TinyOS: Getting Started

Philip Levis

March 27, 2002

Introduction

This document is intended to provide an introduction to TinyOS developers. TinyOS is an event-based operating system intended for use in sensor networks. TinyOS uses a programming model that is based on the concept of 'wiring' software components together to produce a working program.

The TinyOS programming model places requirements on how programs are written. One issue in sensor networks is that TinyOS might have very limited resources available (e.g. 512 bytes of RAM); this requires very efficient resource utilization. Another is the wiring metaphor, which requires being able to map a single function call (an output wire) to multiple functions being called (input wires). In addition, TinyOS uses C preprocessor macros heavily to allow alternative compilation modes (such as simulators). For these reasons, aspects of the programming environment can can be very confusing at first.

Information on how to download, compile, and install TinyOS is provided in the many user resources, and is not provided here. This document provides information on how to write TinyOS programs from scratch.

Component Model

In TinyOS, the distinction between the OS and applications is merely a semantic one; the two are compiled together and run in the same address space. As application components are tested, become more stable, and grow to be components used by other applications, they can transition to system components. One example of this is ad-hoc routing, which started as an application and is now a system component. TinyOS components are composed into a directed acyclic graph; the "top" of the graph is composed of application level components, while the "bottom" of the graph is composed of components that sit directly on top of hardware.

TinyOS abstractions: events, commands, and tasks

TinyOS presents three abstractions of computation: events, commands, and tasks. Commands represent downcalls in the component graph; a component calls commands on components lower than it in the component graph. Events are upcalls in the component graph;

a component signals events to components higher than it in the component graph. Event handlers may call commands, but commands may not signal events. Events are used heavily for split-phase execution; if a component calls a command that can take a long time (e.g. send a packet), the callee returns a status value on whether the command was successful, then later signals an event when the command is completed. The third abstraction, tasks, are a mechanism for asynchronous, long-running computation. Tasks run synchronously in respect to one another (are never preempted), but can be preempted by events. Therefore, if a TinyOS task has real-time requirements (e.g. is posted every 100 ms and must complete within 40ms of being posted), tasks must not execute for a long period of time.

As events are often the result of interrupts (and have can correspondingly have interrupts disabled), it is critical that they be as short as possible. Similarly, as commands can be called by events, they must be short to ensure an event does not run for a long time.

For example, the `SEC_DED_RADIO_BYTE` component, which encodes and decodes radio packets for error correction, uses TinyOS tasks for encoding and decoding. When the component's command to transmit a byte is invoked, it stores the byte in its frame, posts an encoding task, and returns. Unless there are other tasks waiting to run, when the current path of execution completes TinyOS will start executing the task. When the encoding task completes, it stores the encoded data in the component's frame. All this while, the lower level radio component has been firing events on the `SEC_DED` layer, telling it when it can transmit bits (basically, radio clock interrupts), but `SEC_DED` hasn't been ready. After the encoding is finished, the next time the event is fired `SEC_DED` takes an encoded bit and then issues a command to the radio layer to transmit it.

Component Files

TinyOS components are composed of three parts, a `.c` file, a `.comp` file, and `.desc` file. Note that there is no `.h` file; these are automatically generated at compilation from the `.comp`.

Component files (`.comp`)

Component files are similar to `.h` files in that they specify an interface to a component. Unlike a header file, which only specifies functions are defined, a component file also specifies the functions the component requires (must be defined elsewhere). In essence, it states the exported and the unresolved symbols of a component.

A component file is broken into a preamble and five sections. The preamble states the name of the component (which states what its `.c` and `.desc` are). The five sections are ACCEPTS, which are the TOS_COMMANDs that can be called, SIGNALS, which are the TOS_EVENTs that the component generates, HANDLES, which are the TOS_EVENTs that the component handles, USES, which are the TOS_COMMANDs it calls, and INTERNAL, which are C functions that should only be called internally by the component itself. This table summarizes the first four:

	Provides	Calls
Events	HANDLES	SIGNALS
Commands	ACCEPTS	USES

Here's an example `.comp` file:

```
TOS_MODULE AM_STANDARD;

ACCEPTS{
        char AM_SEND_MSG(short addr,char type, TOS_MsgPtr data);
        char AM_POWER(char mode);
        char AM_INIT(void);
};

SIGNALS{
        char AM_MSG_SEND_DONE(TOS_MsgPtr msg);
        TOS_MsgPtr AM_MSG_HANDLER_0(TOS_MsgPtr data);
        TOS_MsgPtr AM_MSG_HANDLER_1(TOS_MsgPtr data);
...
        TOS_MsgPtr AM_MSG_HANDLER_254(TOS_MsgPtr data);
        TOS_MsgPtr AM_MSG_HANDLER_255(TOS_MsgPtr data);
};

HANDLES{
        char AM_TX_PACKET_DONE(TOS_MsgPtr msg);
        TOS_MsgPtr AM_RX_PACKET_DONE(TOS_MsgPtr packet);
};

USES{
        char AM_SUB_TX_PACKET(TOS_MsgPtr data);
        void AM_SUB_POWER(char mode);
        char AM_UART_SUB_INIT(void);
        char AM_UART_SUB_TX_PACKET(TOS_MsgPtr data);
        void AM_UART_SUB_POWER(char mode);
        char AM_SUB_INIT(void);
};
```

This component is the active messaging layer in TinyOS, which is generally used for all radio-based communication. The 256 different `AM_MSG_HANDLER` functions are part of the AM system; every AM packet has a single byte type value in its header. Different applications or transport protocols can use their own handler numbers, allowing two separate networks to work concurrently without worrying about applications accidentally handling a different program's packets. Components can also have a a `IMPLEMENTED_BY` or `JOINTLY IMPLEMENTED_BY` directive after the `TOS_MODULE` declaration. The presence of these determines whether the component has a `.c`, a `.desc`, or both.

Directive	`.desc`	`.c`
none		•
IMPLEMENTED_BY	•	
JOINTLY IMPLEMENTED_BY	•	•

The AM layer only accepts three `TOS_COMMAND` calls, the first for sending a message, the second for setting a power level (say, to turn the radio off for power conservation), and the third for initialization.

Looking at the `SIGNALS` set, however, shows that, like most components in TinyOS, sending an AM message is a non-blocking operation. Instead, when the message send is completed, `AM_TX_PACKET_DONE` is fired. An application can register an event to be triggered when this happens, so it knows when it can send another packet. The message handlers are signaled when a packet is received; if an application wants to receive a type of AM message, it registers a `TOS_EVENT` with the handler event.

The AM component handles two types of events, which are generated by a lower network layer. The AM layer isn't responsible for actually sending out or receiving packets; instead, it marshals packets (adding addresses, type values) for a lower layer to send.

There is no `INTERNAL` section in the AM component.

When writing a component, the `.comp` file is a good place to start, as one might start writing with a `.h` file in a standard C program.

Source files `.c`

`.c` files (source files) are where TinyOS code resides. Any new TinyOS functionality is written in a source file. Component files are in all caps (e.g. `AM.c`) while lower case files are code that is not part of any component, instead being an integral part of the system (e.g. `sched.c`, `heap.c`). It is possible to have a component that doesn't have a `.c` file; `GENERIC_COMM` is one such component; it takes a group of components that form a complete network stack and just links the components so they'll work properly, introducing no new code.

In addition to an interface, components also have a fixed size storage frame. One should never declare global variables; instead, one should declare them in a frame. This is the active messages layer frame:

```
#define TOS_FRAME_TYPE AM_obj_frame
TOS_FRAME_BEGIN(AM_obj_frame) {
    char state;
    TOS_MsgPtr msg;

}
TOS_FRAME_END(AM_obj_frame);
```

This frame has only two fields; the `TOS_MsgPtr` is set when a user sends a message. As messages are not sent synchronously (a task is enqueued to send it), the pointer must be stored after the call to `AM_send_task` completes. `state` stores whether the AM layer is currently transmitting a packet or not, which allows the AM layer to refuse additional transmission requests until the first one completes.

Variables in a frame are referenced with the `VAR()` macro. For example:

```
char TOS_COMMAND(AM_POWER)(char mode){
    TOS_CALL_COMMAND(AM_SUB_POWER)(mode);
    VAR(state) = 0;
    return 1;
}
```

As one can see from these examples, TinyOS components use preprocessor macros heavily. Calling a `TOS_COMMAND` requires using `TOS_CALL_COMMAND`, and calling a `TOS_EVENT` requires using `TOS_SIGNAL_EVENT`. Most of these macros are defined in `tos/include/tos.h`, but a few (such as those pertaining to hardware signals and interrupts) are defined in `tos/include/hardware.h`. `TOS_SIGNAL_HANDLER`s disable interrupts during their execution, while `TOS_INTERRUPT_HANDLER`s leave them enabled.

Description Files `.desc`

The `.desc` file is where components are linked together in a manner similar to a circuit diagram. Description files are often the source of compilation errors, as mistakes in the file can be very difficult to find. Be careful when writing a `.desc` file.

A simple explanation of a description file is that it basically `#define`s one function name to another. If one looks at the `super.h` generated for an application (in `gensrc`), there's a long list of `#define` statements. For example, the entry

```
AM_STANDARD:AM_MSG_HANDLER_7 BLESS:DATA_MSG
```

in `BLESS.desc` (an ad-hoc routing component) can result in the line

```
#define AM_MSG_HANDLER_7_EVENT DATA_MSG_EVENT
```

in an application's `super.h`.

Description files actually do much more than simple aliasing. First, they collapse `.desc` aliasing to a single name. If component A aliases something component B accepts, which is merely something that component B aliases to a function component C implements, then component A's name will be aliased to component C's. This is very useful in the situation of components such as `GENERIC_COMM`, which have no code, instead just linking and presenting a unified interface to several components.

Aliasing becomes a bit complicated with regards to `USES` and `HANDLES` directives. Consider `MAIN`, the component that boots TinyOS. `MAIN.comp` names a few commands that it uses. These commands are included from proper initialization of components at boot. For example, `AM_STANDARD.c` needs `AM_INIT` to be called before messages are sent. The way this is accomplished is by aliasing `AM_STANDARD:AM_INIT` to `MAIN:MAIN_SUB_INIT` in a `.desc`.

```
USES{
char MAIN_SUB_INIT(void);
char MAIN_SUB_START(void);
char MAIN_SUB_POT_INIT(char val);
};
```

However, what if there are multiple components that need to be initialized? That is, what if multiple `INIT` commands need to be called when `MAIN.c` calls `MAIN_SUB_INIT`? The TinyOS compilation tools handle this case for you. If a series of `.desc` entries reference multiple implementations (things defined in `ACCEPTS` and `HANDLES`), then compilation will automatically generate a dispatch function that will result in all of the events or commands being processed properly. Their order of execution is undetermined, as they are assumed to be independent.

For example, the application `bless_test`, which is a simple test of the BLESS ad-hoc routing protocol, has two required initializations. The first is the radio (RFM) and the second is a random number generator. In `bless_test`'s `super.h`, this function is defined:

```
char RFM_INIT_COMMAND(void);
char LFSR_INIT_COMMAND(void);
static inline char dispatch__NET_32_COMMAND(void){
char retval;
retval = RFM_INIT_COMMAND();
retval = LFSR_INIT_COMMAND();
return retval;
}
```

One important role that `.desc` files fill (besides event/command multiplexing, as described above) is that they allow compilation optimizations. Additionally, the combination of `.comp` and `.desc` files allows compile time checking of the restrictions placed on TinyOS command execution and event firing (commands can only be called down the component hierarchy, while events can only be fired up).

There is currently a bug in the TinyOS `.desc` processing utilities. Specifically, if command A is aliased to commands B and C, and command B is aliased to command C, then calling B will result in calling B and C (because of A). This results in an infinitely recursive call when B is called.

Example TinyOS Component: `ROUTER`

`ROUTER` is a beacon-based ad-hoc routing protocol that communicates data along a routing tree to the base station. The protocol works by transmitting a beacon out from the base station. As each node receives the beacon, it sets the current route (or parent) to the base station. After updating its parent point, the node rebroadcasts the beacon. If a parent is not heard from after a given number of clock ticks, the node waits for the next beacon and sets that beacon's source node as its parent. When a `ROUTE_UPDATE` message, or beacon, is received, the following logic is performed:

```
TOS_MsgPtr TOS_MSG_EVENT(ROUTE_UPDATE)(TOS_MsgPtr msg){
    short* data = (short*)(msg->data);
    TOS_MsgPtr tmp;

    //if route hasn't already been set this period...
    if(VAR(set) == 0){
        //update the parent
        VAR(route) = data[(int)data[0]];
        // this parent should time out if we don't hear
        // from it after 8 clock ticks
        VAR(set) = 8;
        // update the number of hops
        data[0] ++;
        //create a update packet to be sent out.
        data[(int)data[0]] = TOS_LOCAL_ADDRESS;

        //send the update packet.
        if (VAR(route_send_pending) == 0){
          TOS_CALL_COMMAND(ROUTE_SUB_SEND_MSG)
            (TOS_BCAST_ADDR,AM_MSG(ROUTE_UPDATE),msg);
          VAR(route_send_pending) = 1;
          return msg;
        } else{
          return msg;
        }
    }
    return msg;
}
```

The `ROUTER` component (also known as `AM_ROUTE`) is easy to use from the perspective of the user. It provides only three functions to be called and sets up the routing tree automatically.

```
ACCEPTS{
        char AM_ROUTE_INIT(void);
        char AM_ROUTE_START(void);
        char ROUTE_SEND_PACKET(char* data);
};
```

`INIT` and `START` are both invoked at node boot. An application can send a packet by calling `ROUTE_SEND_PACKET`. If the node cannot currently send data through the network, it will return failure to the calling component.

The `AM_ROUTE` component also handles four event call-backs:

```
HANDLES{
        TOS_MsgPtr ROUTE_UPDATE(TOS_MsgPtr data);
        TOS_MsgPtr DATA_MSG(TOS_MsgPtr data);
        void AM_ROUTE_SUB_CLOCK(void);
        char ROUTE_SEND_DONE(TOS_MsgPtr data);
};
```

.

The first is called when the node receives a beacon; the `AM_ROUTE` layer sets its parent in the tree. The second is called when a node receives a data message from a node upstream. The packet is resent by the current node to its parent. The third is merely a system clock call-back which is used for network timeouts (if a mote hasn't heard its parent for a while it selects a new parent). The last is merely for notification of packet send completion, so `AM_ROUTE` knows when it can send its next packet. These are all call-backs provided by the `AM_STANDARD` (Active Messages) component. An application can also alias its own clock function to `AM_ROUTE_SUB_CLOCK`; the application can then send out data on the same time scale (in terms of timeouts, etc.) as `AM_ROUTE`.

`AM_ROUTE` also uses a large number of commands in sub-components for blinking the LEDS, etc.

The `ROUTE_UPDATE` event is called by the underlying `GENERIC_COMM` component used to interface with the RFM radio. In the `apps/router/router` directory is the description file, `router.desc`. Notice that `ROUTE_UPDATE` commands are called by the `GENERIC_COMM` message handler number 5.

```
GENERIC_COMM:GENERIC_COMM_MSG_HANDLER_5 AM_ROUTE:ROUTE_UPDATE
```

Also note that this component includes the following other components:

```
include modules{
        GENERIC_COMM;
        MAIN;
        AM_ROUTE;
        CLOCK;
        LEDS;
};
```

`MAIN` is a component required of all TinyOS applications. `AM_ROUTE` is a reference to the router application itself. `GENERIC_COMM` provides all the underlying communications support including an interface to the RFM radio and methods to send packets over the UART and into a host PC. The `CLOCK` and `LEDS` components are used to trigger clock events and controls the LEDs on the node.

Writing Your Own TinyOS Components

The easiest way to start writing TinyOS code is to take existing code and modify it slightly. For example, take the `AM_ROUTE` application and change the clock event command such that each node sends a packet to the base station during each clock tick. One could also modify `AM_ROUTE` and add a sensor component to it, so that it sends sensor data to the base station. Once you have a grasp of the different file formats, you can try implementing system components, such as a more efficient network routing scheme that combines multiple children's data and sends them in one packet instead of forwarding every packet up the network.