TinyOS
An operating system for microcontroller platforms

Sandro Degiorgi
71desa1bif@hft-stuttgart.de

Stuttgart University of Applied Sciences
A contribution to seminar2, summer term 2009
Supervision by Prof. Dr. Stefan Knauth

Abstract: TinyOS is an open source component based and event driven operating system designed for microcontroller platforms. It was initially released in 2000 by UC Berkely in cooperation with Intel. By now TinyOS is the de facto standard targeting wireless sensor networks. There are several hundred developers involved worldwide having ported TinyOS to over a dozen platforms [Tin04]. TinyOS offers features to address the constraints that come with sensor networks. These are above all the limit of memory, slow cpu speed and the inherent shortage of power. TinyOS' rich component library offers e.g. network communication, testing (RSSI), attachment of different sensory components and debugging features to enable developers to deploy seamless and fast. This seminar paper will give a thorough explanation of TinyOS. The following chapters will talk about application development with TinyOS (in nesC - network embedded systems C) and the TOSSIM simulation framework. TOSSIM is a very useful tool when it comes to testing and debugging a wireless sensor network - in fact debugging is a big topic in this field.

Introduction and Motivation

Terms like *pervasive computing*, *everyware*, the *Internet of Things*, *ubiquitous computing* and similar buzzwords have become ubiquitous to us. But what exactly do they mean ? In the "early days" there was one computer for many persons. As said by - IBM says it's a misquote - T.J. Watson 'I think there is a world market for maybe five computers' ([IBM07], p. 25), CEO of IBM in the 1950s. People would be using the computers for heavy duty calculations and paying for the CPU time used (mainframe era). Then the ratio changed to somewhat one computer per person (desktop era). Nowadays we have a tendency towards many computers per person. And these computers, or let's better call them devices, get smaller and smaller and begin to interconnect and interact with each other. Even more, they begin to disappear inside everyday objects. The following words by M. Weiser appeared 1991 in the Scientific American (see [Wei91]):

> "The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it."

And that's what the above terms refer to. A future of more and more self-configuring interconnected devices, which are literally everywhere, not recognisable as a computer anymore, suitable for all kinds of uses and helpers. Building and office automation, assistance systems, intrusion detection, localisation and whatnot imaginable. Rough estimates talk about 1000 - 5000 devices per person, that would go up to 100 billion devices newly connected to the internet - the *Internet of Things*.

Walking down the path to small intelligent devices all around, several problems must be solved. As for the well known Moore's Law (even the refined version in [Int05]), we estimate everything gets double-fast and half-sized every two years. But that's not true for the energy per instruction ratio. In fact it costs more energy per instruction over time. To address this energy issue these devices are to use low energy microcontrollers and newly designed operating systems. The de facto standards here are Atmel and Texas Instruments microcontrollers running an operating system called TinyOS.

TinyOS is an open source component based and event driven operating system particularly designed for microcontroller platforms. It was initially released in 2000 by UC Berkely in cooperation with Intel. TinyOS offers the needed features to adress the constraints that come with sensor networks. This seminar paper will give a thorough explanation of TinyOS. Chapters 2 and 3 will talk about application development with TinyOS, and ways to test, simulate and visualize sensor networks.

1 TinyOS at a glance

1.1 A word about wireless sensor networks

Wireless sensor networks (WSN) were originally developed by the military for uses like ad hoc battlefield surveillance. WSN consist of autonomous self organizing devices. Nowadays WSN are already used in several industrial and civilian applications like monitoring, automation, assistance systems, environment monitoring, traffic control and many more.

1.2 TinyOS

Figure 1: TinyOS Logo

TinyOS is an open source operating system designed for microcontroller systems. Some may refer to it as pseudo operating system or development environment only. It minimizes overhead and enables developers to learn and deploy in a short period of time without digging too much into the hardware details. It takes the inherent constraints of sensor networks into account.

TinyOS has already been ported to several different hardware platforms, microcontrollers and sensor boards. It is used by different universities and companies to develop, test, analyze and simulate sensor network algorithms and protocols around the world.

”New releases see over 10,000 downloads. Over 500 research groups and companies are using TinyOS on the Berkeley/Crossbow Motes. Numerous groups are actively contributing code to the sourceforge site and working together to establish standard, interoperable network services built from a base of direct experience and honed through competitive analysis in an open environment” (see [Tin04]).

1.3 Constraints

There are several bottlenecks one has to deal with in the area of wireless sensor networks. This section wants to name at least the three most important:

CPU Speed is limited with microcontrollers. As mentioned above, following Moore’s Law gives hope that in a near future it is possible to use fast(er) 32bit processors. As for now mostly 8bit microcontroller devices are used. Manufacturers to name in this field are Atmel and Texas Instruments.

Memory RAM is always short. Chips get smaller and smaller, and the smallest gets to be the most expensive. In the field of research, price is not that important, but when it comes to selling first products, it is common to have only around several hundred kB of RAM available.

Energy The bottleneck #1. The sending and recieving of radio packets cost a lot of energy. Even the ”waiting” for incoming packets is very energy consuming. Most of ad hoc sensor networks need to run a long time though. So an intelligent energy management is mandatory.

1.4 What’s so different with TinyOS ?

- No HAL and no kernel, so direct hardware access
- No process management, so one task at a time
- No memory management, so single linear memory
- No dynamic memory allocation, so static allocation at compile time
- No software calls or exceptions, so function calls / events

1.5 TinyOS Components and Interfaces

Components in TinyOS should not be thought of as a component in an object oriented programming language - same name, different meaning.

TinyOS already offers a rich component library. Developers can reuse existing components or integrate them in their own work. Some examples for such prewritten components are timers, the ability to do data transfers, the use of network protocols up to different drivers for sensory units. In TinyOS components are either modules or configurations. Interfaces are used to group components.

Components A module holds the written code. It is the implementation of functionality. Configurations stitch components together, defining the intended way how they should interact and work together.

Interface An interface is a collection of functions. With interfaces it is easy to group functions or when it comes to defining the ”uses” of a component (the functions the component is about to use) it reduces the effort to maintain, extend and use in implementation code.

1.6 Split Phase Programming

Split Phase programming is an asynchronous model. When it comes to the variety in sensor nodes hardware it is important to have a ”flexible hardware/software boundary” ([Lev06], p.27). The two phases are the signaling of an event and the actual work. This way no data or information is lost, since incoming information gets noticed immediately. The interupt handler tries to be very short. It just does the minimal work, sets a flag and returns. Then, when there is time or some scheduler fires, the flag indicating that there is still work to do is detected, and the work is finally performed.

1.7 Event driven execution model

”TinyOS’s event-driven execution model enables fine-grained power management yet allows the scheduling flexibility made necessary by the unpredictable nature of wireless communication and physical world interfaces” (see [Tin04]). One may think of it as the paradigm when it comes to GUI programming.

1.8 nesC

The development in TinyOS 2.x is done in network embedded system C (nesC) (see [Tin04]). It’s a ”component based C dialect”.

2 Application development with TinyOS

By now TinyOS is in its 2.x days. There is not much to no compatibility between the 1.x and 2.x versions of TinyOS. There are so called TinyOS Enhancement proposals (TEP), ”1 - 100 are BCP (Best Current Practices) and 101+ are informal , Documentary or Experimental” ([Tin07], [Tin08]). The ”core” enhancements are all finalized at the time of writing, and already several 101+ TEPs are as well (see [Tin08]).

If not noted differently this entire chapter uses information off [Lev06] by Philip Levis, the guy behind TinyOS 2.x, TOSSIM and Mate (Berkeley [Lev05], now Junior Professor at Stanford [Lev09]).

2.1 Namespaces

The usage of nesC is quite similar to the usage of C. Problems mostly arise when it comes to incorporating new code into existing one. The biggest difference is the nesC linking model, that is combining the written or to be used components into a working application.

In C a global namespace is used for variables and functions. Of course there are different even hierarchical scopes and one can use *static* to prevent the variable to enter the global namespace.

When breaking down a C source into several source files they can only reach each other through declarations in the global scope (#include...). But this creates unforeseen and unknown dependencies between the source files. One way to escape there is to use function pointers that bind at runtime (e.g. GUI buttons, Virtual File Systems on Windows). ”Shadowing” is quite legal in C, but this will also lead to problems, since it’s hard to find out in which scope the variable is found.

C++ is very similar to C with the extension one can use inheritance. So class methods and members do not enter the global namespace. This way functionality can easily be extended without the binding issues, which lead to factory patterns to break down the coupling between files and classes even more.

When it comes to decoupling, Java has similarities with C++. Java classes more often than C++ take objects as parameters, having the association occur at runtime (e.g. think of adding an actionListener rather than adding a click method on whatever mouse input field).

”Referencing a function requires referencing a unique name in the global namespace, so code uses a level of indirection - function pointers or factories - in order to decouple one implementation from another”([Lev06], p. 24).

2.2 And nesC ?

nesC components encapsulate state and functionality. Unlike C++ and Java objects though, which refer to functions and variables in a global namespace, nesC components use a purely local namespace. nesC components name the functions they implement (”implements”), but also the functions they call (”uses”) in a specification block.

In practice nesC components would use interfaces, that are ”collections of related functions”, rather than naming all ”uses” that may occour. For example in context with power management it is often important to turn something on and off. ”The StdControl interface is a common way to express this functionality” (see [Lev06], p. 23),

```
interface StdControl {
  command error_t start();
  command error_t stop();
}
```

So with this existing interface one could define an imaginary RoutingLayer:

```
module RoutingLayerC {
  provides interface StdControl;
  uses interface StdControl as SubControl;
}
```

Unless SubControl is wired to a provider a call on RoutingLayerC.SubControl.start() would lead to an undefined symbol error. This needs wiring to a completely decoupled provider.

```
module PacketLayerC {
  provides interface StdControl;
}
```

So with these two wired together RoutingLayerC.SubControl.start() would fire up PacketLayerC.start().

As shown nesC however breaks up code into ”discrete units of functionality”. These components can only access their local namespaces. The wiring of users on providers happens at compile time (no runtime allocation, no function pointers and no complex indirection - nesC knows all possible call graphs).

All of this static linking is only possible because there is no user input, no loading of additional software or addition of new and different hardware. It is embedded computers with specific hardware and uses. Another reason for this static approach is, that one can’t just restart or reboot it, these systems are meant to run unattended long-term.

2.3 Split Phase versus Blocking

Let's say there is need to incoporate different sensors. One may sensor-read all the time and only record changes for itself. One could easily call that sensor, retrieve information and go back to normal operation. The other may be a sensor where a conversion between analog and digital signals need to occur. So a call for that information would make the system literally wait for the result to come out of the ADC (Analog Digital Converter) - and this might take a while. In the meantime the mote would waste time, CPU cycles and energy just for waiting.

The traditional solution is threads. Threads though are rather RAM and CPU consuming - which both are bottlenecks to deal with on sensor nodes.

TinyOS' approach now is to make everything split phase, that is, asynchronous. Important on these split phase interfaces is that they are bidirectional. First phase calls for information on the component (down-call), then directly returns to normal operation. The called component will call back (up-call) when the data is ready. In nesC the down call is called "command" and the asynchronous return (up) call is an "event". In most cases interfaces already define both ways of communication.

```
interface SaturDayNight {
  command error_t orderBeer(message_t* brand);
  event void orderBeerDone(message_t*
              dummyNotice, error_t error);
  ...
}
```

Whether the component uses or provides an interface defines on "which side" the component resides - command or event.

Interfaces can take parameters, but the types of arguments when wiring users and providers must match. It is also common to use these parameters to distinguish between different implementations - the parameter itself is never used in the implementation though (e.g. timer).

On modules there is always an implementation block following the module definition. This is the place where one would set up variables and write functionality pretty similar to C. The following module and implementation could be used to do some sort of periodic reading of data.

```
module PeriodicReaderC {
  provides interface StdControl;
  uses interface Timer<TMilli>;
  uses interface Read<uint16_t>;
}
implementation {
  uint16_t lastVal = 0;
```

```
    command error_t StdControl.start() {
      return call Timer.startPeriodic(1024);
    }
    command error_t StdControl.stop() {
      return call Timer.stop();
    }
    event void Timer.fired() {
      call Read.read();
    }
    event void Read.readDone(error_t err, uint16_t val) {
      if (err == SUCCESS) {
        lastVal = val;
      }
    }
  }
```

2.4 Tasks

Problems may occur when signaling an event from within a command. Imagine some sensor collecting sensory data is called with the command *read()* ("give me your data"). This command then would signal out the event *readDone()* containing the data. Some module would then keep on calling *read()* until the buffer is read entirely. This would create a long call loop between the two - worst case leading into an overflowing stack, maybe crashing the application. The problem is the *read()* command signaling the *readDone()* event. There is obviously the need to schedule that command for later - this is where **tasks** come on stage. A module can *post* a task to the TinyOS scheduler.

```
  task void readDoneTask();
```

So rather than signaling the event of *readDone()* directly in the command, the command would rather *post* the *readDoneTask()* and return success for now. The *readDoneTask()* will signal the *readDone()* event and quit. Since the task is scheduled for later start and resides in the local namespace of the module there is no need for either return value or call parameters.

There is only one task running at a time in TinyOS so they **have** to return gracefully. There is no way to end a running task until it completes. So it is mandatory to sanitize tasks to end in any case within a very short time. Therefore long computations should be split up into several tasks - a task may even post itself again. On the other hand this way the task code can be quite simple, since there is going to be no interference with other tasks trying to get their hands on whatever module, hardware or distort the RAM.

2.5 I am eager now - where to start ?

For application development under TinyOS in nesC everything needed ”out of the box” is available from the official homepage. It makes sense to work under Linux, but there are solutions available for Windows environments as well (e.g. using cygwin).

There is an eclipse plugin available made public by the ETH Zuerich called YETI2, which, as for the release date, seems pretty up to date.

For this paper a virtual machine called ”xubunTOS” (based on Ubuntu) was used, which can be downloaded for free. It can be run on any operating system which is able to play VMware images. It contains everything needed from the compilers to the includes.

2.6 Skeleton - the obligatory ’Hello World’

To keep this paper in size this part was omitted. Please see [Tin05] for details. The Wiki there is a good source for beginners.

2.7 TinyOS/nesC condensed programming hints

To keep this paper in size this part was omitted. Please see [Lev06], p. 15-16 for details.

3 Simulation and Visualization

It’s with sensornets as with everything else. As long as everything runs smooth noone really cares that much. But what if it fails ? Debugging can be a drag, debugging sensor networks **is** a drag for sure. Did a node fail, crash, is the routing algorithm failing, is there interference with ”foreign traffic” ? It is very important to be able to simulate (to a certain degree) sensor networks. It’s not only debugging, it’s also testing, being able ro recreate an environment and the ability to ”play” with the scalability - not always twice as big is twice as good.

The visualization aspect is not only to see what is happening in the network, but it is also to interact with it on a visual level.

3.1 TinyOS Simulation (TOSSIM)

This section mainly presents information that can be found at [ea06].

TOSSIM enables the developer to actually debug the applications, digg into details on a bit level, reduce costs and time for bug-hunts and even repeat the failures found. It is also possible to interact with the network, be an actuator, inject radio packets and so on.

First is to simulate a mote - this could be attached to the USB port of the developers PC. But wouldn't it be reasonable to ask for the simulation of 10, 100 or even thousands of motes ? TOSSIM offers a framework to compile nesC code against. Then it is possible to start motes locally on a simulation platform.

TOSSIM is used to replace components like standard timers but is also capable to simulate "special hardware". One could for example have the need to simulate an Atmega128 hardware clock. TOSSIM allows to choose whether to be that specific or to use the standard component. Same goes for radio communication - it is possible to either simulate a packet-based communication or to go deeper and replace/simulate a special radio chip component.

TOSSIM uses a discrete event simulator that pulls events off the event queue and does the job. Again, depending on the level of detail, these events are high-level system events (like *Read.readDone()*) or "real" hardware interrupts to deal with. TOSSIM also allows to post tasks, which go as delayed events.

TOSSIM is a framework. So one must actually "write" the simulation, either in C++ or Python. There is support to transform code between the two. C++ helps to prevent "interpreter bottlenecks" but Python actually allows to interact "real-time" with the simulation.

TOSSIM still lacks a lot of reality when it comes to simulating different radio models and signal strengths, radio interference or batteries running low (see [Shn06]). There are empirical data sets, that can be used to simulate different radio models and topologies.

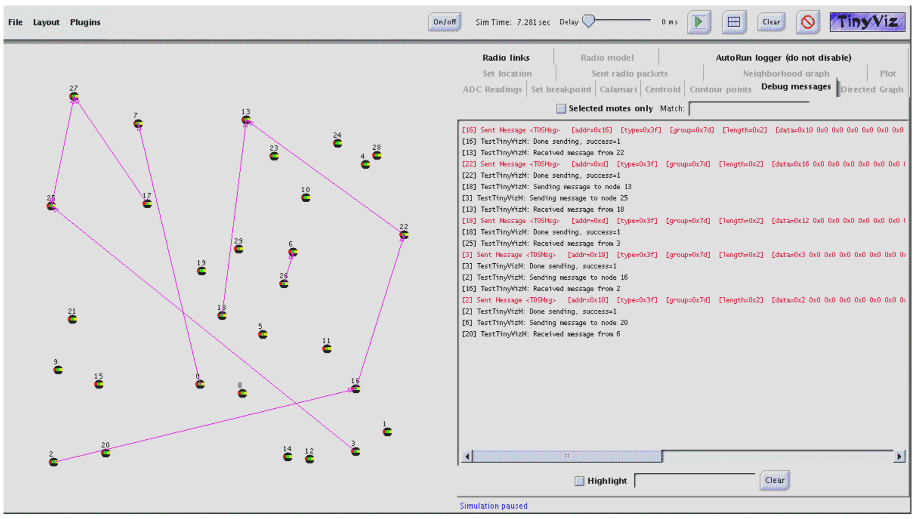
Also, due to the the discrete event simulator, there will never be a preemptive interrupt, which might make a mote go bluescreen. Also if some handler will run too long (might lead to memory squeeze), this will be no problem in the simulation environment.

The interaction with the network, like network monitoring and packet injection, is done via the SerialForwarder, which is the standard TinyOS interface tool. Available radio models are simple and lossy, where with the latter it is possible to define different bit error rates between the motes.

3.2 Visualization with TinyViz

TinyViz is a Java application for visualization of the simulated motes, sensor readings and radio links. It is customizable and extensible, so it is no hassle to add application specific features. There are many plugins available already (changing mote options, event subscription, send commands, asf.).

TinyViz offers the ability to simulate actuation in the sensor network and TOSSIM simulation - changing radio model and sensor readings. It offers a communication subsystem, an event bus and the ability to synch.



```
[16] Sent Message <TOSMsg>   [addr=0x16]  [type=0x3f]  [group=0x7d]  [length=0x2]  [data=0x10 0x0 0x0 0x0 0x0 0x0 0x0
[16] TestTinyVizM: Done sending, success=1
[13] TestTinyVizM: Received message from 22
[22] Sent Message <TOSMsg>   [addr=0xd]  [type=0x3f]  [group=0x7d]  [length=0x2]  [data=0x16 0x0 0x0 0x0 0x0 0x0 0x0 (
[22] TestTinyVizM: Done sending, success=1
[18] TestTinyVizM: Sending message to node 13
[3] TestTinyVizM: Sending message to node 25
[13] TestTinyVizM: Received message from 18
[18] Sent Message <TOSMsg>   [addr=0xd]  [type=0x3f]  [group=0x7d]  [length=0x2]  [data=0x12 0x0 0x0 0x0 0x0 0x0 0x0 (
[18] TestTinyVizM: Done sending, success=1
[25] TestTinyVizM: Received message from 3
[3] Sent Message <TOSMsg>   [addr=0x19]  [type=0x3f]  [group=0x7d]  [length=0x2]  [data=0x3 0x0 0x0 0x0 0x0 0x0 0x0 0x
[3] TestTinyVizM: Done sending, success=1
[2] TestTinyVizM: Sending message to node 16
[16] TestTinyVizM: Received message from 2
[2] Sent Message <TOSMsg>   [addr=0x10]  [type=0x3f]  [group=0x7d]  [length=0x2]  [data=0x2 0x0 0x0 0x0 0x0 0x0 0x0 0x
[2] TestTinyVizM: Done sending, success=1
[6] TestTinyVizM: Sending message to node 20
[20] TestTinyVizM: Received message from 6
```

Figure 2: TinyViz Screenshot

Summary

The *Internet of Things* is not just postulated but actually ante portas. Following Moore's Law we hopefully all will be still alive in a time when we are surrounded by thousands of little networkable nodes to help and assist us. We will have no more problems regarding assistance in our homes, calling help, localizing things and people, maybe ourselves in a "where is the next exit" or "how do i get to Villa Boghese" situation, up to driver assistance systems in our cars and trucks.

Many people in the field of wireless sensor networks say, that 90 percent of all effort will go into software development and just 10 percent into hardware. There is a lot work to be done here.

References

[ea06] Philip Levis et al. TinyOS Wiki. *http://docs.tinyos.net/index.php/TOSSIM*, 2006.

[IBM07] IBM. Frequently Asked Questions. *http://www-03.ibm.com/ibm/history/documents/pdf/faq.pdf*, 2007.

[Int05] Intel. Moore's Law. *ftp://download.intel.com/museum/Moores_Law/Video-Transcripts/Excepts_A_Conversation_with_Gordon_Moore.pdf*, 2005.

[Lev05] Philip Levis. Homepage at UC Berkeley. *http://www.cs.berkeley.edu/ pal/mate-web/*, 2005.

[Lev06] Philip Levis. TinyOS Programming Guide. *http://www.tinyos.net/tinyos-2.x/doc/pdf/tinyos-programming.pdf*, 2006.

[Lev09] Philip Levis. Homepage at Stanford. *http://csl.stanford.edu/ pal/*, 2009.

[Shn06] Shnayder. Power Loss Simulation with TOSSIM. *http://www.eecs.harvard.edu/ shnayder/ptossim/*, 2006.

[Tin04] TinyOS. TinyOS Mission Statement. *http://www.tinyos.net/special/mission*, 2004.

[Tin05] TinyOS. TinyOS Getting Started. *http://docs.tinyos.net/index.php/Getting_Started_with_TinyOS*, 2005.

[Tin07] TinyOS. TinyOS 2.x Documentation. *http://www.tinyos.net/tinyos-2.x/doc/*, 2007.

[Tin08] TinyOS. TinyOS Enhancement Proposals. *http://tinyos.stanford.edu:8000/TEP_review*, 2008.

[Wei91] Mark Weiser. The Computer for the 21st Century. *http://www.ubiq.com/hypertext/weiser/SciAmDraft3.html*, 1991.