iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 2 **Issue:** XI **Month of publication:** November 2014

DOI:

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

ISSN: 2321-9653
International Journal for Research in Applied Science & Engineering Technology (IJRASET)

Implementation for Wireless Sensor Network of Tiny Operating System

Upender Yadav[1], Nikhil Kalra[2]
Computer Science Department, Dronacharya College of Engineering, India

Abstract— We provide overview of tiny operating systems for Wireless sensor network and present the significant features of each one of micro threaded operating system. It is a technology which has capability to change many of the information communication aspects in the upcoming era. Tiny-OS meets these challenges well and has become the platform of choice for sensor network research area. Tiny-OS system architecture is an active messages communication system. Tiny-OS maintains a two-level scheduling structure, so a small amount of processing associated with hardware events can be performed immediately while long running tasks are interrupted. A critical step towards achieving the vision behind wireless sensor networks is the design of software architecture such as Tiny micro threading operating system, Tiny-os execution model and Tiny-os component model are the important tiny- os features that are elected to organize the accessible WSN.

Keywords— Tiny-OS, Wireless sensor network, Software Architecture, Micro threading.

I. INTRODUCTION

A significant step towards achieving the vision at the back wireless sensor networks is the design of a software structural design that bridges the gap between raw hardware capabilities and a complete system. The demands placed on the software of wireless sensor networks are numerous. It must be efficient in terms of memory, processor, and power so that it meets strict application requirements. It must also be agile enough to allow multiple applications to simultaneously use system resources such as communication, computation and memory. The extreme constraints of these devices make it impractical to use legacy systems. Tiny-OS is an operating design explicitly for network sensors.

Tiny-OS draws strongly from previous architectural work on lightweight thread support and efficient network interfaces. Included in the Tiny-OS system architecture is an active messages communication system. We believe that there is a fundamental fit between the event based nature of network sensor applications and the event based primitives of the active messages communication model. In working with wireless sensor networks, two issues emerge strongly: these devices are concurrency intensive - several different flows of data must be kept moving simultaneously, and the system must provide efficient modularity hardware specific and application specific components must snap together with little processing and storage overhead. We address these two problems in the context of current network sensor technology and the tiny micro threaded OS. Analysis of this solution provides valuable initial directions for architectural innovation.

II. TINY MICRO THREADING OPERATING SYSTEM

Small physical size, modest active power load, and micro standby power load must be provided by the hardware design. However, an operating system framework is needed that will retain these characteristics by managing the hardware capabilities effectively, as well as supporting concurrency-intensive operation in a manner that achieves efficient modularity and robustness. Existing embedded device operating systems do not meet the size, power and efficiency requirements of this regime. These requirements are strikingly similar to that of building efficient network interfaces, which must also maintain a large number of concurrent flows and juggle numerous outstanding events [1]. In network interface cards, these issues have been tackled through physical parallelism [2] and virtual machines [3]. We tackle it by building an extremely efficient multithreading engine. As in TAM [3] and CILK [4], Tiny-OS maintains a two-level scheduling structure, so a small amount of processing associated with hardware events can be performed immediately while long running tasks are interrupted. The execution model is similar to finite state machine models, but considerably more programmable.

Tiny-OS is designed to scale with the current technology trends supporting both smaller, tightly integrated designs, as well as the crossover of software components into hardware. This is in contrast to traditional notions of scalability that are centred on scaling up total power/resources/work for a given computing paradigm. It is essential that software architecture plans for the eventual integration of sensors, processing and communication.

In order to enable the vision of single-chip, a lower cost sensor node, Tiny-OS combines a highly efficient execution model, component model and communication mechanisms.

III. TINY-OS IMPLEMENTATION MODEL

For availability of the extreme levels of operating efficiency that is required in wireless sensor networks, Tiny-OS uses event based execution. The event model allows for high levels of concurrency to be handled in a very small amount of space. In contrast, a thread based approach requires that stack space be pre-allocated for each execution context.

Additionally, the context switch overhead of threaded systems is significantly higher than those of an event-base system. Context switch rates as high as 40,000 switches per second are required for the base-band processing of a 19.2 Kbps communication rate.

This is twice every 50 us, once to service the radio and once to perform all other work. The efficiency of an event-based regime lends itself to these requirements.

A. Event Based Programming

In an event based system, a single execution context is shared between unrelated processing tasks. In Tiny-OS, each system module is designed to operate by continually responding to incoming events. When an event arrives, it brings the required execution context with it. When the event processing is completed, it is returned back to the system. Researchers in the area of high performance computing have also seen that event based programming must be used to achieve high performance in concurrency intensive applications [5, 6]. In addition to efficient CPU allocation, event-based design results in low power operation. A key for limiting power consumption is to identify when there is no useful work to be performed and to enter an ultra-low power state. Event-based systems force applications to implicitly declare when they are finished using the CPU. In Tiny-OS all tasks associated with an event are conduct rapidly after every event is signalled. When an event and all tasks are fully processed, unused CPU cycles are spent in the sleep state as opposed to actively looking for the next interesting event [9]. Eliminating blocking and polling prevents unnecessary CPU activity.

B. Tasks

A limiting factor of an event based program is that long-running calculations can disrupt the execution of other time critical subsystems. If an event is not complete, all other system functions would halt. To allow for long running computation, Tiny-OS provides an execution mechanism called tasks. A task is an execution context that runs to completion in the background without interfering with any other system events.

Tasks can be scheduled at any time but it will not execute until current pending events are completed. Additionally, tasks can be interrupted by low-level system events. Tasks allow long running computation to occur in the background while system event processing continues.

Currently task scheduling is performed using a simple FIFO scheduling queue. While it is possible to efficiently implement priority scheduling for tasks, it is unusual to have multiple outstanding tasks. A FIFO queue has proven adequate for all application scenarios we have attempted to date.

C. Atomicity

In addition to providing a mechanism for long-running computation, the task Tiny-OS primitive also provides an elegant mechanism for creating mutually exclusive sections of code. In interrupt-based programming, data race conditions create bugs that are difficult to detect. In Tiny-OS, code that is executed inside of a task is guaranteed to run to completion without being interrupted by other tasks [12]. This guarantee means that all tasks are atomic with respect to other tasks and eliminate the possibility of data race conditions between tasks.

Low-level system components must deal with the complexities associated with re-entrant, interrupt based code in order to meet their strict real-time requirements. Normally, only simple operations are performed at the interrupt level to integrate data with ongoing computation. Applications can use tasks to guarantee that all data modification occurs atomically when viewed from the context of other tasks.

IV. TINY-OS COMPONENT MODEL

In addition to use the highly efficient event-based execution, Tiny-OS also includes a specially designed component model

targeting highly efficient modularity and easy composition. An efficient component model is essential for embedded systems to increase reliability without sacrificing performance. The component model allows an application developer to be able to easily combine independent components into an application specific configuration.

In Tiny-OS, each module is defined by the set of commands and events that makes up its interface. In turn, a complete system specification is a listing of the components to include plus a specification for the interconnection between components. The Tiny-OS component has four interrelated parts: a set of command handlers, a set of event handlers, an encapsulated private data frame, and a bundle of simple tasks. Tasks, commands, and event handlers execute in the context of the frame and operate on its state. To facilitate modularity, each component also declares the commands it uses and the events it signals.

These declarations are used to facilitate the composition process. As shown in Figure-1, composition creates a graph of components where high level components issue commands to lower level components and lower level components signal events to the higher level components.

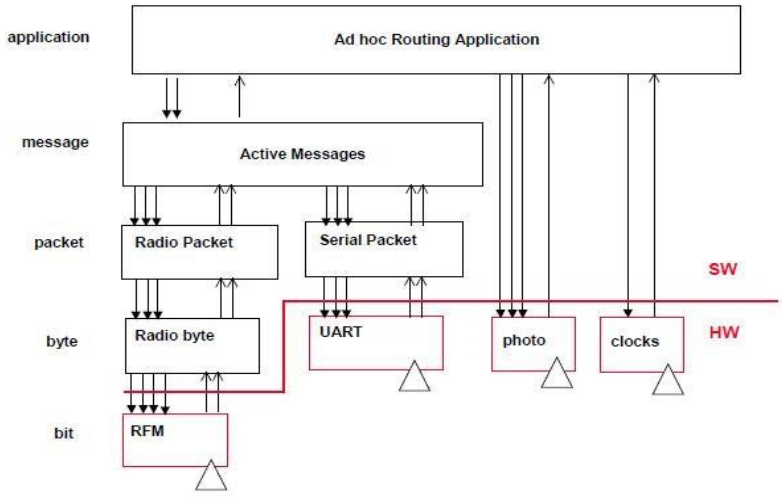


Fig. 1 Component graph for a multi-hop sensing application.

V. CONCLUSION

The objectives of this paper to provide overview of tiny operating systems for Wireless sensor network and to present the significant features of each one of micro threaded operating system. Tiny-OS system architecture is an active messages communication system. Tiny-OS maintains a two-level scheduling structure, so a small amount of processing associated with hardware events can be performed immediately while long running tasks are interrupted. Context switch rates as high as 40,000 switches per second are required for the base-band processing of a 19.2 Kbps communication rate. Event-based systems force applications to implicitly declare when they are finished using the CPU. Tasks can be scheduled at any time but it will not execute until current pending events are completed. All tasks are atomic with respect to other tasks and eliminate the possibility of data race conditions between tasks.

REFERENCES

[1] C. L. Liu And James W. Layland *"Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment"* Project MAC, Massachusetts Institute of Technology Jet Propulsion Laboratory, California Institute of Technology IEEE Vol 20 No.1 1973

[2] GC Buttazzo, G Lipari, M Caccamo *"Elastic Scheduling for Flexible Workload Management"* - IEEE 2002 Vol 54 computer.org

[3] C.Steiger, H.Walder and M.Platzner, *"Operating Systems for Reconfigurable Embedded Platforms:Online Scheduling of Real-Time Tasks"* IEEE Nov. 2004 Vol 53 Issue: 11

[4] Krzysztof M. Sacha *"Measuring the Real-Time Operating System Performance"* 1068-3070/95 1995 IEEE

[5] AbouGhazaleh, N. Mosse, D. Childers, B. Melhem, R. Craven, M. *"Collaborative Operating System and compiler power management for real time applications"* Dept. of Comput. Sci., Pittsburgh Univ., PA, USA IEEE 2003 Vol 56 10.1109/RTTAS.2003.1203045

[6] C. Centioli, F. Iannone, G. Mazza, M. Panella, L. Pangione, V. Vitale, and L. Zaccarian *"Open Source Real-Time Operating Systems for*

Plasma Control at FTU" IEEE VOL. 51, NO. 3, JUNE 2004

[7] The Performance and Energy Consumption of Embedded real time operating systems

[8] Culler, D.E., J. Singh, and A. Gupta, Parallel Computer architecture a hardware/software approach. 1999.

[9] Esser, R. and R. Knecht, Intel Paragon XP/S - architecture and software environment. 1993: Technical Report KFA- ZAM-IB-9305.

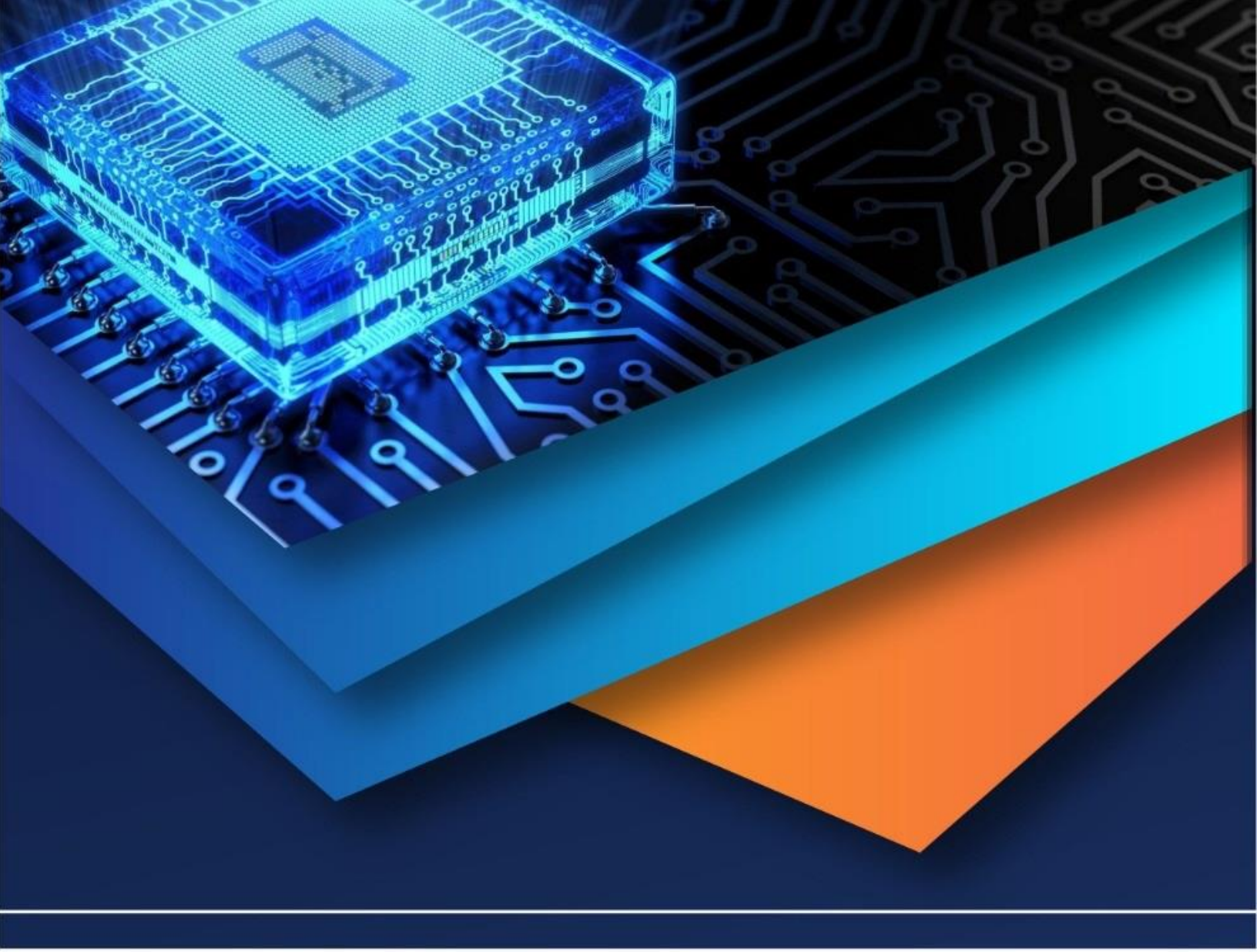
[10] Culler, D.E., et al. Fine-grain parallelism with minimal hardware support: a compiler-controlled threaded abstract machine. 1991.

[11] Blumofe, R., et al., Cilk: An Efficient Multithreaded Runtime System. Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming, 1995.

[12] K. Geihs, Middleware Challenges Aheadl, IEEE Computer, Juni/2001, S. 24-31.

[13] Rodoplu, V. Meng, T.H. Minimum energy mobile wireless networks. IEEE J. Sel. Area. Commun. **1999**, 17, 1333-1344.

[14] Li, L. Halpern, J.Y. Minimum-energy mobile wireless networks revisited. IEEE Int. Conf. Commun. **2001**, 1, 278-283.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY