EVENT DRIVEN SENSOR SYSTEMS

GROUP MEMBERS -->

1. CHETAN FEGADE 1205166348
2. NIKHIL MASCARENHAS 1204752753

MENTOR -->

Dr. YANN-HANG LEE

Submitted on December 11, 2012
INDEX

1. Introduction ............................................................................................................................................. 3
2. TinyOS ..................................................................................................................................................... 3
3. Middleware ............................................................................................................................................... 5
4. TinyGALS .............................................................................................................................................. 7
5. nesC ....................................................................................................................................................... 10
6. galsC ....................................................................................................................................................... 11
7. Future advances and Conclusion ........................................................................................................... 12
8. Commercial use of TinyOS .................................................................................................................... 12
9. References ............................................................................................................................................... 13
Introduction

In this project report we present to you our research work on an operating system that is specifically designed for event driven wireless sensor networks, “Tiny OS”. Also in this report we present our study on other aspects around Tiny OS like its programming model i.e “Tiny GALS”, middleware, galsC and nesC programming languages.

First of all we start with defining Wireless Sensor Networks (WSN). A WSN is a network of sensor nodes distributed in space in order to sense environmental changes, changes in location of objects, etc and report the findings to a center base station. The event driven approach qualifies such a WSN to react only when an event takes place. Next our report presents a detailed study of Tiny OS, is an embedded operating systems designed to suit the needs of event driven systems. Tiny OS, by itself cannot interact with hardware. It needs another software layer in between to be able to efficiently interact with the hardware. In our report we don’t concentrate on a specific middleware. Rather we have stressed upon the requirements of middleware. Tiny OS is written in nesC language, a dialect of C, which has constructs and semantics to implement the event driven approach. Lately nesC is being replaced by a more efficient language and compiler known as galsC. In this report we have described specific language constructs and recent developments of both the languages.

Tiny OS

Wireless sensor networks are gaining popularity and are widely used in habitat monitoring, health service, position tracking and data collection like temperature, light and pressure. Due to advances in VLSI technologies it is now possible to fabricated processor, external memory, sensors, radio module and battery module all on the single chip. This has greatly reduced the size and cost of these devices.

Requirements that Tiny OS has to meet:

- Wireless sensor networks employ thousands of tiny devices called mote and each of these devices carry out extensive communication with each other and also perform computation of data. Due to the physical constraints, memory and supply voltage is limited.
- A mote is responsible to sample data, perform computations, send the results to root node, route incoming packets etc. Plus some applications require these tasks to be real time.
- Just as each mote vendor product is different for each other in the same way each application is different from each other.

In short tiny OS design is based on the above three points i.e limited resources, reactive concurrency and flexibility.

Tiny OS Design

Tiny OS application consists of scheduler and a graph of components. This implies that in true sense tiny OS is not an OS rather an application specific OS that is built using components. By wiring components together a logical graph is formed which represents the applications. This wiring is done via interfaces on each component. Components consist of commands, events and tasks. Commands are request to perform some actions while event are feedback signals that the action is completed. It uses an event driven model rather than the traditional thread model. Events may also come from hardware interrupts. In this design there is no distinction as kernel space and user space.

Tiny OS uses split phase execution wherein commands and events are decoupled. When components issue a command for some operation, they post tasks and immediately return. Such tasks are executed later by the scheduler. Tasks perform the desired computation and are written keeping in mind run to completion model.
Once the execution is done a signal is sent to indicate end of function. Tasks do not perform polling or blocking and represent intra component concurrency. Unused cycles of the CPU are put to sleep.

Tiny OS is written in nesC, a dialect of C which not only provides the event driven approach but also has features that are suitable for this OS. As stated any application is an interconnection of components. A component has two types of interfaces. One that its ‘provides’ and other that it ‘uses’. ‘provides’ means that this component has a function to be executed when a command is issued. ‘uses’ means that it uses a function of another interface and calls this function via signal.

“nesC has two types of components: modules and configurations. Modules are code written in nesC for calling and implementing commands and events “[1]. A component declares all variable as private. Configurations on the other hand, define how to connect the components via the interfaces. Each component has its own namespace with which its calls commands or signals events. When wiring between components you actually connect local name of interface of one component to the other.

As stated earlier that nesC has features suitable for TinyOS and can be supported as follows. Firstly nesC does not allow address passing or pointers. Hence the compiler knows the entire call path during compilation and nothing is dynamic. Because of this it can optimize call path by removing unnecessary overhead as well as inline code for small function calls. Secondly nesC does not support dynamic memory allocation. This prevents memory leaks and helps the compiler to optimize scary memory efficiently.

Wireless sensor networks are real time systems and hence events can arrive anytime and the OS must be capable of handling such request cleanly. Tiny OS incorporates a technique in which event handler is executed immediately. In nesC, tasks are schedule such that they run to completion. Programs post task to the scheduler. Scheduler is a FIFO structure and executes task in any order. Tasks are atomic with each other but can be preempted by interrupts.

Example:
Following represents a snippet taken from the reference paper[1]. In this example component TimerM interfaces HWClock is connected component HWClock interface clock.

/*module timerM has 3 interfaces */

module TimerM {
    provides {
        interface StdControl ;
        interface Timer;
    }
    uses interface Clock;
}

Code for interfaces:
/* interface has 3 command. Hence it must include the service code for all the commands */
interface StdControl {
    command result_t init();
    command result_t start();
    command result_t stop();
}

Interface Timer {
    Command result_t start(char type, unit32_t interval);
5

Command result_t stop();
Event result_t fired();
}

Interface Clock{
Command result_t setRate(char interval, char scale);
Event result_t fire();
}

Configuration:

```
/* configuration define how components are wired. The last line wires the interface together.*/
configuration TimerC provides {
interface StdControl;
interface Timer;
}
implementation{
components TimerM, HWClock;
timerM.clk -> HWClock.Clock
}
```

**Tiny Active Messages**

TinyOS uses active messages to communicate between components. Messages contain names of the handlers to be invoked and data (arguments) for that handler. It is the duty of the handler to extract the message, do the computation and send response back.

**Middleware**

Middleware is software that separates the operating system from underlying hardware. It provides hardware abstraction such that programmers can concentrate on the application than worrying about how to operate the hardware. According to Wikipedia, “middleware is computer software that provides services to software applications beyond those available from the operating systems”[2]. Services like abstraction, reusable code and efficient resource management like power management are provided by such computer programs. This implies that middleware is a piece of code that helps the applications interacts with the hardware and carries out the desired functionality.

In the context of wireless sensor networks, developing of middleware is a challenge. Due to the growing demands of these sensor networks, the traditional middleware do not stand up to the task. Following we have listed some design challenges for middleware [3],

1. **Abstraction support**

Devices or so called motes are manufactured by different vendors each having their own specifications. Hence a code running on vendor A mote does not guarantee that it will function correctly when run on vendor B mote.
2. **Context Awareness**
Distributed systems middleware technique provides transparency by hiding context information but for wireless sensor network context is important. Mobile computing middleware supports context aware technique but these concentrate on individual nodes rather that entire system as a whole. Sensor networks usually work in collaboration of nodes and hence middleware for such network have to different.

3. **Data compatibility**
Data collected form sensors have to be modified to suite the applications demands. Middleware should be capable of performing such transformations on data.

4. **Resource Constraints**
Size of middleware software has to small since sensor networks have stringent memory constraints. Also middleware should be optimized to use minimum power and work satisfactorily with the given computational power.

5. **Dynamic Topology**
Sensor networks are deployed in harsh environments and node failure, communication failure are a common phenomenon. Middleware should be capable of handling such dynamic changes in topology.

6. **Scalability**
Middleware should not have a constraint on the numbers of motes to be deployed in the application. It should be capable to handle vast changes in the numbers of devices.

7. **Security**
For sensitive applications, although OS provides some sort of security, middleware should be able to support and provide another layer of security.

A middleware solution should include the following components: 1. Programming abstractions, systems services, runtime support and QoS mechanism. (Refer below for figure)

**Programming abstractions:**
This component provides high level interface to the developer so that developer do not have to worry about hardware issue and can concentrate on applications requirements. Further programming abstractions can be viewed as abstraction level, programming paradigm and interface type. Abstraction level defines if the programmer views the entire system as a one entity called system level or at node level. “Programming paradigm refers to the model of programming the applications”[3]. This depends on the application in concern. And lastly interface type refers to style of the programming the interface.

**Systems Services**
This aspect refers to the different types of services that the middleware should provide. They include services like code and data management, resource discovery, resource management and integration. All the above are common services usually all WSN applications have to use. The other type of service is domain service. In this type services are specific to an application.

**Run time support**
Run time support means to provide additional functions to the existing embedded operating system and provide functions like task scheduling, IPC, memory management, run time component addition and deletion etc.
**QoS Mechanism:**
This property of a middleware helps to judge the performance of the system. It includes parameters like message delay, jitter, throughput and latency.

---

**TinyGALS (Globally Asynchronous Locally Synchronous)**

Previous developments in embedded software with concurrency oriented systems (event driven) has been done with improvements in low level languages, which could obtain appreciable results with very small memory requirement but when complexity of these systems started growing, it became very difficult to proceed approach. Use of high level sequential languages such as C, C++ which are considered to be one of the best approaches for sequential flow systems also was not possible in this case. Hence to ease the programmers work, allowing them to focus more on application development, this programming model has been developed which can directly generate code by accepting details of application structure.

TinyGALS is a programming model for programming Event driven embedded systems which uses TinyOS as its underlying component. We have studied this model concentrating on programming of event driven Wireless Sensor networks. This model provides globally asynchronous and locally synchronous communication among the event driven sensor network elements. ‘Asynchronous’ (taking larger amount of time) and ‘Synchronous’ (quickly completing computations) in sense of computing systems are differentiated on the basis of time required to perform computation.

Implementing globally asynchronous approach, it provides clear separation between Reactive part and Computation part of the network elements.

Starting from Initial models of TinyGALS to the model which we have studied are developed on the basis some assumptions [5] such as,

1. This programming model would work on single processor systems, which is most of the times true for the cases of WSN nodes.
2. A complete application runs in a one thread and it breaks the flow of execution only when hardware interrupt is there.
3. These interrupts are non-reentrant and the specific interrupt is masked when it is being served.
4. The only way of preemption is hardware interrupts which would be obvious in an event driven system.

Programming model:
The basic unit of TinyGALS programming model is a “Component”[6] which abstracts a hardware element. This model follows a hierarchy, as a Program consists of an Application, which is composed many “Modules” glued together and each of these modules consists of “Components’. Components communicate each other with synchronous method calls and modules communicate with each other asynchronously by message passing. It also implements a supporting model for guarding shared variable data among modules known as TinyGUYS (Guarded Yet Synchronous) Let us have look at each of these hierarchy elements :-

Components:
Component ‘C’ can be given as,

\[ C = (V_C, X_C, I_C) \quad [6] \]

where,
1. Internal variables \((V_C)\) which defines state of a component and change when state changes,
2. External variables \((X_C)\) which are local to a module in which component is integrated and can be accessed by read and write calls and
3. Group of methods \((I_C)\). These methods are further classified as ACCEPTS\(_C\) and USES\(_C\).

ACCEPTS\(_C\) methods are inputs to the component and can be called by other components while USES\(_C\) methods are outputs of component, which may belong to other to other components.

As mentioned above, Component is a software level abstraction of hardware element in a sensor network. So this component is activated in following events. First when, abstracted hardware generates an interrupt (Source component), Second when, there is an event on input port of an module to which component is connected via method call (Triggered component) and Third when, it is activated by another component connected before it (Called components).

To avoid problems of race conditions in components, Source components should not have any inputs associated with any event occurring in network system.

Modules:
Modules integrate one or more components and are considered as basic structural blocks of TinyGALS program. TinyGALS module ‘M’ can be given as,

\[ M = \{ \text{COMPONENTS}_M, \text{INIT}_M, \text{INPORTS}_M, \text{OUTPORTS}_M, \text{PARAMETERS}_M, \text{LINKS}_M \} \quad [6] \]

where,
1. \text{COMPONENTS}_M are the TinyGALS components of which the module is composed of,
2. \text{INIT}_M gives the module’s component methods which are used for initialization purpose (eg. assigning initial values to all internal variables),
3. \text{INPORTS}_M gives list of the input ports and
4. \text{OUTPORTS}_M is list of output ports of a module. These input and output ports are connected to specific components within an module.
5. \text{PARAMETERS}_M are list of global variable mappings specific to module and
6. \text{LINKS}_M specifies the interconnection of components, connection between components and input or output ports via method calls.

A specific module may be activated in following cases. First when, one of the Source components in it activated via hardware interrupt or Second when, one of the Triggered component is activated by an event at input port. The execution of these modules is handled by scheduler of in FIFO (First In First Out) manner. Cyclic executions of components in a module are not permitted, as control flow would enter in an infinite loop and hence Direct Acyclic Graph approach is implemented while developing logics for module.

Application:
TinyGALS Application is a collection of modules which collectively perform a specific task. The elemental structure of TinyGALS application ‘A’ can be given as,
\[ A = (\text{MODULES}_A, \text{GLOBALS}_A, \text{VARMAPS}_A, \text{CONNECTIOSN}_A, \text{START}_A) \] [6]

where, 1. \text{MODULES}_A is a list of all the modules included in an application, 2. \text{GLOBALS}_A is a list of global variables used by application, 3. \text{VARMAPS}_A is a list of global variable mappings with respect to the parameters discussed above in module, 4. \text{CONNECTIOSN}_A specifies all the interconnections between modules included in an application and 5. \text{START}_A is the input port of one of the modules from which execution of application will start after initialization.

For every connection in an application, there is queue associated with it at the Input port, which saves the arguments obtained from the corresponding output port and forwards these arguments to the connected method call activating the respective component. These queues are responsible for the separation of control flow and globally asynchronous nature of this programming model. When a certain event occurs, the task to be performed is pushed into FIFO queue of scheduler and whenever the task reaches the head of the queue, the respective module is activated by event on its input port and corresponding components complete their operations depending on the links in that module and arguments associated with that task are obtained from the queue associated with that connection, as explained above. Cyclic executions of modules in an application are allowed.

**TinyGUYS:**

By implementing the globally asynchronous model, the modules are decoupled from each other [5] and hence can be developed independently. But these modules have shared global data which must be handled with lot of care. TinyGUYS provides mechanism with which global variables used by modules can be read quickly (Synchronously) but writing to these variables takes place atomically in safe conditions (decided by scheduler when to update values).

Let us have a look at a simple example of semi-psuedo code for implementing this programming model from Reference [6]

---

**Component interface for COUNTER.**

\[
\text{COMPONENT COUNTER;}
\text{ACCEPTS}{\text{char init(void); \text{void fire(void);}};}{\text{USES}{\text{char fireOut(short value);}};}
\]

**Module definition for count.**

\[
\text{include components}{\text{COUNTER; \text{CLOCK;}};}{\text{init}{\text{COUNTER.init;}};}{\text{ports in}{\text{count_start;}};}{\text{ports out}{\text{count_out;}};}{\text{CLOCK.fireOut COUNTER.fire \text{count_out}};}{\text{COUNTER.fireOut count_start \text{CLOCK.init}}}
\]

**Application definition.**

\[
\text{include modules}{\text{count; \text{leds;}};}{\text{count_out -> leds_in 50 \text{START@ count_start}}}
\]
nesC (Networked Embedded Systems C)

nesC is a dialect of C and aptly suitable for event driven model. TinyOS being an event driven OS is itself built in nesC language and the OS itself is a scheduler plus a graph of components connected to each other. Components are wired together to build a program. Component has two scopes, 1. Specification and 2. Implementation. Specification of a component behavior is defined by set of interfaces. Interfaces can be provides or uses. A provided interface is the one which provides some sort of service to the caller. Used interface is the one which uses someone else function to complete its task. Components are connected to each other via interfaces. This linking is done at compile time. At compile time the compiler is capable of detecting potential race conditions. The compiler for nesC generates one executable file for the entire program.

Interface Specification:
An interface definition contains a unique identifier name, C – type parameters (optional), list of commands and events. If a type parameter is included in the definition then it is called generic interface definition. Interface definition have global scope.
An interface type is a reference to the interface. If you are using generic interface definition then include the corresponding type argument as well.

This piece of code is taken from [14], which declares two interfaces namely getbuffer and Led. Interface Led is a generic interface.
Sample component has Myled which provides interface of type Led and Mygetbuffer which uses interface of type getbuffer. This component must write service code Myled.blink command and use Mygetbuffer.get_resp event.

Component Specifications
One of the parts of Component declaration is specification. In specification you declare which interface the component uses and which it provides. Now in the above example, Sample is the component which declares Myled as provide interface of type Led. Similarly Mygetbuffer is an object of type getbuffer which is a ‘uses’ interface. You can also declare bare commands and events without them being declare in any interface type. The storage class of these declarations should be command or event.

Component Definition
“A nesC component definition contains name, arguments (optional), specification and implementation[1]” A component can be any one of the following kind: module, configuration, component, generic module and generic configuration. Implementation is module, binary component or configuration.
Examples:
Module C {provides interface Led} - simple module
Generic configuration C(t) {uses interface getbuffer } – generic configuration with a type parameter ‘t’. Binary components have no implementation. Their implementation is in some object other file which has to be linked during compilation.
Configuration implementation defines have components are wired to form the application.

Call and signal
To call commands or signals events nesC uses the ‘call’ and ‘signal’ keyword respectively.
Task
In nesC tasks are lines of code that carry out a service. Tasks are generally postponed for deferred execution. A task is declare as follows: task void do_task() {}. The storage class is task, return type is void and no arguments. When a task is postponed it returns 1 immediately to the call to specific that the post was successful else 0. To post a task use the keyword ‘post’.

Atomic
Statements written under the atomic parenthesis are executed without interruption.

Concurrency
NesC concurrency model revolves around tasks and interrupts. Tasks and interrupt handlers are posted in the task queue. The scheduler can schedule any task anytime. Tasks cannot preempt any other task, but an interrupt handler can preempt any task. nesC code can be divided into synchronous code and asynchronous code. Synchronous code represents code that is reachable from tasks. Asynchronous code represents code that is reachable from interrupt handlers. To avoid race condition, nesC compiler checks to see if there is a competition to access any shared variable between SC and AC code or AC and AC code.

garsC
garsC is a language and compiler designed for use with the TinyGALS programming model, which uses TinyOS as the underlying component model[8]. It extends nesC programming language. As we have focused our study on event driven sensor networks, we have studied garsC as a platform for developing easier solutions to problems related to this field. It was developed to achieve higher level programming comforts and better task level programming, with earlier version of TinyOS, but recent work in this domain shows that, it is still proving to be a very useful development platform with recent updated versions of TinyOS.

The main advantage of using garsC for developing event driven sensor network applications is, [4] it provides basic concurrency constructs and generates executable code, including an application-specific operating system scheduler, from high-level specifications. Though garsC is said to be the extension of nesC, the major advantages provided by it is garsC compiler provides better type checking and code generation which saves a lot of time in debugging the application code [11].

Language constructs:
The language constructs are very much similar to TinyGALS programming model and hence follows same hierarchy, but with additional features. Component ‘C’ (basic element) consists of executed code and list of connection between interface methods which are used by that component. Next abstraction level is Actors or what we referred to as Modules in TinyGALS discussion are considered to be the building blocks of a garsC program. Actors mainly consist of, 1. Interfaces which has list of I/O ports and Parameters (global variables) 2. Components (i.e. list of components) and 3. Connections (describing the flow between components integrated in the actor). Final level of abstraction is Application which has all the actors, interconnections between actors and list of Global variables required.

An important differentiating point in implementation of garsC than TinyGALS programming model is way in which TinyGUSYS(Parameters – global variables) implemented. A component interface or actor port can directly read or write the global data, by calling the corresponding function by passing just one argument it. These parameters have global names which are mapped to local names of parameters in each actor. [11]
Since the earlier versions, a basic unit in development of application using galsC has been a Component, it is evident that application is developed by making logic and code for individual sensor node or mote[2], but current advances are being tried to make this language efficient to develop a program for whole sensor network. This would make it possible integration of number of sensor networks to make a very high end sophisticated application.

Future Advances and Conclusion

According to our study, one of the most important changes in execution model of TinyOS is the introduction of TOS threads library, which is being used to develop multi-threaded programs as hardware advances would allow use of multiprocessors in WSN in nearest future. Unlike previous versions, a complete separation between kernel and application spaces has been introduced, which also separates kernel and user level threads, later being lower priority, run when kernel is idle.

Moreover, as we have discussed, initial versions of TinyOS had First In First Out (FIFO) scheduling algorithms where there were chances of lower priority task being executed before higher priority task, currently developers are trying to implement sophisticated real time scheduling algorithms like Earliest Deadline First(EDF) and Shortest Job First(SJF).

In TinyOS 1.x, components could post same task multiple times in shared task queue, which would overflow the task queue. In this scenario, if an event is signaled, it does not get place in queue and corresponding command which posted it in task queue will wait for infinite time. But in 2.x, each task has dedicated slot in queue and component can post the task in a queue only once. In second attempt to post the same task which is already in queue, then that task is rejected and a bit is set. Now when task is removed from queue, the bit is reset and same task is allowed to be posted in task queue, this prevents overflowing of task queue.

Hence we conclude by saying that TinyOS is an optimal solution for event driven sensor networks. Its component based programming model coupled with event intensive approach makes it favorable for Wireless Sensor Networks. TinyOS have many power management modes and small memory footprint which satisfy the basic needs of a WSN.

Commercial use of TinyOS

Here I am including some links which can give reader a better idea of practical applications of TinyOS and its products which would be marketed in nearest future.

1. http://www.zolertia.com/ ->
   Their signature product Z1, wireless sensor nodes is using TinyOS
2. http://www.advanticsys.com/ ->
   This company manufactures nodes used in Wireless Sensor Networks. The hardware platform used is TELSOB. TinyOS and ContikiOS are the operating systems used in these nodes.
References

[1]. Jason Hill, Robert Szewczyk, Alec Woo, Philip Levis, Sam Madden, Cameron Whitehouse, Joseph Polastre, David Gay, Cory Sharp, Matt Welsh, Eric Brewer and David Culler “Tiny OS : An operating system for sensor networks”


[5]. ‘TinyGALS: A Programming Model for Event-Driven Embedded Systems’ by Elaine Cheong, Judy Liebman, Jie Liu, Feng Zhao

[6]. ‘Design and Implementation of TinyGALS: A Programming Model for Event-Driven Embedded Systems’, a report of research project by Elaine Cheong


[10]. http://ptolemy.eecs.berkeley.edu/other/galsC.htm


[12]. Article – “Operating Systems for Wireless Sensor Networks: A Survey” by Muhammad Omer Farooq and Thomas Kunz, Department of Systems and Computer Engineering, Carleton University Ottawa, Canada

     University of California, San Diego

[14]. “nesC 1.3 language reference manual” by David Gay, Philip Luis, David Culler, Eric Brewer, July 2009