Nano-kernel: A Dynamically Reconfigurable Kernel for WSN

Susmit Bagchi
ATD/CS Research Laboratory
Samsung India Software Operations
Bagmane Technology Park
Bangalore, India

susmitbagchi@yahoo.co.uk

ABSTRACT

The Wireless Sensor Networks (WSN) have received considerable research attention in recent time. The sensor devices of a WSN are severely resource constrained having a very limited operational lifetime. Such sensor devices have to adapt to the changing environment at deployment site and need dynamic reconfiguration. The operating systems supporting the sensor devices should be capable of realization of dynamic reconfiguration at kernel level as well as at application layer. This paper proposes a design framework of nano-kernel, a lightweight operating system for sensors. The proposed nano-kernel architecture incorporates dynamic reconfiguration capability by decoupling the kernel data objects from the policies implemented in the kernel subsystems. Based on the modular design approach, an implementation direction is outlined.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design.

General Terms

Design, Theory.

Keywords

Wireless Sensor Networks, Mote, Kernel, Modules, Software Components, Object Oriented Design.

1. INTRODUCTION

The Wireless Sensor Networks (WSN) have gained a substantial research attention in recent time [1][2]. The wireless sensor nodes, sometime referred to as mote or mote-class devices, are the resource constrained computing devices equipped with a set of sensors and wireless networking capability, often deployed in-situ [1]. The sensor nodes sense the environmental phenomena upon deployment based on integrated hardware platforms composed of an array of sensors and the wireless networking system having the backend data services [1]. In general, the majority of the tasks running on the sensor nodes are interrupt-driven IO intensive in nature and a few are the CPU intensive tasks needing periodic wake-up schedule [12]. The mote-class device or sensor hardware is typically controlled by the embedded operating systems. Based on the multitasking lightweight operating systems, the WSN nodes support a set of complex tasks such as, signal processing and target tracking [1], time synchronization [7][8] and data compression along with encryption [9]. Designing the embedded and lightweight operating system for WSN is challenging due to severe resource constraints of the micro sensor nodes along with restricted operational lifetime based on battery capacity [1][12]. The desirable features of such embedded operating system for WSN are reconfiguration capability [10][11][13] and supporting the dynamic reprogramming of the mote [1]. Reconfiguration capability of the sensor system indicates that the system software as well as the applications can change in runtime [1][10]. The design and development complexity of the kernel having reconfiguration capability is balanced by the need of adaptability of the sensor nodes and the WSN system in total to the changing environment such as, temperature, pressure and humidity etc. and changing system resources such as, radio bandwidth and battery power [10][11]. Researchers have proposed to design the reconfigurable embedded operating system for WSN [11][23] as well as the general purpose reflective operating system [14]. The examples of WSN operating systems capable of dynamic loading the application modules include SOS [21] and Contiki [25]. In this paper, an architectural framework of a dynamically reconfigurable operating system kernel for WSN is described. The design philosophies followed are the decoupling of the kernel data objects from the policies or algorithms implementing the kernel subsystems and high modularization of the kernel components by splitting the kernel address space into lightweight nano-kernel core and a set of kernel devices. The various kernel devices are "hooked on" into the nano-kernel core as the dynamically reconfigurable components having detachable interfaces to the nano-kernel. These kernel devices implement the policies or the algorithms and consume the kernel data objects external to them through import/export interfaces. The properties of the proposed design architecture are as followings.

● *Decoupling the kernel data objects from the implementation of algorithms or policies of kernel subsystems.*

● *Persistency of the kernel data objects retaining the previous states of computation.*

● *Dynamic change in the policy or algorithm of a subsystem, which incorporates the dynamic reconfiguration capability of the kernel.*

MOBILWARE 2008, February 13-15, Innsbruck, Austria
Copyright © 2008 ICST 978-1-59593-984-5
DOI 10.4108/ICST.MOBILWARE2008.2500

● *The newly loaded kernel devices or kernel subsystems can use the previous states of the subsystems saved into the persistent kernel data objects, following the checkpointing philosophy.*

The rest of the paper is organized as follows. Section 2 describes the related work. The architectural framework of nano-kernel is illustrated in section 3. Section 4 describes the comparative analysis of the proposed nano-kernel architecture with respect to the other systems. Section 5 and section 6 describe the implementation direction and conclusions, respectively.

2. RELATED WORK

Reconfiguration capability of a kernel is an important feature of the operating system intended to embedded systems and the WSN [10][11]. Specially, in case of WSN, where the sensor devices are distributed in-situ, the reconfiguration and adaptation at application layer as well as at the kernel level are required in order to accommodate the changes in the environment. One way to reconfigure the kernel of the operating system for WSN is to conduct remote reprogramming [1]. It is reported that, the added complexity in the kernel design due to multithreading along with preemptive scheduling based on time quanta can be accommodated in MICA2 mote [1]. An example of multithreaded operating system for WSN is MANTIS [1]. The MANTIS kernel manages the available RAM as the heap, however, the applications running on MANTIS kernel are not encouraged to allocate dynamic memory from the heap in order to reduce the computation overhead [1]. An energy-aware and resource-centric real-time operating system for WSN is Nano-RK [22], which supports priority-based preemptive multitasking. The multithreading optimization techniques for WSN operating systems are illustrated in [24]. In an event-driven kernel design approach, the TinyOS is designed for sensor networks [3]. TinyOS handles the IO requests as a set of events, where an event can interrupt a running task [1][3]. TinyOS uses a modularized static programming language, nesC, which analyzes the code and handles the concurrency issues within the language and not in the user space [1][3]. TinyOS eliminates the context switching and a task runs towards its completion after it is started [3]. Apart from TinyOS, the other standard sensor network systems include MetaCricket [4], Location-aware cricket [5] and BTNodes [6]. In another approach, the THINK component based software architecture is proposed allowing applications to reconfigure at a low cost [11]. Based on the software component model, the THINK architecture encapsulates internal states and offers an abstraction of the states to the environment [11]. The components have a set of interfaces and a binding controller, which binds the components dynamically. THINK core architecture is implemented in C language, not in Java, in order to reduce performance degradation [11]. The main limitation of THINK component model is the increase in main memory footprint. The concept of protected virtual memory system is incorporated in the t-kernel design through the load-time instrumentation of the application code [12]. In t-kernel, the "branch" and "jump" instructions of the application code are "naturalized" to produce "natins" [12]. This imposes the memory access boundary protection on the applications under the control of t-kernel. However, such load-time instrumentation of the application code enhances the code dilation and heavy computation overhead. Hence, the applications tend to slow down considerably [12]. A general purpose reconfigurable and extensible operating system kernel, named Kea, is designed based on the concept of micro-kernel architecture [13][16]. The Kea kernel architecture is comprised of "domains" and "portals" [13][16]. The inter domain calls are controlled through IDC [13], which requires the run-time application stack alteration. The context switching time of Kea kernel is considerably high involving 70% of the time needed for IDC [13][16]. Based on the object oriented design philosophy, a framework for constructing an operating system for mobile computing environment is proposed [14]. The Apertos framework is designed based on the object and meta-object separation as well as maintaining meta-hierarchy among the objects. The limitation of Apertos kernel is the severe performance degradation and higher exception handling latency, which is in the order of 76 microseconds [14]. The other extensible operating systems are exokernel [15], DEIMOS [19] and SPIN [20]. Other than the extensible and reconfigurable designs of operating system kernels, researchers have proposed to decouple the data objects and algorithm implementations in order to introduce flexibility in software architecture [17][18]. The dynamic reconfiguration of the software system architecture gets easily realizable by decoupling the core data objects from the algorithm implementation.

3. NANO-KERNEL ARCHITECTURE

The set of IO intensive tasks or applications runs on the sensor devices in WSN has specific characteristics such as, interrupt-driven periodic or sporadic activation having a short execution lifetime. However, there can be a few CPU bound tasks such as, data encryption algorithm, which will be having relatively longer execution lifetime. As the available RAM at a sensor node is very limited, the kernel of the operating system should be having a small footprint. In addition, the kernel design architecture will be dynamically reconfigurable allowing remote programming. In view of these operational and environmental characteristics of a mote-class device or sensor, the nano-kernel architecture is conceptualized. The proposed kernel architecture is designed based on decoupling the kernel data objects from the logical computation part or the algorithms implementing the kernel subsystems or kernel devices in order to introduce structural flexibility. Inspired by the design philosophy of dynamic modules of Linux kernel, the kernel devices of the proposed architecture are highly modularized by minimizing direct dependency between two kernel devices. The schematic representation of the proposed reconfigurable kernel architecture is shown in Figure 1. The entire kernel address space is logically divided into two parts, nano-kernel core and the other kernel devices implementing policies. This means that, the nano-kernel core is a lightweight single thread of execution and it treats the other components of the kernel viz. scheduling policy, file system, device drivers and memory/resource managers as the separately executing devices within the kernel protection boundary. The kernel devices are decoupled from the persistent kernel data objects to enable the kernel subsystems to be dynamically reconfigurable. The interactions between the nano-kernel core and kernel devices are through two interfaces involving kernel data objects and the nano-kernel/device capability interfaces as shown in Figure 1. This design architecture allows easy replacement of a kernel device or subsystem dynamically without affecting the other kernel components. The nano-kernel does not directly control the peripheral devices of a sensor node. The very low-level device handling is done by hardware specific device drivers.

3.1 Nano-Kernel Core

The nano-kernel core is composed of the single thread of execution controlling the data object access, maintaining interrupt vector table and synchronizing the operations of other kernel devices. The internal architecture of nano-kernel core is illustrated in Figure 2. The applications can invoke a set of system calls (syscalls) to interact with the kernel devices through the nano-kernel core. The functions of the nano-kernel core can be grouped into two categories viz. branching to interrupt handlers based on the sensor interrupts and routing syscalls as well as kernel device calls crossing the device-domains. When an application invokes a system call to interact to the kernel device, the nano-kernel core routs the call to the appropriate kernel device and returns the result to the application. This enables the nearly straight forward application-device interaction with minimum abstraction reducing response time and enhancing portability of applications. In other case, when a kernel device (KD1) calls a service from another kernel device (KD2), then the nano-kernel routs the call from KD1 through the capability interface of KD2. This design rationale helps in creating the highly modularized kernel architecture along with data and policy decoupling. The capability interfaces between the nano-kernel core and kernel devices have two parts denoting *nano-kernel entry points for devices* and the *entry points of devices for the nano-kernel core*. The capability interfaces are constructed through dynamic binding following the module registration design mechanism. The various kernel data objects are created and maintained by the nano-kernel core. The objects reside in the object store maintained of the nano-kernel core and are persistent as long as the system is in the power-on state. The data objects can be created statically at compile time or dynamically by the nano-kernel core and are tagged according to the domain of existence of the data objects. The domain of the persistent kernel data objects are classified into two subclass types namely, hardware data object (HWDO) and software data object (SWDO) as shown in Figure 2. The HWDOs are the objects attached to any specific sensor hardware of the mote-class device. The examples of HWDOs are radio_object associated to wireless radio subsystem and camera_object associated to the camera sensing subsystem of the sensor device. Accordingly, the examples of SWDOs are scheduling_object, memory_object and file_object. The objects of the subclass type HWDOs are accessible to the device driver modules. The objects of the subclass type SWDOs are accessible to various kernel subsystems depending on the object association with the particular subsystem. For example, the scheduler subsystem of the nano-kernel can access the scheduling_object of the object store. The scheduling_object contains all the data variables, scheduling parameters and the list of process control blocks (PCB) related to the scheduling activity. Likewise, the memory_object contains the set of data items related to the memory management system. In addition to the decoupling of the data objects from the policy implementations, the access to the individual data objects within the persistent object store is also domain specific. The kernel object access control mechanism checks and controls any access to kernel data objects from the persistent object store by the kernel devices as shown in Figure 2. This implies that the scheduler subsystem or scheduling device cannot directly access the memory_object. The nano-kernel core checks and controls the import/export of the data objects to/from object store at runtime.

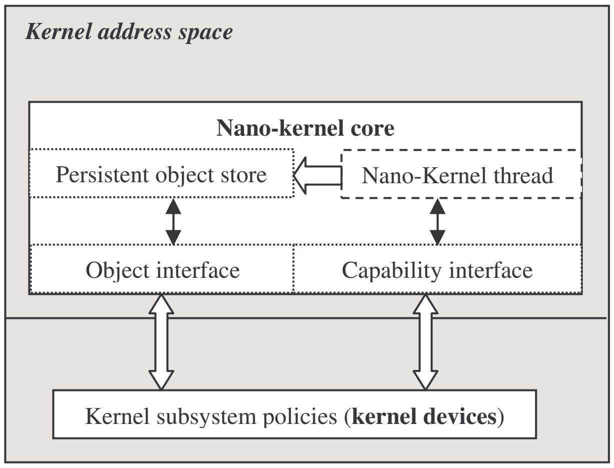


Figure 1. Nano-kernel Architecture Overview.

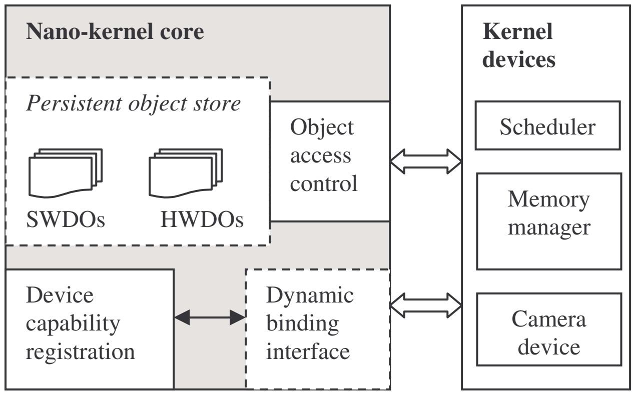


Figure 2. Internals of Nano-kernel Core.

3.2 Kernel Devices

The kernel devices are synonymous to the kernel subsystems implementing the computational and functional policies or algorithms. The nano-kernel core can replace an existing kernel device with another device dynamically. Hence, for example, to the nano-kernel core, the scheduler is a scheduling device, which is a dynamically loadable module. The internal components of a kernel device are shown in Figure 3. The “Local data” (LD) member is a set of local data variables/structures used by a kernel device. The interfaces between the kernel devices and the nano-kernel core are categorized into two types namely, data object interface (DOI) and capability interfaces (CI). The data object interface is used by the kernel devices and the nano-kernel core to implement export/import of the persistent kernel data objects. Likewise, the capability interface between the kernel devices and nano-kernel core is used to invoke services across the domains. The kernel devices import the domain related data objects from the object store, upon loading the module within the nano-kernel core, denoted by “Imported data” (ID) member in Figure 3. The “Computation logic” (CL) of Figure 3 denotes the algorithm or policy implemented by the kernel device, which consumes the imported data objects. The “IO ports” (IOP) member of the kernel device is optional and indicates the hardware port addresses controlled or used by the corresponding device. It is possible that some of the kernel devices do not have IOP, for example the scheduling subsystem. While loading a kernel device, the nano-kernel exposes the data object interface and a set of kernel core capabilities to the device denoting the kernel core entry points (Kernel entry points). In response, the kernel device registers the device entry points within the nano-kernel core denoting the capabilities of the device (Device entry points). Before unloading a kernel device module, the nano-kernel core sends a STOP message through the capability interface of the device. In response the device freezes its current state into the imported data object and exports the updated data object to the nano-kernel core through the DOI. On receiving the exported object within the persistent object store, the nano-kernel core unloads the kernel device.

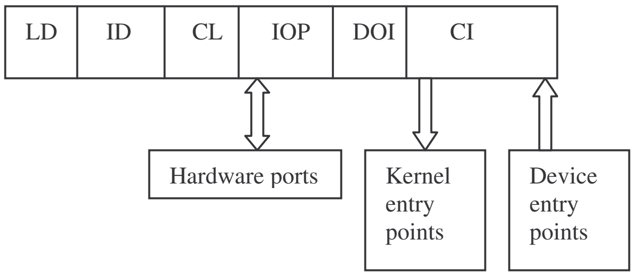
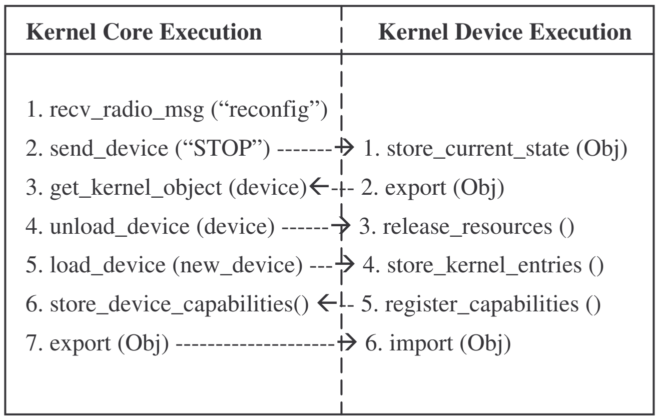


Figure 3. Components of Kernel Devices.



Kernel Core Execution	Kernel Device Execution
1. recv_radio_msg (“reconfig”)	
2. send_device (“STOP”) --------→	1. store_current_state (Obj)
3. get_kernel_object (device)←--	2. export (Obj)
4. unload_device (device) -------→	3. release_resources ()
5. load_device (new_device) ---→	4. store_kernel_entries ()
6. store_device_capabilities() ←--	5. register_capabilities ()
7. export (Obj) --------------------→	6. import (Obj)

Figure 4. Dynamic Reconfiguration Sequences.

3.3 Dynamic Kernel Reconfiguration

The incorporation of dynamic reconfiguration capability of the nano-kernel for WSNs is supported by two design philosophies. These are *(1) decoupling the data objects from the policies or algorithms* and *(2) the highly modularized nano-kernel architecture consisting call routing reducing the inter-dependencies of the kernel subsystems*. In addition, the fault tolerance capability of the nano-kernel is enhanced because, due to reduced inter-dependencies of kernel subsystems, failure of one kernel device would not stop the whole system from functioning. Figure 4 illustrates the reconfiguration sequences in the kernel. Upon receiving the radio message (reconfig) to reconfigure a kernel device (policy), the nano-kernel core sends a “STOP” message to the corresponding device through device entry point. In response, the kernel device gathers the current state of execution into the imported data object and exports the data object to the nano-kernel core. Next, the nano-kernel core unloads the kernel device and loads a new kernel device as a replacement. Upon loading a new device, the dynamic binding of capability interfaces is accomplished. In the next step, the kernel device imports the corresponding kernel data object of matching subclass type from the nano-kernel core. The nano-kernel core does not interpret any data present in the kernel data objects; rather, it acts as the facilitator of the resources.

4. COMPARATIVE ANALYSIS

This section provides the comparative analysis of the proposed nano-kernel architecture with respect to the other contemporary systems. The THINK component model [11] follows the exokernel-like architecture [15]. The nano-kernel does not follow the exokernel-like architecture and hence, the memory footprint is not dilated in case of nano-kernel. Unlike the Kea architecture [13][16], the proposed reconfigurable nano-kernel does not introduce inter domain calls crossing memory protection boundary requiring MMU hardware support, which is absent in the CPU of the majority of sensor nodes, as well as computation overhead due to the runtime instrumentation of the stacks. The DEIMOS [19] design model discourages the implementation of complex policies within any kernel subsystem. Unlike DEIMOS, the nano-kernel does not restrict any implementation of the complex policies in dynamically reconfigurable manner as the kernel subsystems. Unlike the TinyOS [1][3], the proposed nano-kernel does not rely on any special static programming language except the general purpose system programming languages such as C and the assembly language. The SOS [21] and Contiki [25] advocate dynamic reloading of application modules. However, the proposed nano-kernel architecture uses the mechanism of dynamic reloading of kernel modules. In contrast to RETOS [23], the nano-kernel architecture does not need dynamic relocation of data section of a module as the data blocks used by the module are resident in kernel as persistent objects.

5. IMPLEMENTATION DIRECTION

This section provides the implementation roadmap of the reconfigurable kernel architecture. It is envisioned that following the design mechanism of dynamically loadable module of Linux kernel, the proposed nano-kernel architecture can be implemented. The nano-kernel core is a single thread of execution started by bootloader of the BSP (Board Support Package) of the sensor devices. The nano-kernel core builds the interrupt vector table, loads the interrupt handler modules and installs the handler entry points. Depending on the configuration of a sensor device, the persistent object store is created and filled with the data objects of different subclass types. Next, the kernel devices are loaded by the nano-kernel core by constructing the data object interfaces and registering the capability interfaces of the loaded kernel devices. At this point, the loaded kernel devices may use the data object interface to import the kernel data objects from the persistent object store of the nano-kernel core. The data object interface is consisting of the functions having prototypes such as, int import_obj (int subclass, string device_name, object_pointer *input …) and int export_obj (int subclass, string device_name, object_pointer *output …), where subclass variable identifies the subclass type of the object, device_name indicates the name of the kernel device requesting the object and the object_pointer is the reference to the object in the persistent data object store. This means that the copying the object is not needed. The return values of the functions indicate whether the import/export of the data objects are successful or failed. The nano-kernel core, on receiving the calls from data object interface, must check the device_name and subclass of the object to determine the validity of the import/export requests. The capability interfaces between nano-kernel core and kernel devices are consisting of a set of functions specific to the kernel device domain. For example, the capability interface of the scheduling device may be consisting of the following functions, dev_start (…,args,…), dev_stop(….,args,….), dev_get_next_schedule(…,args,…), dev_get_highest_priority(…,args,…) etc.

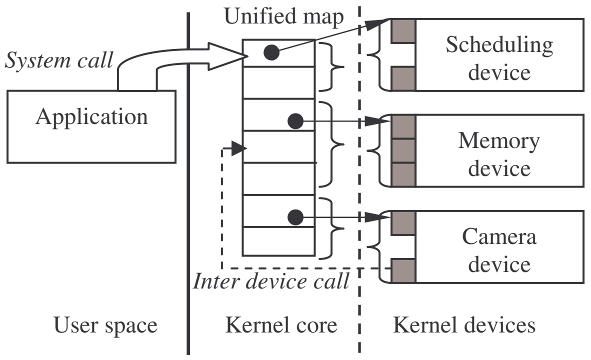


Figure 5. The Unified Mapping and Call Routing Abstraction.

In order to establish call routing for the syscalls invoked by applications as well as the inter-device calls, the nano-kernel creates a unified mapping of the incoming calls to the appropriate functions of the device interfaces. The unified abstraction of the system calls interface maps the system calls and device entry points and may be represented as followings, sys_get_priority (task_id), sys_get_memory (size) etc. The system calls are grouped internally by the nano-kernel core according to the destination kernel devices in order to fast determination of the destination interface function address. For example, sys_get_priority (task_id) system call will be grouped to the mapping of the scheduling device capability interface addresses and sys_get_memory (size) system call will be grouped to the mapping of the memory device capability interface addresses. The single unified mapping architecture helps in easy and fast implementation and reduces the overheads related to multilayered-abstractions such as, code dilation and increased response time. The unified mapping and call routing abstraction is illustrated in Figure 5. The timer interrupt is handled by the nano-kernel core. While handling the timer interrupt, the nano-kernel core checks the scheduling requirement of the system and calls the scheduling device to receive the reference to the PCB of next schedulable task. Maintaining the design philosophy, the context switcher module is isolated from the scheduling device because the context switcher is hardware specific in nature and can be viewed as the CPU-driver. If the scheduling device replies that no task is ready to execute, the nano-kernel puts the system into sleep mode or power-down mode to save battery power. The interrupt handling is done by directly branching to the addresses of the interrupt handler functions and the system call handling is done by routing the calls to the appropriate capability interface functions through unified mapping mechanism. The pseudo code representation of the nano-kernel implementation framework is illustrated in Figure 6.

```
Proc kernel_device (kernel_core_entry *kptr[]) {
obj_pointer *obj; int subclass; string device_name;
dev_entry *ptr[] = {&dev_start(args), &dev_stop(string),..};
internally_store_kernel_core_entry_points (kptr);
kptr →register_dev_entry_points (device_name, ptr);
int x = kptr → import_obj (subclass, device_name, obj);
if (x) initialize_object (obj);
else prepare_to_unload_module ( );
}
Proc dev_start (args){  ………..}
Proc dev_stop (“STOP”) {  stop_device ( );
 save_current_state (obj);  release_resources ( );
 kptr →export_obj (subclass, device_name, obj);
}
```

```
dev_entry_ptrs *dev[];
Proc nano_kernel_core ( ) {
kernel_core_entry  *kptr[]  =  {&import_obj(obj_pointer*),
&export_obj(int,string,obj_pointer*),
&register_dev_entry_points (string, dev_entry *)};
install_interrupt_handlers_once (&dev_interrupt_handler);
install_timer_interrupt_vector_once (&nano_kernel_core);
load_kernel_devices_once ( );
for (all_kernel_devices) {kernel_device (kptr[]); }
while (TRUE) { int x = check_scheduling ( );
    if (x) { PCB_ptr p = dev_get_next_schedule (args);
              context_switcher (p); }
    else sleep (time_quanta); }
}
Proc dev_interrupt_handler ( ) {
switch (interrupt_number) { case radio_intr :
       string msg_recv = check_radio_message_buffer ( );
       if (msg_recv.RECONFIGURE) {
       dev[msg_recv.device_name].dev_stop(“STOP”);
       check_export_obj_completion();
       unload (msg_recv.device_name); load_new_device ();
       kernel_device (kptr[]);} break;
       case camera_intr : camera_intr_hlr( ); break;
       case  TRAP : int index =
        unified_mapping (syscall || inter_device_call);
        route_call_to_device (index); break;
          …………               }
  interrupt_return;
}
Proc register_dev_entry_points (string s, dev_entry *p[])
{  dev[device_name] = s; dev[s].ptrs[] = p[]; }
```

Figure 6. The Nano-kernel Implementation Framework.

6. CONCLUSIONS

The operating systems intended to WSN should be lightweight and fast responding to an external sensor input. This paper proposes a dynamically reconfigurable and lightweight operating system kernel architecture called nano-kernel. The proposed nano-kernel design architecture is consisting of a nano-kernel core and dynamically reconfigurable kernel subsystems. The design philosophies are to decouple the kernel data objects from the implementations of algorithms or policies and to create the highly modularized kernel architecture. All the subsystems of nano-kernel are designed as kernel devices implementing device specific policies or algorithms. The nano-kernel core is a single thread of execution, which dynamically binds the kernel devices through the capability interfaces. The system calls and inter-device calls are routed through a unified mapping abstraction. The kernel devices are two types: a hardware specific module, such as camera driver or a software module, such as scheduler subsystem. Kernel data objects are persistent, imported/exported by kernel devices and retain the last states of computation allowing incorporation of checkpointing feature of the nano-kernel core.

7. REFERENCES

[1] Bhatti S. et al., *MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro-sensor Platforms*, ACM Kluwer Mobile Networks & Applications (MONET) Journal, August, 2005.

[2] Akyildiz I. F. et al., *A Survey on Sensor Networks*, IEEE Communications Magazine, August, 2002.

[3] Hill J. et al., *System Architecture Directions for Networked Sensors*, In Proc. of 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), November, 2000.

[4] Martin F., Mikhak B., Silverman B., *MetaCricket: A Designer's Kit for Making Computational Devices*, IBM System Journal, Vol. 39, No. 3 & 4, 2000.

[5] Priyantha N. B. et al., *The Cricket Location-Support System*, In Proc. of 6th Annual ACM International Conference on Mobile Computing and Networking, August, 2000.

[6] Leopold M., Dydensborg M. B., Bonnet P., *Bluetooth and Sensor Networks: A Reality Check*, 1st ACM Conference on Sensor Systems, LA, November, 2003.

[7] Elson J., Girod L., Estrin D., *Fine-Grained Network Time Synchronization using Reference Broadcasts*, In the Proc. of OSDI-2002, Boston, December, 2002.

[8] Dai H., Han R., *TSync: A Lightweight Bidirectional Time Synchronization Service for Wireless Sensor Networks,* ACM SIGMOBILE Mobile Computing and Communications Review, Vol. 8, No. 1, January, 2004.

[9] Zhao J., Govindan R., Estrin D., *Computing Aggregates for Monitoring Wireless Sensor Networks,* In the Proc. of 1st IEEE International Workshop on Sensor Network Protocols and Applications, Anchorage, May, 2003.

[10] Raatikainen K. E. E., *Operating System Issues in Future End-User Systems*, PIMRC, 2005.

[11] Senart A., Charra O., Stefani J. B., *Developing Dynamically Reconfigurable Operating System Kernels with the THINK Component Architecture*, In Proc. of the Workshop on Engineering Context-aware Object Oriented Systems and Environments, OOPSLA, 2002.

[12] Gu L., Stankovic J. A., *t-kernel: Providing Reliable OS Support to Wireless Sensor Networks*, In the Proc. of 4th International Conference on Embedded Networked Sensor Systems, Colorado, 2006.

[13] Veitch A. C., *A Dynamically Reconfigurable and Extensible Operating System*, PhD Thesis, Deptt. Of Computer Science, The University of British Columbia, July, 1998.

[14] Yokote Y., *The Apertos Reflective Operating System: The Concept and its Implementation,* In the Proc. of Conference on Object Oriented Programming Systems, Languages and Applications, Vancouver, 1992.

[15] Engler D. et al., *The Operating System Kernel as a Secure Programmable Machine*, In the Proc. of ACM SIGOPS European Workshop, 1994.

[16] Veitch A. C., Hutchinson N. C., *Kea- A Dynamically Extensible and Configurable Operating System Kernel,* In the Proc. of 3rd International Conference on Configurable Distributed Systems, IEEE CS, Washington DC. 1996.

[17] Nguyen D., Wong S. B., *Patterns for Decoupling Data Structures and Algorithms*, In the Proc. of 30th SIGCSE Technical Symposium on Computer Science Education, ACM Press, New Orleans, 1999.

[18] Cattaneo M. et al., *GAUDI – The Software Architecture and Framework for Building LHCb Data Processing Applications,* In the Proc. of International Conf. on Computing in High Energy and Nuclear Physics, 2000.

[19] Clarke M., Coulson G., *An Architecture for Dynamically Extensible Operating Systems*, In the Proc. of 4th International Conference on Configurable Distributed Systems, Annapolis, May, 1998.

[20] Bershad B. N. et al., *Extensibility, Safety and Performance in the SPIN Operating System*, In the Proc. of 15th ACM Symposium on Operating Systems Principles, 1995.

[21] Han C.C. et al., *A Dynamic Operating System for Sensor Nodes*, In the Proc. of the 3rd International Conference on Mobile Systems, Applications and Services, USA, 2005.

[22] Eswaran A. et al., *Nano-RK: an Energy-aware Resource-centric RTOS for Sensor Networks*, The 26th IEEE International Real-Time Systems Symposium (RTSS), IEEE CS, USA, 2005.

[23] Cha H. et al., *RETOS: Resilient, Expandable and Threaded Operating System for Wireless Sensor Networks*, International Conference on Information Processing in Sensor Networks (IPSN), ACM Press, USA, 2007.

[24] Kim H., Cha H., *Multithreading Optimization Techniques for Sensor Network Operating Systems*, The 4th European Conference on Wireless Sensor Networks (EWSN), Springer LNCS 4373, Netherlands, 2007.

[25] Dunkels A. et al., *Contiki – a Lightweight and Flexible Operating System for Tiny Networked Sensors*, In the Proc. of 1st IEEE Workshop on Embedded Networked Sensors (Emnets-I), USA, 2004.