocation in TinyOS

Klaus S. Madsen
klausss@diku.dk

May 18, 2005

Introduction

Buffer management in TinyOS relies solely on buffer trading (buffer-swap) and transfer of ownership. These techniques are used in a straightforward way in the current radio interfaces of TinyOS. These techniques allows for static allocation of communication buffers, thus relieving the application from the hassle of memory management. The straightforward use of these techniques also gives rise to serious problems for packet based radio interfaces or when the communication speed and bandwidth of the radio increase. In the following we illustrate these problems using the BTNodes and their Bluetooth radio as an example, and we propose a solution.

Terminology

We will use the term application, when we mean the user-written component that uses the radio interfaces. When we talk of the lower levels, we mean everything not written by the application developer, ie. all the components provided by the TinyOS package.

The Mica radio implementation

On the Mica motes, the communication between the MCU and the TR1000-radio uses the MCUs SPI interface, communicating at 40 kbps. The TR1000-radio is not a SPI device, but using the SPI interface frees a lot of processing power on the MCU. The upper layers of the radio interface is implemented by the `MicaHighSpeedRadioM` component[2].

When the radio is not used for sending or receiving, another component called `ChannelMonC` samples what the radio receives in order to detect a start symbol. When it encounters a start symbol, it suspends itself, and signals the `MicaHighSpeedRadioM`, which then uses the SPI interface to receive the data after the start symbol. This sampling happens at 20 kHz and simply samples the data pin on the MCU, bypassing the SPI interface.

Since the radio simply modulates its signal, according to the input it receives from the MCU, all data needs to be encoded, in order to have error correction and detection of errors. This encoding is handled by the `SecDecEncoding` component¹, which encodes every byte into three. So while the raw communication speed is 40 kbps, the data rate is only 13.3 kbps.

When `MicaHighSpeedRadioM` detects that it has received a packet, it posts a task which sends an event to the application that a packet has been received. The radio is not serviced again before this task is completed. The reason for this is that the `MicaHighSpeedRadioM` component only has one receive buffer, so it has to wait for the task to complete, before a new buffer is available. This

¹Other encodings have been implemented, but the `SecDecEncoding` component is the one used by default.

receive buffer is statically allocated in the `MicaHighSpeedRadioM`, and is passed to the application through the posted task. The application can either process the data in the buffer and return the same buffer immediately, or it can choose to return another buffer, and process the data later.

The interface is optimized for small messages that can be fit into one packet. If one mote wishes to send enough data so that two packets are needed, it will have to wait at least 80 of the 20 kHz ticks between the two packets, as a part of the back-off algorithm. This corresponds to 16,000 clock cycles on the motes which have a 4 MHz MCU. Once the initial back-off is complete, the `ChannelMonC` component monitors the radio for further 12 of the 20 kHz ticks, in order to ensure that no other mote is sending. So if all motes are within range, the minimum gap between two packets will be 12 20 kHz ticks, or 2,400 clock cycles.

Bluetooth and the BTNode

On the BTNode the MCU and the Bluetooth module use the serial interface to communicate. The Bluetooth module automatically takes care of the MAC layer, and presents the MCU with an interface for connection oriented, packet based communication - this interface is called HCI (Host Controller Interface). The HCI allows the Host Controller, which in this case is the MCU, to control the Bluetooth module by sending control and data packets to it. The module can send events packets back to the MCU, either in response to the control packets from the MCU, or due to data received by the radio. These packets can be up to 668 bytes long, but the Host Controller are only required to be able to receive packets that are 255 bytes long[4].

All the control and data packets contain a length parameter. This parameter is used by the MCU and Bluetooth module to determine when one packet ends and a new one begins. The result of this is that the MCU must not miss a single byte. If a byte is missed, the MCU splits the packets wrongly, and this cannot be detected immediately. The only way to detect the missing byte is to have a timeout that is triggered when the MCU has spent more than a certain amount of time receiving the same packet.

Furthermore the Bluetooth module is capable of communicating with the MCU at speeds up to 400 kbps. While the Bluetooth module can communicate at lower speeds, the communication speed need to be 400 kbps in order to saturate the radio bandwidth. This is desirable because the energy consumption of the Bluetooth module is very high, around 30 mW when idle[3]. The only way to lower this energy consumption is to power off the Bluetooth module. Turning it on and initializing it takes around 3 seconds, or 22,118,400 clock cycles for the BTNodes 7.3728 MHz MCU. Therefore the module is better suited for sending data in large bursts, where the high bandwidth is well suited.

Because the Bluetooth module is so different from the TR1000-radio, it is inconvenient to use an implementation similar to the one used for the radio on the Mica motes. This can be illustrated quite easily:

The BTNode has a MCU that is comparable to the MCU on the Mica motes², but as previously mentioned it runs at 7.3728 MHz. The UART on the MCU contains an 8 bit receive buffer. When this buffer is filled, the contents are copied to a register, and an interrupt is signaled. This means that the BTNode within the time it takes to receive the next 8 bits, needs to have read this value from the buffer, in order to make sure that it is not overwritten. If the MCU discovers that it has received an entire packet, it must switch to a new buffer. This need to be done within a further 7 bits time frame. Otherwise the still unprocessed byte will be overwritten.

On the Mica mote, a new packet will only be sent if there has been no radio communication for at least 2400 clock cycles (12 bits at 20 kbps) or 0.60 ms. On the BTNode where the UART speed is 400 kbps it takes as little as 0.020 ms to receive 8 bits and 0.038 ms to receive 15 bits. This

²The original Micas use the ATmega103 MCU, but newer versions (like the one produced by XBow) use the ATmega128 MCU, just like the BTNode

means that the BTNode needs to be able to detect that it has received a packet, and switch the buffers in 0.038 ms. The MCU on both the BTNode and the Mica has an average CPI³ of 1.4[1]. So when running at 7.3728 MHz, 0.038 ms equals approximately 200 instructions. Compare this to the Mica mote where 2400 instructions can be executed before it must be ready for the next packet.

Detecting that the entire packet has been received, and switching the buffer should be possible in 200 instructions without too much trouble, if a new buffer is available. But it is impossible to post a task in order to find a new buffer, unless the lower layers has more than one receive buffer ready. However some of the event packets can be as small as 8 bytes, so these packets will be transmitted to the MCU in around 840 instructions time. Therefore a lot of buffers will be needed, if a task is used to retrieve new buffers.

It is undesirable to allocate many buffers in the lower levels of the Bluetooth stack, because they need to be allocated with enough room to fit an entire Bluetooth packet. And since a Bluetooth packet can be up to 668 bytes large, a lot of the BTNodes internal 4 KiB⁴ memory can be consumed by these buffers. The amount of buffers needed at the lower layers depends on how the application is going to use the Bluetooth radio: If it just sends a single packet occasionally, few buffers are needed, but if application needs the full bandwidth of Bluetooth, many buffers will be needed.

Providing buffers for the lower layers

The problem of having buffers available in the lower layers in order to switch receive buffer quickly can be solved in several ways. The simplest is not to post a task to signal the receive event to the higher layers, but simply signal the application directly. Doing this requires that the receive event becomes asynchronous (marked with the `async` keyword), since it must be called from the context where we discover that the packet is fully received. This context will be in some sort of interrupt procedure, which are asynchronous in nature.

The problem with this approach is that a lot of responsibility is placed on the application programmer. He must make sure that the asynchronous receive event returns a new buffer in less than 0.038 ms, otherwise the lower layers risk dropping bytes and in turn packets. Since the application usually need to process the received data, it will in many cases have to defer the packet to a buffer, and return a new one, in order to meet the time constraints. This is not very intuitive for the application developer.

Another solution is to split the problem into two separate cases: Before the lower layers can return a buffer to the application, they must have a new buffer ready to receive the next packet in. In order to obtain this new buffer, the lower layers signal a `getBuffer` event to the application in order to get one. This event must be asynchronous, as it needs to be executed immediately. Once a new packet is in place, the old buffer can safely be put aside, in order to return it to the application through a posted task. This way, it is simple for the application programmer to keep within the time constraints of the `getBuffer` event. The application is also responsible for allocating the buffer space, thus allowing the amount of memory used for these buffer to be fine-tuned for the specific application. The drawback with this approach is that the application still has to implement handling of the asynchronous `getBuffer` event.

What happens when sending data?

Until now we have only concerned ourselves with the handling of packets received *from* the radio-interface. We still need to explore what happens when the application sends packets to the radio.

³Cycles per instruction

⁴The BTNodes also have 60KiB of external memory, but it is very energy consuming, which is why we do want to use it.

Where the Micas receive interface uses buffer-swap, the send interface uses transfer of ownership. This means that when the application is ready to send a message, it presents the lower layers with a pointer to the buffer. Once the call has completed, the application is not allowed to alter the contents of the buffer, before the lower layers returns the pointer.

This simple interface works nicely, because when data is sent, the application is in control of when it starts sending data. That is, there is no risk of dropping bytes, if the application does not have a new packet ready the moment the lower layers are finished sending the previous packet.

The Mica stack works by allocating a packet for each different packet type the application needs to send. These packets are allocated statically in the main memory, so if the application uses many different packets, it will use a lot of the available memory for the packets.

The Mica solution work when the number of different packets the application needs to send is small. The HCI interface needs a lot of different configuration packets in order to simply control the Bluetooth module, e.g. to configure the module when it is turned on 3 different packets are required. More packets are needed in order to locate other nodes, or to listen for incoming connections. As many of these packets are only needed once when configuring the radio, it would be nice if the memory they consume could be used for the application afterwards.

Another problem appears when the application needs to saturate the bandwidth of the radio. In order to do so, the application must have a new packet ready to send, as soon as the radio is finished with the previous one. Therefore the node needs to prepare the next packet, while it is still sending the previous one to the radio, and therefore two send buffers are needed.

Introducing a buffer-manager

In the previous we have outlined several problems that needs to be solved in order to take full advantage of the Bluetooth module. Now we will try to solve these problems by introducing a dedicated component, a buffer-manager, to provide packets for both the application and the lower layers.

When the lower layers emit the asynchronous `getBuffer` event to the application, the application needs to implement a asynchronous interface. If the lower layers instead emit the `getBuffer` event to the buffer-manager, the application does not need to handle asynchronous events. This also ensures that the lower layers are provided with new packets within the required time-frame, removing this requirement from the application.

Also by allowing the application to request packets from the buffer-manager, it is no longer necessary to statically allocate packets in the application. The application can simply request a new buffer from the buffer-manager, when it needs to prepare a new packet. This reduces the memory requirements, and makes it simple for the application programmer to saturate the radio bandwidth.

When both the application and the lower layers are using the buffer-manager, it is necessary to ensure that at any given time, there is enough packets left to service the lower layers. This can be solved either by using having separate buffer-manager components for the application and the lower layers, or by having one buffer-manager with two interfaces. Using two buffer-manager components is not desirable, as it will cause code duplication, thereby raising the applications memory requirements. On the other hand a single buffer-manager component with two interfaces, can be implemented without significant overhead. When the buffer-manager get a `getBuffer` event from the application, it should only return a new buffer, if there is sufficient free buffers left.

Another problem introduced by a dedicated buffer-manager component, is how to allow the application programmer to fine-tune the number and size of the buffers in the buffer-manager to the specific application. One solution is to provide several buffer-manager components, with the different number of buffers, and different buffer sizes. If enough of these components are provided,

the application programmer can simply choose the component that fits the needs of the application. However having many components providing basically the same service is not a very elegant solution. Also this solution cannot provide the application programmer with ultimate flexibility, and the many similar components will be a problem from a maintenance point of view. The generic component interfaces, introduced in NesC version 1.2, can be used to solve this problem instead. A generic buffer-manager component should then have two parameters, one which controls the number of buffers allocated by the buffer-manager, and one which controls the size of these buffers. So by introducing a buffer-manager, many requirements are moved from the application, into this component. Also it is possible to implement it in such a way that the application programmer still maintains full control over the applications memory requirements.

Conclusion

From the above analysis it should be clear that to ensure reliable packet transmission and to utilize the full capabilities of the radio, the solution used for the Mica cannot be used for the Bluetooth radio, or for other advanced and fast radios. Some sort of buffer manager is needed. It is also clear that the burden on the application programmer is minimized if we introduce a dedicated buffer manager, and that it should be simple to provide the application programmer with sufficient flexibility to fine-tune the memory consumption to a specific application.

Introducing a buffer manager is not entirely in the spirit of TinyOS. One of the major design points for TinyOS, is that it does not do dynamic memory allocation, and this is essentially what buffer management is. However the number of issues solved by introducing a buffer manager, presents a strong case for its use, especially in conjunction with advanced and high bandwidth radio technologies like IEEE 802.15.4 and Bluetooth.

References

- [1] Heyley Iben, Ali Lakhia, and Rachel Rubin. Watchdog Designs for TinyOS Motes. CS 252 Final Project, May 2002. Available from: <http://www.cs.berkeley.edu/~iben/Classes/cs252/projectpaper.ps>.
- [2] Nelson Lee, Philip Levis, and Jason Hill. *Mica High Speed Radio Stack*, September 2002. Available from: <http://www.tinyos.net/tinyos-1.x/doc/stack.pdf>.
- [3] Martin Leopold, Mads Bondo Dydensborg, and Philippe Bonnet. Bluetooth and Sensor Networks: A Reality Check. In *Proceedings of the First International Conference on Embedded Networked Sensor Systems*, pages 103–113, 2003. Available from: <http://www.distlab.dk/public/distsys/publications.php?id=38>.
- [4] Bluetooth SIG. *Specification of the Bluetooth system - Core*, 1.1 edition, February 2001. Available from: <http://www.bluetooth.org/>.