A Tutorial for Programming in TinyOS*

Chris Merlin
ECE 245 — Spring'09
WCNG, University of Rochester.

January 26, 2009

This document provides a quick introduction to TinyOS-1.x and includes the content of your first lab in Section 6. Make sure you answer all the questions, using full sentences. Add your code when asked.

1 Introduction

TinyOS is an operating system developed for the various Berkeley mote platforms. A mote is a small wireless communication device composed of a radio, a CPU with memory, and sensors. Motes are typically small in size, energy efficient, and limited by their relatively low compute power.

TinyOS provides a relatively simple method of developing and quickly implementing wireless sensor networks. The motes are programmed with a language called nesC. NesC is an extension of C developed to be used with the TinyOS platform and that was meant for component-based programming. The focus of this document is the MoteIV's Tmote Sky devices. The aim of this document is to provide background information on TinyOS, nesC and the Tmote Sky motes.

1.1 TinyOS Tree, MoteIV Directory, and Java Tools

On windows computer, programming the motes is performed through the Cygwin interface. Cygwin is a UNIX/Linux style command line interface. The TinyOS files are found in the `/opt` directory. Within this dictory there are the `tinyos-1.x`, `moteiv` and `msp430` directories. The majority of interest is in the `tinyos` and `moteiv` directories.

1.1.1 TinyOS Tree

Usually, only the `apps`, `tools` and `docs` directories are commonly used by novices. The `apps` directory holds all the example applications provided with TinyOS. The `tools` directory is where all the java applications are stored. When interfacing TinyOS with Matlab, it is necessary to modify

*Parts of this Tutorial were written with the help of R. Aures and M. Holland.

several files in this directory. The docs directory has lots of documentation on TinyOS, including an in depth tutorial.

1.1.2 MoteIV Tree

This directory contains files particular to the MoteIV products. If the installation of TinyOS used is from the TinyOS website and not MoteIV's installer, most of these files can be found in */opt/tinyos-1.x/contrib/moteiv*. Again most interest is in the *apps* and *docs* directories. The *apps* directory contains the applications designed to specifically work on MoteIV products. It is recommended that the version of the application TOSBase is used from this directory if the platform used is `tmote`. The *docs* directory contains a useful listing of all the available TinyOS components as well as illustrations showing how the components are interconnected.

1.1.3 Java Tools

Once data has been gathered on the network it is often desirable to process or display the data on a computer. This is done by connecting a mote to the USB port and taking the data using the serial interface. While any program can be written to gather the data off the motes, there are three commonly used java programs that come packaged in TinyOS. This section can be used to make sure that these applications have been properly installed or referred back to when necessary. If using the *bashrc* file provided, the three programs serial forwarder, listen and oscilloscope have the shortcuts `sf`, `listen`, and `osc`.

The first application is the serial forwarder. As its name implies, this program can forward the data from the serial port to other applications (such as Matlab). A basic use has a window showing the number of packets received. This is done at the Cygwin command line by typing:

```
java net.tinyos.sf.SerialForwarder -comm serial@COM10:telos
```

This is shown with the mote connected to port 10. To determine what port your mote is connected to, type the command `motelist` at the Cygwin command line. The `COM10` from the above command should be replaced with whatever port your mote is attached to. This needs to be done for all the subsequent applications as well.

The next application is *listen*. This program prints each received raw packet at the Cygwin command line in hexadecimal numbers. The program is called using the following two lines:

```
export MOTECOM=serial@COM10:telos
java net.tinyos.tools.listen
```

The final program discussed is the oscilloscope program. This program is designed to be used in conjunction with the *moteiv/apps/Oscilloscope* nesC program. It graphicly displays the data from all the sensors on the mote. Other applications could be made to use this if their packets were made to conform to the ones used in the *Oscilloscope* nesC program. The command used to call the java oscilloscope is:

```
export MOTECOM=serial@COM10:telos
java net.tinyos.oscope.oscilloscope
```

1.2 Compiling and Installing Programs

To compile, you need to have Makefile in your folder. Whether your target is Tmote, Telosb, or pc, your Makefile should contain different flags (which can be found in the 'apps' folders of /opt/). Take a look at:

```
COMPONENT=Component_Name
include /path_to/Makerules
MOTEIV_DIR?=/opt/moteiv/
DEFAULT_LOCAL_GROUP = 0x7E
```

The latter variable is particularly important because it determines what group your node belongs to. If two groups of nodes share the same group number, nodes receive one another's packets. Different group numbers will "isolate" nodes whose program was built with the same number[1].

`MOTEIV_DIR ?=/opt/moteiv/` is needed when making for the tmote platform.

Cygwin provides a basic command line interface for programming the motes. The programming of the Mote is done with the `make` command. The make command must be typed in the same directory as the Makefile.

The command `make [platform]` compiles the program for the particular platform and creates an `xml` image. For instance, `make tmote` will compile the program for the `tmote` platform. The MoteIV tmote sky motes are compatible with both the `tmote` and the `telosb` platforms, though it is recommended that the `tmote` platform be used. If no platform is specified, the PC platform is used.

Note that the Makefile's used for the Tmote platform and for the PC platform have different flags. Look for the Makefile provided in the Pinger example to select and comment out the correct flags.

Installing the program on a mote is done with the `make [platform] install` and `make [platform] reinstall` while in the application directory. `Install` complies the program, creates an `xml` image, and then installs it on the mote while `reinstall` uses the current image to program the mote.

1.3 Example: Install Blink

This section will go through step by step the process to install the blink application onto a mote.

1. *Navigate to the Blink directory* - this is found in */opt/tinyos-1.x/apps/blink*
2. *Type* `ls` - This shows the contents of the directory. Note that if this is your first time compiling *blink*, there are only the .nc files and the makefile.
3. *Type* `make tmote` *or* `make telosb` - You should now have a build directory.

[1] In fact, all nodes' radios receive packets from different groups, but the data link layer discards these packets

4. *Connect the mote* - Connect the mote to the USB port of the computer. If you have not done so already, install the drivers.

5. *Type* `make tmote reinstall` - This takes the files that were generated when you typed `make tmote` and uses them to install blink on the mote.

6. *Done* - The mote should now be blinking.

This is meant to show each step of installing a program on a mote. It would be more typical to navigate to the directory and type `make tmote install`. Because we compiled separately beforehand, there was no point in recompiling. It is good to be conscious of the fact that modifying the .nc files will not automatically regenerate your compilation files, you must recompile or use `make tmote install` each time you update your program.

2 Basic Programming

2.1 Components, Implementations, and Wiring

TinyOS is written in nesC, a C-based language intended for programming structured component-based applications.

2.1.1 Components and Interfaces

A TinyOS program can be broken in various blocks or *components*, *e.g.,* application, routing, radio. Components can be viewed as black boxes whose input and output functions are known (***interfaces***).

There are two types of components: configurations and modules.

Configurations: The configuration of a component is marked by the codeword `configuration` and lists the interfaces it provides. Configurations are used to assemble other components together, linking interfaces used by components to interfaces provided by others.

Interfaces define a set of functions that can be used bi-directionally by any component. Components can only be linked to one another using and implementing interfaces. Interfaces must be put in a file in the form `Interface.nc`, and usage has interfaces' first letter capitalized.

Consider the example of a routing protocol component *Routing* that provides the *SelectRoute* interface, whose role is to find the next-hop node along a path.

```
configuration Routing
{
    provides
    {
        interface SelectRoute;
    }
}
```

```
implementation
{
    ... (see 2.1.2)
```

Modules: Modules provide application code and may *use* and *provide* several interfaces (for instance the interfaces **Leds** and **Timer** for a blink application).

A module must first declare what interfaces it uses and provides; it then gives the corresponding code in the `implementation` section.

Example:

```
module RoutingM
{
    uses
    {
        interface Leds;
        interface SendMsg;
        interface ReceiveMsg;
    }
    provides
    {
        interface SelectRoute;
    }
}

implementation
{
    command result_t SelectRoute.function...
}
```

2.1.2 Wiring

The operation of linking an interface used by a component to an interface provided by another one is called 'wiring'. A configuration file wires modules or other components interfaces in its implementation section.

Continue the previous example: *RoutingM* needs to toggle LEDs and send and receive packets, as well as provide *SelectRoute*.

```
configuration Routing
{
    provides
    {
        interface SelectRoute;
    }
}

implementation
```

```
{
    component LedsC, GenericComm as Comm, RoutingM;

    RoutingM.Leds -> LedsC.Leds;
    RoutingM.SendMsg -> Comm.SendMsg[AM_TYPE];
    RoutingM.ReceiveMsg -> Comm.ReceiveMsg[AM_TYPE];

    SelectRoute = RoutingM.SelectRoute;
}
```

- **SendMsg** is an interface provided by *GenericComm* whose `send(...)` function takes care of transmitting the packet over the medium. `send(...)` takes the packet's destination, its pointer, and its length.
- **ReceiveMsg** is also provided by *GenericComm* and takes care of receiving packets.
- **Leds** deals with the red, yellow, and green LEDs. Commands include `redToggle()`, `greenOff()`, etc.

Routing wires *LedsC*, *GenericComm*, and *RoutingM* together. *RoutingM* is the module that implements the various routing protocol functions. It turns LEDs on and off for debugging purposes and uses the radio to send and receive packets. The LEDs are handled by *LedsC*, so the **Leds** interface used by *RoutingM* is provided by *LedsC*. This user / provider relation is signified by the arrow "→".

On the other hand, some components may provide interfaces that are *directly* provided by other components. Here, our routing component protocol chooses to simply call the **SelectRoute** interface provided by *RoutingM*. When the application wants to find whether a route is available to the data sink using the **SelectRoute** interface, it really directly uses the *RoutingM* component's **SelectRoute** interface. In such a case, the relation is shown with a simple equal sign "=".

Note that when the interface used by a component has the same name as the interface provided by another one, it can be omitted on the right side of the assignment.

Example:

```
RoutingM.Leds -> LedsC;
```

2.2 Wiring to Send and Receive Packets

Next, we look at our first program that will send numbered packets and flicker LEDs. We need several interfaces and their commands: **StdControl**, **SendMsg**, **ReceiveMsg**, **Leds**, and **Timer**, as well as a pointer to the packet. Under TinyOS, a packet is a structure of fields defined by `TOS_Msg`, which contains various fields: address of the next-hop, packet length, signal strength, etc. The radio does not necessarily send *all* the fields defined in `TOS_Msg`.

Commands are immediately executed and are always functions declared in an interface. They must be invoked by `call`. Functions that are not part of an interface or subroutines may be directly invoked if they are within the file's scope.

- **StdControl** is implemented in almost all components and takes care of initializing, starting, and stopping modules. It includes three functions: `init()`, `start()`, and `stop()`. In the first one, the user may code initializations to all elements in his/her implementation. For instance, we will be setting the LEDs to turn off, and the packet number variable to 0. `stop()` prepares the component to turn off (for instance, turning off the radio).

- **Timer** provides functions to control LEDs, and *LedsC* instances of a timer. Commands are `start(mode, time)` (where *mode* can take `TIMER_ONE_SHOT` or `TIMER_REPEAT` and *time* is the time before the timer goes off) and `stop()`. The event `fired()` is triggered when the set time expires.

If set properly, the `send` command may be used thusly:

```
call SendMsg.send(destination, length, pointer)
```

The *destination* may be `TOS_BCAST_ADDR` or any 2 byte integer. `TOS_LOCAL_ADDRESS` designates the node's address.

To send a message, a `TOS_Msg` packet (or a pointer to this packet) has to be created in the global scope of the implementation, and not merely in the function `SendMsg.send`. Why not? Think about what would happen if you declared the packet within the scope of `SendMsg.send` *only*: when the packet is passed along to *GenericComm*, `SendMsg.send` simply exits (with usually a `SUCCESS` outcome), and the `TOS_Msg` packet is deconstructed. The packet is removed from the memory, and the radio has nothing to send.

The event `sendDone(TOS_MsgPtr pointer, result_t outcome)` is signaled when the packet has been sent (*outcome* is a success) or dropped (*outcome* is a failure).

When a packet is received, the following event is signaled:

```
event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr rMsg)
{
    ...
    return(rMsg);
}
```

The `receive` function should always return a pointer.

The **SendMsg** and **ReceiveMsg** interfaces may be parameterized: an integer is added to the interface and works as an identifier for packet types, much like the wiring above (recall `RoutingM.SendMsg -> Comm.SendMsg[AM_TYPE];`). Consequently, if an interface is parameterized, an event may fire for which there is no component wired to handle. For instance, *GenericComm* may receive 256 packet types, but has component logics for only a small subset of them. The codeword `default` is used to provide for the cases when no component was specifically wired to handle a certain message type.

2.3 Collecting Data and Debugging

2.3.1 TOSBase

TOSBase is a program that listens to the medium for packets. It is a great tool for debugging radio implementations. TOSBase is meant to be run onto a mote that is plugged into the USB port of a computer running the `listen` java tool. We often refer to the program and this mote as TOSBase. In order to avoid any confusion in this tutorial, we will call the mote running TOSBase the *BaseMote*.

Two major disadvantages of TOSBase are:

- TOSBase drops packets from the BaseMote radio to the computer if the UART data rate is smaller than that of the radio. In other words, packets may be all received by the BaseMote radio, however, they may not all be sent to the computer.
- The packets's information bytes are not reordered before being printed on the computer: memory storage is little-endian.

One of the great virtues of TOSBase is the possibility to read the data received by the BaseMote (and snoop on the network). Matlab can also be used to collect data from the network using TOSBase. With the proper installation, Matlab can cast packets and retrieve data fields from known packet types.

2.3.2 TOSSIM

Generalities TOSSIM is the TinyOS simulator. Debugging traces can be set in the form: `dbg(DBG_TEMP, "Message");` and a `make pc` will compile the program for the simulator. Running the simulation is done with: `DBG=temp, build/pc/main.exe -l=speed_of_sim num_of_nodes`.

TOSSIM does not model radio propagation; instead, it provides a radio abstraction of directed independent bit errors between two nodes. TOSSIM does not model power draw and energy consumption. As a consequence, energy drain must be evaluated through other tools, or additional code must be provided to model the radio, sensor, and CPU drains.

Compiling and Running a Simulation After your `make pc` successfully compiles, you can execute `main.exe` with some of the following options:

```
Usage: ./build/pc/main.exe [options] num_nodes
    [options] are:
    -a=<model>  specifies the ADC model (generic is default)
                options: generic random
    -b=<sec>    the time (in seconds) over which motes boot (default: 10)
    -l=<scale>  runs the simulation at <scale> time(s) the real time
    -r=<radio>  specifies a radio model
                options: simple static lossy
    -rf=<file>  specifies the input file for the lossy model
    -s=<num>    boots only <num> nodes
    -t=<time>   runs the simulation for <time> virtual seconds
    num_nodes   specifies the number of nodes
```

Lossy Radio Model The lossy radio model can help you create a multi-hop network of nodes by specifying the bit error rate on each link n_i to n_j (where $i \neq j$ and $i, j \in [0, num\_nodes]$). TOSSIM looks for the file "lossy.nss" by default, but an alternate file can be specified with the `-rf` tag. The following format is used:

```
<mote ID>:<mote ID>:bit error rate
```

So, for example, specify:

```
0:1:0.01
1:0:0.05
```

for an asymmetrical link between nodes 0 and 1 with bit error rates of 1% and 5%. Use this option to create new topologies.

More information on TOSSIM can be found at *http://www.tinyos.net/nest/doc/tutorial/tossim-lesson.html* and in [3].

3 Advanced Programming Considerations

Very often, TinyOS beginners will hit a plateau after they have implemented simple code for the **Send** and **Receive** interfaces. Further consideration needs to be paid to timing issues and the handling of several layers.

3.1 Atomicity

The motes' hardware is made of both synchronous and asynchronous components: for instance, a sensor reading does not necessarily return a value immediately, and when it does so, it may interrupt functions running on the processor. Thus, TinyOS provides support for "split-phase" functions which are typically marked by a call function that starts a process, and an interrupt that lets components know that the process has completed.

3.1.1 Async and Sync Codes

Interrupt handlers preempt other pieces of code running on the processor[2]. Consequently, a variable may be changed by two competing functions (a *race* condition).

The problem that exists with asynchronous codes is illustrated in the following:

```
async command bool toggle()
{
    if (state == 0)
    {
        state  = 1;
        return 1;
```

[2]This is not true of TOSSIM, whose interrupts are not preemptive (they do not interrupt pieces of code already running). As a consequence, race conditions that may not be apparent in TOSSIM might exist on motes.

```
    }
    if (state == 1)
    {
        state = 0;
        return 0;
    }
}
```

Now look what happens if the command is called in the middle of another execution (we start with *state == 0*):

```
INTERRUPT CALL: toggle()
|   since (state == 0)
|   state = 1;
|       -> INTERRUPT CALL: toggle()
|       |   since (state == 1)
|       |   state = 0;
|       |   returns 0;
|       END INTERRUPT CALL
|   return 1;
END INTERRUPT CALL
```

The function will return 1 even though *state* is in fact 0.

Functions that can run preemptively are labeled 'async': they run asynchronously with respect to tasks. By default, events and commands are synchronous. The `async` codeword specifies when they are not.

All interrupt handlers are asynchronous—but **ReceiveMsg** or **Timer** interfaces [4] are not of them.

For example, the **SendMsg** interface is completely synchronous. **LedsC** on the other hand is fully asynchronous. Contrary to intuition, the **Receive** interface is synchronous. This is because even though the radio or UART asynchronously signals a received message, the interrupt handler reads the received byte from the radio or UART and posts a task that signals packet reception.

3.1.2 Tasks

A task is a very particular function that executes synchronously. Tasks can be interrupted but they are atomic with respect to one another. A task cannot preempt another task, and tasks are executed in the order they were posted (FIFO).

The other particularity of tasks is that they execute after the command or the event that posted them (thus, they introduce latency). Commands and events then run to completion—and a buffer may be returned to underlying layers earlier.

Because tasks do not run immediately, they have no return value. Also we learn in [1] that a task executes within the naming scope of a component: any parameter needed to run the task can

be stored in the component. Thus, posting a task takes no parameter.

Tasks as well should be kept relatively short. Other regions of the protocol stack may post tasks too—for instance, the radio stacks. Since a task cannot interrupt another task, a significant delay may be introduced. A reception rate may be limited by how quickly a radio can post tasks to signal reception.

3.1.3 Atomic Statements

The code word `atomic{...}` can delimit sections of code that run atomically. However, it does not mean that an atomic block cannot be preempted; it only means that two segments that access the same variables cannot preempt one another. In the previous example, you could write:

```
async command bool toggle()
{
    int8_t currentState;
    atomic currentState = state;
    if (currentState == 0)
    {
        atomic state  = 1;
        return 1;
    }
    if (currentState == 1)
    {
        atomic state = 0;
        return 0;
    }
}
```

Atomic blocks should be used when a variable can be accessed by an asynchronous function. However, a programmer should not overuse the `atomic` codeword since it has a CPU cost. A block should also be of small size—the exact length of an atomic statement is a complex matter.

3.2 Encapsulation

Many novice programmers overlook this powerful tool because they tend to design a protocol stack with only one layer. However, encapsulation and packet handling methods exist.

Imagine you want to sense data and send it to a base station, which may be located more than one hop away. You will need to write several components: an application layer that senses the environment, a routing protocol that routes packets over several hops, and a radio link.

Let us start from the bottom up. Every TinyOS message has the same structure, which is defined with the `TOS_Msg` definition:

```
typedef struct TOS_Msg
```

```
{
  /* The following fields are transmitted/received on the radio. */
  uint16_t addr;        // Next-hop destination address
  uint8_t type;         // Identification type for packet: allows dispatching
  uint8_t group;        // Group number
  uint8_t length;       // Packet length
  int8_t data[TOSH_DATA_LENGTH];    // Payload
  uint16_t crc;         // crc check

  /* 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;    // Signal strength for the received packet
  uint8_t ack;          // Request ACK
  uint16_t time;        // Time packet is received
} TOS_Msg;
```

`addr` is the address of the next-hop, `type`, a number assigned to the packet that allows identification. The component *GenericComm* automatically assigns the `AM_TYPE` wired with the **SendMsg** and **ReceiveMsg** interfaces automatically to this field. The same happens with the field `group` receiving the `DEFAULT_LOCAL_GROUP` number defined in the Makefile. `data` is the real payload of the packet that the programmer can fill.

Every time you use a `TOS_Msg` packet and add a payload, you are encapsulating your data with headers and footers. This payload can further be encapsulated. Let us go back to the program that sends and receives packets. Invoking `call SendMsg.send(destination, length, pointer)` should send the packet pointed to by *pointer* with payload size *length* to the *destination*. The pointer must be of type `TOS_Msg`:

```
TOS_Msg sMsg;
TOS_Msg *sMsgPtr = &sMsg;
```

In order to add a payload to this packet, another pointer must be created to access that memory space and fill it: this is the job of the routing protocol, which in this particular case sits above *GenericComm* and below the application.

Example:

```
typedef struct RouterMsg
{
    uint8_t routerType;
    int8_t data[(TOSH_DATA_LENGTH - 1)];
} RouterMsg;
```

defines the packet structures used by the routing protocol. Then, you can do the following: `RouterMsg *route = (RouterMsg*)sMsgPtr->data;`.

This will cast the payload to a `RouterMsg` type and return a pointer to that area. However, one of the fields of the `RouterMsg` type is `data`, an area that the application can use and fill however it wants. The previous operation must be repeated.

However, the routing protocol has certain conventions, such as the use of a `Send.getBuffer()` command, which returns a pointer to the usable "`data`" area in `RouterMsg`. This is a great tool to abstract lower layers: the layer issuing a packet does not need to know what fields other layers use.

```
typedef struct AppMsg
{
    float sensed;
} AppMsg;
```

defines the fields used by the application layer.

The application does not need to know the structure of the headers and footers used by the routing protocol (this allows easier protocol swapping). Thus, `getBuffer` needs to return the *data* field of `RouterMsg` packets.

A possible implementation of `getBuffer` is:

```
command void* Send.getBuffer(TOS_MsgPtr sMsgPtr, uint16_t *length)
{
    RouterMsg *route = (RouterMsg*)sMsgPtr->data;
    *length = TOSH_DATA_LENGTH - offsetof(RouterMsg, data);
    return(route->data);
}
```

When sending a packet, the application calls a `Send.send` function that passes a pointer to the message payload.

To fill the *sensed* field, the application may do the following:

```
AppMsg *application = (AppMsg*) call Send.getBuffer(&sMsg, &length);
// Alternatively: (AppMsg*) call Send.getBuffer(sMsgPtr, &length);
application->sensed = (float) value;
```

Upon receiving a `send` command, the routing protocol is given a pointer to the message, and may do the following:

```
command result_t Send.send(TOS_MsgPtr sMsgPtr, uint16_t payloadLength)
{
    RouterMsg *route = (RouterMsg*)sMsgPtr->data;
    uint16_t packetLength = offsetof(RouterMsg, data) + payloadLength;

    route->routerType = (uint8_t) rType;
}
```

Note that if the routing protocol knew the application message structure, it could modify its fields by doing the following:

```
AppMsg *application = (AppMsg*)route->data;
application->sensed = (float) mod_value;
```

Reading the information contained in a received packet is done similarly: a cast must be performed on the `data` area of `TOS_Msg` packets, implying that the packet type must be known, thus justifying why there is a field `type` defined in the `TOS_Msg` packet.

3.3 Dynamic Memory Allocation

You may have heard that TinyOS did not allow dynamic memory. This is both true and false.

True, because calls to the function `void* malloc(int size)` will occasionally crash on motes, and because the function `void* realloc(void *object, int size)` is not supported on these motes.

False, because `malloc`, `realloc`, as well as `free(void *ptr)` will successfully execute on TOSSIM. False again, because some level of dynamic memory allocation is allowed through the **MemAlloc** interface for real mote deployments.

3.3.1 Memory Model of TinyOS

We learn in [5] that TinyOS, being a simple and lean Operating System, only allows static memory allocation, which requires no management of the dynamic heap. When calling `malloc`, a space of appropriate size must be found in the memory, and this operation can be very resource consuming when there is little space left or when the memory is very fragmented. However, nothing prevents someone from writing a memory manager, and some people, like Sam Madden, already have.

3.3.2 The Memory Allocation Component and Interface

MemAlloc is provided by the component *tos\lib\Util\TinyAlloc*. Some of the commands are:

- Allocate a memory region: `result_t allocate(HandlePtr handle, int16_t size);`
- Reallocate a region to a new size, copying data over: `result_t reallocate(Handle handle, int16_t size);`
- Deallocate a memory region and return it to an allocatable memory: `int16_t free(Handle handle);`

The `MemAlloc.allocate` function is split-phase, the following events had to be defined:

- Signal when an allocation requests has completed: `result_t allocComplete(HandlePtr handle, result_t success);`
- Signal when a reallocation request has completed: `result_t reallocComplete(Handle handle, result_t success);`

In order to use the *TinyAlloc* component, you need to make sure that the proper flags are set in your Makefile: `PFLAGS += -I%T/lib/Util`.

3.3.3 *TinyAlloc*

TinyAlloc implements a managed heap through a simple array of handles. Its size can be set, but it is fixed at runtime (the number of memory chunks that can be allocated is limited). The memory regions are referenced indirectly by another array, *i.e.,* the returned value of `allocate` only contains a pointer to this intermediate array. In other words, to access the memory, double dereferencing is needed. This technique allows more flexibility to *TinyAlloc*.

3.3.4 An Example

We clarify what double referencing entails for the programmer. Let us look at this example where memory is allocated and freed (only relevant code shown):

```
typedef struct
{
  uint8_t number;
  float value;
}ArrayEntry;

Handle array;

call MemAlloc.allocate(&array, 2*sizeof(ArrayEntry));

event result_t MemAlloc.allocComplete(HandlePtr h, result_t result)
{
  if (h == &array)                          // Must be the return for requested alloc
  {
    ArrayEntry *entry = (ArrayEntry*)(*array+sizeof(ArrayEntry));
    entry->number = 1;
    entry->value = 4.75;
  }
}

call MemAlloc.free(array);
```

4 Analog to Digital Converters

The MoteIV Tmote Sky has 6 pin and 10 pin expansion headers that are configurable so that additional devices (microphones, sensors, displays, inputs) may be added and controlled by the motes. Among these 16 pins are 6 ADC inputs, as well as an Analog VCC and Ground. The designated pins for the two headers can be found on page 21 of the Tmote Sky datasheet. These 6 ADC inputs have access to 6 12-bit ADCs on board the MSP430 microcontroller on the mote. Along with these ADC inputs, there are several sensors already attached to the mote, including a Total Solar Radiation (TSR) sensor, and Photosynthetically Active Radiation (PAR) sensor, a Humidity sensor, and a Temperature sensor. These are connected to internal ADC's which are readily accessible.

4.1 ADC Inputs - Sample Code

The ADC Inputs to the motes require a 0V to 3V input, which is then quantized to 12 bits. The sampling rate is adjustable. Details about the ADCs can be found on the Texas Instruments MSP430 Datasheet. Most devices will require some signal conditioning in order to be amplified or shifted to the correct input levels. The best way to debug or verify ADC code is to connect a signal generator with a 0 to 3V signal directly to the ADC pin.

The following is an example application called *RemoteDesign*. This application will take an input from the ADC pin, convert it to digital samples, and packetize the data to be sent out over the radio. The full *RemoteDesign* code can be found in *CD/nesC Programs/RemoteDesign* or by contacting Ryan Aures or Matt Holland.

4.1.1 Configuration File - *RemoteDesignC*

In the configuration file, several lines must be added to the implementation. The MSP430 component must be added to the components list, and wired to `Main.StdControl`:

```
components MSP430ADC12C;

Main.StdControl -> MSP430ADC12C;
```

The module file, which we will investigate next, will be called *RemoteDesignM*. The ADC component in the module file must be wired to an instantiation of the MSP430 ADC component. There are two ways to call the ADC, single and multiple. The single interface gets one sample at a time from the ADC, and will be used in this example. The multiple interface fills a buffer with samples and then returns the buffer. For high data rate applications, the multiple interface should be used.

```
RemoteDesignM.ADC1 -> MSP430ADC12C.MSP430ADC12Single[unique("MSP430ADC12")];
```

4.1.2 Module File - *RemoteDesignM*

In the module file, we will call the ADC, and define properties such as sampling rate, reference voltage, and sample and hold time. We will then packetize the data and send it over the radio. We must first provide an interface for the ADC:

```
interface MSP430ADC12Single as ADC1;
```

ADC1 is the name of the interface we will use from now on to call the ADC. Notice it has the same name as that referenced in the component file. We will now move into the implementation section of the module file, where we define the ADC's actions. We will first call *ADC1* in the `StdControl.init` command:

```
call ADC1.bind(ADC12_SETTINGS(INPUT_CHANNEL_A0,
                              REFERENCE_AVcc_AVss,
                              SAMPLE_HOLD_384_CYCLES,
                              SHT_SOURCE_ADC12OSC,
                              SHT_CLOCK_DIV_1,
```

```
                                    SAMPCON_SOURCE_SMCLK,
                                    SAMPCON_CLOCK_DIV_1,
                                    REFVOLT_LEVEL_1_5));
```

The preceding section of code initializes the ADC determines which pin we are using along with the reference voltage, the sample and hold time, the source clock, and several other properties. The definition and options available can be found in: *cygwin/opt/tinyos-1.x/tos/platform/msp430/MSP430ADC12.h*. There are two ways to acquire data from the ADC. By using:

```
call ADC1.getData();
```

the ADC will retrieve data once, and will not be called again. This call may be triggered by a timer in order for it to sample continuously, or the second option could be used:

```
call ADC1.getDataRepeat(500);
```

This will call the ADC every 500 clock cycles, based on the clock chosen in the bind operation. Since *SMCLK* runs at about 1MHz, this line will call the ADC every 0.5ms. Once the ADC has a full 12 bit sample, it will trigger an interrupt, and the *dataReady* event will be executed. The data arrives in the 16 bit word *data2*. Since the ADC sample is only 12 bits and arrives in the 12 MSB, the data is shifted to be LSB. The data is then put into the structure for our message, and loaded into the buffer *pack*. The if statement checks to see if our buffer is full. If so, it posts the *dataTask* and resets the buffer counter. If not, it waits until we can put the next sample in the buffer.

```
async event result_t ADC1.dataReady(uint16_t data2)
{
    atomic
    {
        uint8_t data;
        data = (data2>>4);
        pack = (struct bpMsg *)msg[currentMsg].data;
        pack->data1[buffercounter++] = data;
        if ((buffercounter) == BUFFER_SIZE)
        {
            post dataTask();
            buffercounter = 0;
        }
    }
    call Leds.yellowToggle();
    return SUCCESS;
}
```

Notice the `post dataTask()` line. When this task is posted, it starts executing right after *dataReady* completes. In this case, the task being called will send the data buffer over the radio. The red LED is flashed to give some visual indication that the packet was correctly sent.

```
task void dataTask()
{
```

```
    call Leds.redOn();
    if (call DataMsg.send(TOS_BCAST_ADDR, sizeof(struct bpMsg), &msg[currentMsg]))
    {
        call Leds.redOff();
    }
}
```

4.2 Internal ADC - Sample code

Along with the ADC input are several internal ADC's which are connected to the Tmote Sky's on board sensors. These internal ADC's are easier to interface with than the ADC input pins, and useful for simple applications such as temperature monitoring. The program *runner* involves taking temperature readings and sending them out over the radio. The full *runner* code can be found in *CD/nesC Programs/Range/runner* or by contacting Ryan Aures or Matt Holland.

4.2.1 Configuration File - *runner*

The configuration file requires a few additions to access the Hamamatsu temperature sensor. Under this implementation, a component must be added, and wired to **StdControl**:

```
components HamamatsuC;

Main.StdControl -> HamamatsuC;
```

The name of the temperature component in the module file (here just *RunnerM*) must then be wired to the Hamamatsu configuration file:

```
RunnerM.Temperature -> HumidityC.Temperature;
```

4.2.2 Module File - *runnerM*

In the module file, the following interface must be provided:

```
uses interface ADC as Temperature;
```

The temperature sensor returns a 14-bit value as opposed to a 12 bit value as previously seen. The 14-bit value is returned in the 16 bit integer *data* once the `dataReady` interrupt is thrown. The information can then be stored in another variable, *temp*. Note that for most of these sensors, the actual measurement is not returned, but must be adjusted to get a meaningful value.

```
async event result_t Temperature.dataReady(uint16_t data)
{
    // Temperature returns 14-bit value
    // actual_temp = -39.60 + 0.01*temp
    temp = data;
    return SUCCESS;
}
```

The 16 bit integer *temp* can now be inserted into a package for sending out, or checked for some value. For details, look in *runnerM.nc*

5 Deluge

Deluge is a utility that provides an easy way to program multiple nodes wirelessly. The program can be loaded onto a single node and use epidemic propagation to disseminate it throughout the network. This can be extremely useful when designing large networks.

5.1 Using Deluge

When a program is compiled using the `make` command a `tos_image.xml` file is created in the build directory. Deluge allows this image to be stored in the flash memory of the mote. The image is then automatically transmitted to any node whose current application has the *DelugeC* component in its current program. If multiple images are injected into the flash, then the user can issue the command to reboot to any of the available images.

It is important to mention that the required voltage for programming the motes is significantly higher than the normal operating voltage. The required voltage to program is 2.7 V while the operating range of the motes is 2.1 V - 3.6 V [6]. If the voltage is too low to reprogram and the reboot command is called, the red LED will blink three times and then reboot the current program.

5.2 Implementation

To make a program compatible with deluge, add the deluge component to your programs configuration file and give it the standard control interface. A section of code is shown below, the required additions are marked with comments:

```
components Main
        , DaprM
        , TimerC
        , GenericComm
        , LedsC
        , DelugeC                   //this is required for Deluge
        , CC2420RadioC
        ;

Main.StdControl -> DelugeC;         //this is required for Deluge
Main.StdControl -> GenericComm;
Main.StdControl -> TimerC;
Main.StdControl -> DaprM;
```

These are the only modifications to the program that are required. Setting up Deluge on a network is fairly easy, but tedious process. For an exact step by step instructions on formatting each node to work with deluge see [7].

6 Lab Work

It is now time to program the motes and apply everything we have seen so far. First, we will see how to send packets (and blink LEDs at the same time) and second, how to receive and handle

them. You should have already received / downloaded the file Pinger.zip.

6.1 Pinger

Pay attention to Pinger.nc: this is the configuration file that allows wiring of the components. Pinger has a main implementation in *PingerM* that uses timers, the radio, and LEDs. Consequently, the configuration files declares the components *TimerC*, *GenericComm*, and *LedsC*. For now, *Main* can be understood as the "conductor" component that initializes and starts other components. **StdControl** is the interface through which components can be controlled.

PingerM uses three interfaces: **Timer**, **SendMsg** and **Leds**. As the configuration file indicates, these are provided by other, clearly identifiable, components. In TinyOS-1.x, each timer must be declared with the `unique` codeword to signify that timers in the protocol stack are indeed different from one another. **SendMsg** is defined with a parameter "`AM_PINGMSG`": this way, the OS knows to route packets of this type to *PingerM*.

PingerM declares using the three interfaces mentioned above, and providing one interface **StdControl** to initialize, start and stop the execution of the component. The rest of the code should be fairly easy to understand for C programmers. The `TOS_Msg m_pmsg` is the full packet itself, while `PingMsg_t *pingmsg` is a pointer to the payload area of `TOS_Msg`. In this part of the memory, we can define custom fields such as `type`, `run`, `nodeID` etc. For a timer to repeatedly go off, the call to `PingTimer.start(TIMER_REPEAT, 900);` is entered.

Finally, let us dissect the following:

```
call SendPingMsg.send(TOS_BCAST_ADDR, sizeof(PingMsg_t), &m_pmsg);
```

`SendPingMsg.send` takes the next-hop destination address, the size of the payload to send, and the pointer to the packet as arguments. `TOS_BCAST_ADDR` is the broadcast address.

Compile and load the program onto one mote. Now, compile and load the TOSBase program onto another mote, but keep it plugged into the computer. In Cygwin type `listen`. When the mote starts sending packets, they should be picked up by TOSBase and displayed.
1. Make sure other groups are running the program. Can you see your packets? Are the packet ID numbers consistent?

In both the Makefiles of Pinger and TOSBase, uncomment or add `DEFAULT_LOCAL_GROUP = 0x1A` and pick a number other than 0x1A (for instance, your position in the classroom, row first and column second). Recompile and reload.
2. Make sure other groups are running the program. Are all the packets from your mote, and only your mote?

The `DEFAULT_LOCAL_GROUP` defines what "group" your mote belongs to: packets can only be sent and received within the same group. Make sure that all your motes run with the same group number or they will not be able to properly communicate. This, however, does not mean that your radio is not receiving *other groups' packets*, but merely that they are being dumped and not presented to upper layers.

6.2 PingerRx

Next, we let the second mote receive and handle the packets. *PingerRx* uses the **ReceiveMsg** interface, which is defined much like **SendMsg**. The difference with *Pinger* is in the implementation of **ReceiveMsg**. `ReceiveMsg.receive` simply takes the pointer to the received message, and ALWAYS returns a pointer, whether the same pointer or a new one.

Change Makefile and recompile so as to run the programm in TOSSIM.

Install the program on two motes, and start them. Make sure one of them has a nodeID of 1. The two motes should be playing ping-pong.

6.3 Try Your Own Modifications

Create two different types of PingMsg: types 1 and 2. If a receiver hears a ping message of type 1, it must toggle its blue light, and send a message of type 2 after one second. If a message of type 2 is received, a receiver must toggle its green light and send a message of type 1. The PingMsg must use the same "port" `AM_PINGMSG`.

Simulate in TOSSIM first, and install on the motes afterwards.

References

[1] Philip Levis, "TinyOS Programming—Chapter 4, Interfaces and Modules", Jun. 2006.

[2] Philip Levis et al., "Ad-Hoc Routing Component Architecture", Feb. 2003.

[3] Philip Levis and Nelso Lee, "TOSSIM: A Simulator for TinyOS Networks", Sep. 2003.

[4] Cory Sharp, "Timers", Sep. 2004.

[5] Robin Züger, "Paging in TinyOS", Aug. 2006.

[6] MoteIV Corporation, "Tmote Sky: Data Sheet", 6 Feb. 2006, page 4, *found in March 2006 at http://www.moteiv.com/products/docs/tmote-sky-datasheet.pdf*

[7] Jonathan Hui, "Deluge 2.0 – TinyOS Network Programming", *found in Feb. 2007 at http://www.cs.berkeley.edu/ jwhui/research/deluge/deluge-manual.pdf*