Chapter 6: Operating System in Sensors

Although operating system (OS) is a typical computer science (CS) topic, WSN engineers who design sensor hardware should also understand WSN OS characteristics since a successful WSN system needs the tight integration of hardware and software. For instance, if a WSN OS has a set of interrupt commands, how do we design those interrupt wires between a microcontroller and analog sensors? If an OS has wake-up command, how do we design a wake-up circuit to trigger a radio transceiver if there is data to send? This chapter will introduce some most popular WSN OS such as TinyOS.

6.1 TinyOS [Levis06]

Due to serious resource constraints in sensors, TinyOS [Levis06] is designed to be a tiny (fewer than 400 bytes), flexible operating system with a set of reusable components. Those components could be programmed and assembled into application-specific systems. TinyOS is an event-driven OS. That is, it defines a set of functions to be triggered by asynchronous sensor network events such as fire event. TinyOS is implemented in the NesC language [TinyOS07], which has similar syntax as regular C language.

Since TinyOS is still an operating system, it needs to have common functionalities of OS. For instance, it needs to manage files, allocate memory for applications, recycle unused CPU resources, and others. However, different from other OS, TinyOS should fit in WSN's tiny memory and slow CPU features. It also needs to minimize energy consumption in order to elongate sensors' battery lifetime.

6.1.1 Overview

Any TinyOS program can be represented as a graph of software components. Each component is an independent computational entity. There are interfaces among different components to ensure that they could refer to each other.

Components have three computational abstractions: *commands*, *events*, and *tasks*. *Commands* and *events* are mechanisms for inter-component (i.e. between components) communication, while *tasks* are used to express intra-component (inside one component) concurrency.

A *command* is sent out by one component to request another component to execute operations (i.e. services). For instance, a software entity may request the sensor to report current readings.

An *event* is a special software entity that is generated from 3 sources: (1) when a *command* is executed, an *event* message is generated to signal the completion of that service. (2) When the sensor hardware has some special event (such as the wake-up of a radio transceiver), a hardware interrupt may be generated to signal such a new *event*; (3) *Events* may also be signaled asynchronously due to network message arrival in a sensor or base-station.

From a traditional OS viewpoint, *commands* are analogous to “downcalls” and *events* are like “upcalls”. Commands and events cannot block each other. They may be done in different time phases. For instance, TinyOS uses a phase to issue the *request for service* (i.e., sending out the *command*), and uses another phase to send out the completion signal (i.e., generating the corresponding *event*). Those two phases are decoupled. The *command* returns immediately; however, the *event* signals could be completed at a later time.

Why do we use *tasks*? In many cases we cannot finish all operations in a command /

event handler immediately, especially if those operations need to use multiple sensor hardware resources (such as radio transceiver, analog sensor, flash memory, etc.). Thus, commands and event handlers may post a *task*, a function to be executed by the TinyOS scheduler at a later time. By executing the *tasks* at a later time, we make commands and events "look" very responsive, i.e., they return results immediately. However, internally we defer any extensive computation to the *tasks*.

Although we could use *tasks* to perform significant computation, a *task* cannot run indefinitely, that is, run-to-completion is its basic execution model. Therefore *tasks* have lighter weight than threads. *Task*s represent internal concurrency within a component and may only access state within that component (that is, a *task* cannot access two components during its execution). The standard TinyOS *task* scheduler uses a non-preemptive, FIFO scheduling policy.

On *components:* In TinyOS all hardware resources are represented as *components*. For example, after a *component* receives the `getData()` *command,* later on it will signal a `dataReady()` *event* as long as a hardware interrupt fires.

TinyOS has defined many *components* for WSN programmers. An application developer writes components to compose an application. Then those components are wired to TinyOS components in order to provide implementations of the required services. In the following we further look into *component* model.

6.1.2 Component Model

Components encapsulate a specific set of services that are specified by *interfaces*. Not only does each WSN program consist of a series of *components*, TinyOS itself simply consists of a set of reusable system *components* along with a *task* scheduler.

A wiring specification can be used to connect an application to a series of components. And the wiring specification defines the complete set of components that the application uses. The concrete component implementations are independent of the wiring specification.

A TinyOS compiler can eliminate some unnecessary components after an analysis and *inlining* of the entire program. The procedure of *Inlining* can operate across different component boundaries in order to improve both program size and efficiency.

On the concept of "interfaces": As shown in Figure 6.1, any component could have two types of interfaces: (1) interfaces it *provides*, and (2) interfaces it *uses*. Through these interfaces a component can directly interact with other components. A component can *provide* or *use* the same interface type several times as long as it gives each instance a separate name.

A component uses an interface to represent a specific service (e.g., sending a message). In Figure 6.1, a component called TimerM has in total 3 interfaces: (1) *provides* the `StdControl` and `Timer` interfaces, and (2) *uses* a `Clock` interface. Interface details are all shown in Figure 6.2.

Place Figure 6.1 here.

Figure 6.1 Specification and graphical depiction of the TimerM component. *Provided* interfaces are shown above the TimerM component and *used* interfaces are below. Bidirectional arrows depict commands and events. The lightning depicts commands. [Levis06]

Place Figure 6.2 here.

Figure 6.2: Sample TinyOS interface types. [Levis06]

As shown in Figure 6.2, Interfaces are *bidirectional* and contain both *commands* and *events*. The *providers* of an interface implement the function of a *command*; while the *users* of an interface implement the function of an *event*. For instance, the `Timer` interface (Figure 6.2) defines two commands: “start”, “stop”, and an event is called “fire”.

Note: in this example, we do not use two separate interfaces (one for its *commands* and another for its *events*) to represent the interaction between the timer and its client. This is because that grouping them in the same interface makes the specification much clearer and helps to prevent bugs when wiring components together.

TinyOS is implemented in NesC language. Components written in NesC consist of two types: *modules* and *configurations.*

Modules can be used to call and /or execute *commands* and *events*. A module can declare private state variables and data buffers.

Configurations use interfaces to wire other components together. Figure 6.3 defines the TinyOS timer service. Its implementation is based on a *configuration* (called `TimerC`), which wires the timer *module* (`TimerM`) to the hardware clock *component* (`HWClock`). *Configurations* allow multiple *components* to be aggregated together into a single *macro component* that exposes a single set of *interfaces*.

Place Figure 6.3 here.

Figure 6.3 TinyOS’s Timer Service: the TimerC configuration. [Levis06]\

A *component* uses its *interfaces* (called *interface namespace*) to refer to the *commands*

and *events* that it uses. A configuration wires interfaces together by connecting the local names of different interfaces together. That is, a component invokes an interface without referring explicitly to its implementation. This makes it easy to introduce a new component in the component graph that uses the same interface.

An interface can be wired to other interfaces for multiple times. Figure 6.4 illustrates an example. The `StdControl` interface of `Main` is wired to `Photo`, `TimerC`, and `Multihop`.

Parameterized interface: In a component, *parameterized interface* can be used to export many instances of the same interface, parameterized by an identifier (typically a small integer). For example, in Figure 6.1, the `Timer` interface is a *parameterized interface* that uses an 8-bit id, which is an extra parameter. Such a parameterized interface allows the single `Timer` component to implement multiple, separate timer interfaces, one for each client component. Because the selection of IDs should be unique, a special unique keyword is used each time a unique identifier is needed.

Now we can see how a TinyOS application is built by NesC: First, we can use NesC to build a top-level configuration. Then we define different interfaces in that configuration to wire all needed components together. Figure 6.4 shows an application called `SurgeC.` It consists of the following components: Main, Photo, TimerC, Multihop, LedsC and SurgeM. Such an application periodically (`TimerC`) acquires light sensor readings (`Photo`) and sends them back to a base station using multi-hop routing (`Multihop`).

Place Figure 6.4 here.

Figure 6.4 The top-level configuration for the Surge application. [Levis06]

NesC is based on C syntax and executions. However, it is different from C in the following two aspects: (1) NesC does not use function pointers. Its compiler knows the precise call graph of a program. This enables cross-component optimizations for entire call paths, which can remove the overhead of cross-module calls as well as inline code for small components into its callers. (2) NesC does not support dynamic memory allocation. Components statically declare all of a program's state. This prevents memory fragmentation as well as runtime allocation failures.

6.1.3 Execution Model and Concurrency

A WSN can generate many events such as abnormal sensor data detection, low-battery alert, sensor sleep/wake-up, etc. The event-centric domain of WSNs requires fine-grain concurrency. And those events can arrive at any time. How do we handle those events? There could be two approaches:

Using traditional OS approaches such as Windows, which atomically enqueues incoming events and runs them in an appropriate time; Executing an event handler immediately in the style of active messages.

Because many of these events are important to WSN applications (for instance, a detected event needs immediate attention), the second approach is more suitable to WSN. While the core of the TinyOS execution model consists of run-to-completion tasks that represent the ongoing computation, the event handlers are signaled asynchronously by hardware.

NesC defines *tasks* as an explicit entity. When the program is executed, *tasks* are sent to the task scheduler for execution. The scheduler can execute tasks in any time order (such as FIFO: First In First Out), but must obey the *run-to-completion* rule, that is, once it picks up a task to execute, it must complete the execution.

Sometimes people use “atomicity” to represent the *run-to-completion* nature of a task. However, *tasks* are not atomic if an interrupt handler comes, or if the program needs to respond to *commands* and *events* that an interrupt invokes.

TinyOS defines synchronous and asynchronous codes (SC and AC) as follows:

Synchronous Code (SC): code that is only reachable from *tasks*.

Asynchronous Code (AC): code that is reachable from at least one interrupt handler.

Components often have a mix of SC and AC codes. TinyOS allows programmers to build responsive, concurrent data structures that can safely share data between AC and SC.

TinyOS uses non-preemption to eliminate *races* (i.e. competitions of CPU resources) among *tasks.* Unfortunately there are still potential races between SC and AC, as well as between AC and AC.

Typically when there is any update to the shared state that is reachable from AC, a data race could possibly occur. Then how do we ensure “atomicity” in such cases? We have two options to avoid the races: (1) we could convert all of the conflicting codes to tasks (here the codes are for SC only), or (2) we could use *atomic sections* to update the shared state. Here the *atomic section* is a small code sequence that is guaranteed to run *atomically*. For an atomic section, we cannot use any loops inside it, and we cannot turn on any interrupts.

In summary, we need to ensure a race-free program execution by the following approach:

Race-Free Invariant: Any update to shared state is either SC-only or occurs in an atomic section. NesC makes sure that the above *race-free invariant* is met during *compile* time. That is, the NesC compiler can prevent nearly all data races.

The data race should be avoided in any program due to the following reasons:

Data race can cause a class of very painful non-deterministic program bugs. If it is race-

free, composition can essentially ignore concurrency, that is, it won't care which components generate concurrency or how those components are wired together because the compiler will catch any sharing violations at compile time.

A wide variety of concurrent data structures and synchronization primitives could be enabled by strong compile-time analysis. NesC has several variations of concurrent queues and state machines to easily handle time critical actions directly in an event handler, even when they update shared state. For example, NesC always handles radio events are always in the interrupt handler until a whole packet has arrived, at which point the handler posts a *task*.

6.1.4 Active Messages

An important issue is: how does TinyOS handle wireless communications among sensors? A concept, called *Active Messages (AM)*, becomes the core TinyOS communication abstraction [TVon92]. An *AM* is a small (only 36 bytes long) packet associated with a 1-byte handler ID. When a sensor receives an *AM*, it immediately dispatches the message (using an event) to one or more handlers. And those handlers are registered to receive such *AM*s. Such *handler registration* is accomplished through static wiring and a parameterized interface, as described above.

TinyOS uses *AM* interfaces to achieve an unreliable, single-hop datagram protocol. *AM* interfaces also provide a unified communication interface to both the radio unit and the built-in serial port (for wired nodes such as base stations).

Multi-hop, reliable communications could be achieved by higher-level protocols on top of the AM interfaces. The exchange of *AM*s is also event-driven. AMs also tightly couple the

local CPU computations and radio communications.

6.1.5 Implementation Status

So far TinyOS has been used in a wide range of hardware platforms. It is suitable to many companies' sensor products. People have extended TinyOS environment by adding visualization, debugging, and support tools as well as a fine-grained simulation environment.

By installing TinyOS in both sensors and other machines such as desktops, laptops, and palmtops, we could build proxies between sensor networks and wired networks, allowing WSNs to integrate with server side tools implemented in Java, C, or MATLAB. TinyOS also allows us to build software interfaces to database engines such as PostgreSQL.

TinyOS is written in NesC language. NesC includes a tool that generates code to marshal between Active Message packet formats and Java classes.

6.1.6 Main features

Absolute Size: Surprisingly, TinyOS is really a tiny operating system because a base TinyOS environment only needs around 400 bytes. If including an associated C runtime primitives (such as floating-point libraries), TinyOS can fit in just over 1KB. If adding some NesC-based applications, in most cases they fit in less than 16KB. But for some extra large TinyOS applications, such as TinyDB, they still fit in less than 64KB so far.

Footprint Optimization: Besides using standard techniques (such as stripping the symbol table) to reduce code size, TinyOS also uses whole-program compilation to prune dead code.

Cross-component optimizations are used to get rid of redundant operations and module-crossing overhead.

As a matter of fact, NesC uses whole-program analysis to remove many of these boundary crossings and optimize entire call paths through extensive cross-component optimizations (Such optimizations include constant propagation and common sub-expression elimination). Such whole-program optimization makes NesC programs smaller and faster than unoptimized codes and the original hand-written C code that predates the NesC language.

Hardware/Software Transparency: TinyOS uses flexible component models to easily shift the hardware and software boundary. For instance, components can generate two types of *events*: software upcalls or hardware interrupts.

6.1.7 Low Power optimizations

TinyOS has a series of features to reduce energy consumption. For example, TinyOS uses split-phase operations and an event-driven execution model to reduce power usage because those operations avoid spinlocks and heavyweight concurrency (e.g., threads). TinyOS scheduler can command the micrcoprocessor into a low-power sleep mode whenever the t*ask* queue is empty. Such sleep mode further reduces power consumption.

6.2 LA-TinyOS - A Locality Aware Operating System for WSN [Huang07]

LA-TinyOS [Huang07] proposes a new WSN operating system that utilizes *locality* to improve event detection performance and at the same time reduce energy consumption. WSN

locality includes two types: *temporal* and *spatial* locality. They are defined as follows:

Temporal locality: When an event occurs, if it is really a WSN system anomaly, it could be observed again for a limited period of time during its first appearance. This is called *temporal locality*.

Spatial locality: If an anomaly is caused by a mobile object passing through a WSN, such an anomaly is likely to be observed again by neighboring nodes. This phenomenon is known as *spatial locality*.

Good Idea

If you could remember the Computer Architecture course, the design of caches also uses the same principle: (1) Based on temporal locality, if an instruction is used in one time, it is likely to be used in a near future. (2) Based on spatial locality, if an instruction is selected to be executed, its neighboring instructions are likely to executed, too. Therefore, the caches could be used to store such locality-aware instructions to speed up CPU execution.

Sometimes both temporal and spatial locality could occur in the same anomaly. For instance, in an environmental surveillance application, when a sensor detects an intruder. Such intrusion event is likely to be continuously raised by the same node for some time, i.e. temporal locality occurs. If the intruder moves around, the intruder may be detected by neighboring nodes shortly, i.e. spatial locality occurs.

Typically a sensor uses a task manager to sense an event. The task manager is activated periodically to sense the event. The detection period could be very long since the task manager usually senses nothing out of the ordinary.

The longer the detection period, the less energy consumption a sensor will have. However, when an anomalous event occurs, due to temporal and spatial locality, a shorter period

is favored since we need to increase the activation frequency of the task manager for observing the anomaly more closely.

It will be good if a task manager is *locality-aware*, i.e. it could *adjust its period automatically* based on the principle of temporal and spatial locality.

Unfortunately most of WSN operating systems do not provide kernel-level support to facilitate the development of *locality-aware* tasks. Therefore, most WSN applications perform no locality-aware tasks, or construct such tasks in user mode (i.e. not implemented in OS). Such user mode tends to be error-prone, less efficient, and redundant.

LA-TinyOS improves TinyOS by considering locality-aware task implementation. It achieves this by adding the `LocalityM` component to TinyOS. `LocalityM` provides an interface called `LocalityControl` for programmers to configure their locality elements. A data structure (see Table 6.1) is maintained by `LocalityM.` Such a data structure can be used to record all *locality* configurations. This table is called the *locality-configuration* table.

Place Table 6.1 here.

Table 6.1 The LA-TinyOS locality-configuration table [Huang07]

registerEvent(string EventName);
configureLocality (event table entry T e,
uint 8 TimerID,
uint 32 GracefulLength,
uint 8 HopCount,
(void_) FuncEnter,
(void_) FuncLeave);
triggerEvent(string EventName);

In the above 3 commands, `registerEvent` registers a new entry in the locality

configuration table, *configureLocality* specifies locality configurations, and *triggerEvent* is called when an anomalous event is detected to enter its locality.

An example code that uses *locality-configuration* data structure and commands is shown in Figure 6.5. It is a locality-aware Oscilloscope in LA-TinyOS.

In Line 8, we can see that an event named "A" is registered in the locality-configuration table. In Line 9 it calls configureLocality to specify locality configurations of this event. In Line 11 the reg operator associates dataTask with this event. If we look back at Table 6.1, its first row shows the locality configurations of this event.

In Line 14 we can see that when the sensed data exceeds a specific value (0x03B0), an anomaly is detected and the event "A" is triggered to enter its locality (Line 15).

In last column of Table 6.1 (called "Adaptation functions"), enterl and leavel are pointers to self-adaption functions provided by this application. It basically means that when "A" is detected, LA-TinyOS executes enterl to enter its locality. And it executes leavel to leave its locality.

Place Figure 6.5 here.

Figure 6.5 Part of locality-aware Oscilloscope [Huang07]

6.2.1 Change Timer to Respond to Temporal and Spatial Locality

Now let's see how LA-TinyOS updates its timer based on locality configurations. A component called TimerM maintains a list of software timers, as shown in Table 6.2. As shown in Table 6.2, the Timer ID tells us that whether it is a one-shot timer (i.e. it is terminated after it

expires) or a periodic timer, its default timer period (a counter's value) and the time left before it expires.

Place Table 6.2 here.

Table 6.2 The list of software timers in the TimerM component [Huang07]

If a timer interrupt fires, the interrupt handler of `HWClock` reduces the value in the *Time-to-Expired* field of each software timer. When the *Time-to-Expired* reaches zero, it means that the timer expires, a corresponding handler is then executed.

When an event enters its *locality*, LA-TinyOS uses the following data structure to change the period of the software timer:

```
setLocailityTimer (     unit8_t          TimerID,
unit32_t         ReducedPeriod);
```

In the above data structure, LA-TinyOS calculates its *reduced period* in order to show its adaptation to temporal locality. Its `TimerID` is obtained by looking up the `localityconfiguration` table. The default period will be saved in a kernel data structure. When an event leaves its locality, `setLocailityTimer` is again applied to reset its period.

In Table 6.1, the third column indicates the *graceful length* of each event. Whenever an anomaly is detected, this counter is reset to its full value. When an event enters its locality, LA-TinyOS reduces its *graceful length* counter at each timer interrupt.

Eventually the *graceful length* counter reaches zero. Then the "period" of its associated software timer (shown in Table 6.2, Column 4) is reset to its default value, to indicate this event is leaving its locality.

The above description was for *temporal locality* case. How does LA-TinyOS implement the *spatial locality*? It does this by broadcasting alerting messages. The number of broadcasting hops defines the alerting area. The number of hops is also available in the locality configuration table (Table 6.1). When a sensor receives an alerting message, it activates a corresponding anomalous event to enter its locality.

6.2.2 MultipleLevel Scheduler

As we mentioned in Section 6.1, TinyOS uses a non-preemptive FIFO (First in first out) scheduler. Now the issue is: such a simple scheduler cannot differentiate tasks that are associated with an anomalous event from tasks that are regular and non-urgent.

To solve such an issue, LATinyOS proposes a three-level scheduler without changing the TinyOS non-preemptive scheduling as follows:

Level 1: When a sensor detects an anomalous event, it registers associated tasks in the locality-configuration table. Those tasks are queued in the first-level and are scheduled to be executed with first priority.

Level 2: For spatial locality, tasks are associated with an event that is triggered to enter its locality by an alerting message. Those tasks are queued in the second-level FIFO queue.

Level 3: When Level 1 and Level 2 tasks do not occur, the non–urgent, normal tasks are served in the third-level FIFO scheduler.

The above three-level FIFO scheduler can make sure that LATinyOS performs tasks according to their importance.

Good Idea

Multi-level hierarchical tree concept has been used many problem solutions. Its basic idea is to avoid flat (i.e. one-level) topology where all nodes are treated as the same case. By distinguishing among different levels, we have the flexibility to handle different priorities.

6.2.3 LATinyOS Code Structure

The code structure of LA-TinyOS is shown in Figure 6.6. We can see that LA-TinyOS enhances TinyOS by adding a *LocalityM* module and a *multi-level scheduler*.

As we discussed before, the *locality-configuration* table is used to register and configure the events under either temporal or spatial locality. And the LocalityM component in turn uses original TinyOS kernel components to automatically adjust the detection "period" of a task when it enters and leaves its locality.

Without *LocalityM*, a programmer can still use the original TinyOS components (see Figure 6.7) to program a *locality*-aware application.

Place Figure 6.6 here.

Figure 6.6 The code structure of LA-TinyOS [Huang07]

Now let's summarize the main advantages of using LA-TinyOS to handle locality-aware applications:

First, LA-TinyOS kernel component houses all of its locality-aware code. The kernel execution is much more reliable than the riginal TinyOS implementation.

Second, LA-TinyOS allows a programmer to easily program locality-aware events since the programmer only needs to register a locality event during initialization phase, and then make a method call to enter its locality when an anomaly event is detected.

Finally, when more than one locality event occur, a programmer can use `LocalityM` to handle the locality-aware code of all events. Therefore LA-TinyOS allows a more efficient implementation with multiple locality events. In contrast, a TinyOS implementation needs to provide redundant locality-aware code for each event.

6.3 SOS [HanC05]

SOS, another improved version of TinyOS, is proposed in [HanC05]. It shows that WSN operating systems can achieve dynamic and general-purpose OS semantics without significant energy or performance sacrifices. Its main features include:

SOS has a common kernel and dynamic application modules. Those modules can be loaded or unloaded at run time. A system jump table is used by modules to send messages and communicate with the kernel. A module can also register function entry points for other modules to call.

SOS contains no memory protection, which is similar to TinyOS. However, it protects against common bugs. This is an improvement over TinyOS. Dynamic memory is used in SOS for the application modules and the kernel. This makes programming easier since it decreases complexity and increases temporal memory reuse.

Just as LA-TinyOS used 3-level task scheduler, SOS also proposes to use priority scheduling to move processing out of interrupt context and provide improved performance for

time-critical tasks.

The SOS kernel has dynamically linked modules, flexible priority scheduling, and a simple dynamic memory subsystem. SOS kernel services provide a higher level API (Applied Program Interface) to free a programmer from managing underlying services or re-implementing popular abstractions.

6.3.1 Modules

A SOS program uses *modules* (which are position-independent binaries) to implement a specific task or function. SOS consists of multiple interacting modules. From functionality viewpoint, the *modules* are similar to the concept of *components* in TinyOS. A SOS programmer implements the primary development including the drivers, protocols, and application components occurs at *the module layer*.

It is challenging to maintain modularity and safety in SOS without incurring high code overhead due to the loose coupling of modules. All SOS modules are self-contained and position-independent. They use clean messaging and function interfaces to maintain modularity. Most applications do not need to modify the SOS kernel unless low layer hardware or resource management capabilities need to be changed.

A. Module Structure

Figure 6.7 shows SOS module interactions. As we can see, SOS maintains a modular structure by implementing modules with well defined and generalized points of entry and exit.

Flow of execution enters a module either from (1) messages delivered from the scheduler, or (2) registered functions (for external use).

Place Figure 6.7 here.

Figure 6.7 Module Interactions [HanC05]

A module-specific handler function handles messages between modules. There are 2 parameters accepted by a handler function: (1) the message being delivered, and (2) the state of the module.

When a module is inserted, SOS kernel produces *init* message. The *init* message handler sets the module's initial state, which includes initial periodic timers, function registration, and function subscription.

When a module is removed, SOS kernel produces *final* message. The *final* message handler releases all sensor resources including timers, memory, and registered functions.

Besides the above *init* and *module* messages, there are also other module-specific messages including handling of timer triggers, sensor readings, and incoming data messages from other modules or nodes.

SOS handles messages asynchronously (i.e. using a queue to store those messages). Similar to TinyOS, the main SOS scheduling loop picks up a message from a priority queue, and delivers the message to the message handler of the destination module.

Module-specific operations need to run synchronously. SOS uses direct function calls between those modules. A function registration and subscription scheme implements those direct function calls.

RAM stores the modules' states. Modules are re-locatable in memory. The location of inter-module functions is exposed through a registration process.

B. Module Interaction

Messages are used to implement interactions between modules. Messaging enables asynchronous communication between modules. Messaging can also break up chains of execution into scheduled subparts. Those subparts are stored into a queue for scheduled execution.

Although the above messaging is flexible, its execution is slow. Therefore, SOS provides direct calls to functions that are registered by modules. Those direct function calls can bypass the scheduler to provide lower latency communication between the modules.

SOS uses function registration and subscription to implement direct inter-module communication and function calls from the kernel to modules. A *function control block* (FCB) is used to store key information about the registered function. FCB is created by the SOS kernel and indexed by the tuple {module ID, function ID}. The FCB includes a valid flag, a subscriber reference count, and prototype information.

The module ID and function ID are used to locate the FCB of interest, and type information is used to provide an additional level of safety. If the lookup succeeds, the kernel returns a pointer to the function pointer of the subscribed function.

A jump table shown in Figure 6.8 is used by modules that need to access kernel functions. Such a jump table also allows each module to remain loosely coupled to the kernel, rather than dependent on specific SOS kernel versions. It also allows the kernel to be upgraded

without the need of recompiling SOS modules. Thus the same module can run in a deployment of heterogeneous SOS kernels.

Place Figure 6.8 here.

Figure 6.8 Jump Table layout and linking in SOS [HanC05]

C. Module Insertion and Removal

Module insertion: A distribution protocol keeps listening to advertisements of new modules in the network. The distribution protocol that advertises and propagates module images through the network is independent of the SOS kernel. SOS currently uses a publish-subscribe protocol similar to a Mobile Oriented Applications Platform (MOAP). When an advertisement for a module is captured by the protocol, the protocol needs to check whether or not the module is the updated version of a module already installed on the node. It also needs to check if the node is interested in the module and has free program memory for the module.

To find out whether the above two conditions are true or not, the distribution protocol examines the *metadata* in the header of the packet. Such metadata has the following information: (1) the unique identity for the module, (2) the size of the memory required to store the local state of the module, and (3) version information used to differentiate each module version. The protocol will abort the module insertion if the SOS kernel finds that it is unable to allocate memory for the local state of the module.

During module insertion, a kernel data structure, which is indexed by the unique module ID included in the metadata, is created and used to store the absolute address of the handler. It also stores a pointer to the dynamic memory holding the module state and the identity of the

module. Finally the SOS kernel invokes the handler of the module by scheduling an *init* message for the module.

Module removal is initiated by the kernel that dispatches a *final* message. This message commands a module to gracefully release any resources it holds. Such a message also informs other modules that depend on the removed module. After the final message, the kernel performs garbage collection by releasing all resources as follows: dynamically allocated memory, timers, sensor drivers, and other resources owned by the module. FCB's are then used after a module removal to ensure the integrity of the platform.

6.3.2 Dynamic Memory

Dynamic memory allocation can use flexible queue length to adapt for the worst case scenarios and complex program semantics for common tasks, such as passing a data buffer down a protocol stack. Dynamic memory in SOS is based on a simple, best-fit fixed-block memory allocation with three base block sizes as follows.

The smallest block size is used for most SOS memory allocations including message headers. Larger block sizes are used for applications that need to move large continuous blocks of memory, such as module insertion. The largest block sizes are actually a linked list of free blocks, and can be used for any complex applications.

All data structures (such as queues, lists, etc.) in SOS dynamically grow and shrink at run time. The dynamic use and release of memory (i.e. dynamic memory), creates a system with effective temporal memory reuse. Dynamic memory can also dynamically tune memory usage to specific environments and conditions.

Modules can transfer memory ownership to reject data movement. SOS annotates dynamic memory blocks by using a small amount of data that is used to detect basic sequential memory overruns. Memory annotations can be used for post-crash memory analysis to identify suspect memory owners, such as a bad module that owns a great deal of system memory or overflowed memory blocks. SOS memory annotations also enable garbage collection on unload.

6.4. RETOS [Hojung07]

RETOS aims to provide a robust, reconfigurable, resource-efficient multithreaded operating system for WSN nodes. Figure 6.9 shows its overall architecture.

Place Figure 6.9 here.

Figure 6.9 RETOS architecture [Hojung07]

Although the *event-driven* approach is commonly adopted for sensor operating systems, mainly due to its efficient implementation in a resource-constrained hardware environment, however, in RETOS, application developers can manage the states of *tasks* and *events* explicitly, via a program split process as follows: RETOS explicitly separates applications from the kernel. An application is separately and dynamically loaded into the system (as does the kernel module). RETOS uses loadable module framework to achieve the kernel re-configurability.

6.4.1 Application Code Checking

RETOS uses a software technique, called *application code checking (ACC),* to perform static and dynamic code checks. The goal of ACC is to prevent user applications from accessing memory outside of its legal boundary and direct hardware manipulation.

To achieve such a goal, ACC inspects the *destination field* of machine instructions. The *source field* of instructions can also be examined to prevent the application from reading kernel or another application's data.

ACC uses *static* code checking to verify direct or immediate addressing instructions and pc-relative jumps during the compile time. (For details of instructions' addressing modes, please refer to the Assembly Language courses).

ACC uses *dynamic* code checking to verify the correct usage of indirect addressing instructions during runtime. Dynamic checking is also required for the Return instruction.

Figure 6.10 is the procedure of constructing trusted codes. Application source code is compiled to assembly code. The compiler then inserts checking codes to the place where dynamic code checking is required.

After dynamic code insertion, the *static* code checking is then conducted on the binary codes. When the compiler cannot detect some application errors, those missed errors will be reported to the kernel. After receiving those reported errors, the kernel informs users of the illegal instruction address and safely terminates the program.

Place Figure 6.10 here.

Figure 6.10 Generating Trusted Code [Hojung07]

6.4.2 Multithreading System

We know that TinyOS is an event-driven operating system, and a TinyOS programmer needs to worry about the optimal execution of their programs through explicit concurrency control.

However, RETOS uses different approach, i.e., multithreading, to inherently provide high concurrency with preemption and blocking I/Os characteristics of the underlying system. Although the multithreading approach is attractive, it is challenging to implement multithreading in resource-constrained sensor node environment. In multithreading environment, each thread needs a stack to maintain state variables. Scheduling scheme is used to perform context switching between the stacks. RETOS has carefully considered memory usage, energy consumption, and scheduling efficiency. RETOS has implemented single kernel stack and stack-size analysis, variable timer, and event-boosting thread scheduler, respectively.

(1) Minimizing Memory Usage

RETOS provides two techniques to reduce the memory usage for the kernel:

- *Single kernel stack* is used to reduce the size of thread stack requirement. The mechanism separates the thread stack into two types: *kernel stacks* and *user stacks*. RETOS has strict, controlled access to the kernel stack. This is to make sure that the system does not

arbitrarily interleave execution flow during the kernel mode, such as thread preemption. With thread preemption, hardware contexts are saved in each thread's control block based on kernel stack sharing.

- *Stack-size analysis* is used to assign an appropriate stack size to each thread autonomously. Accurate thread stack size needs to be estimated to reduce the memory usage. Stack-size analysis has been implemented in RETOS to automatically generate a minimal and system-safe stack for each thread.

(2) Reducing Energy Consumption with Variable Timer

Energy is consumed in multithreading computations, which include *timer management*, *context switching* and *scheduling operation*.

Timer management: In Multithreading system, from energy consumption viewpoint, a variable timer technique (instead of a fixed periodic timer) could be more energy efficient. Timer requests from threads are processed by the system timer, which then updates the remaining time independently of currently running threads. The timer interrupt interval can be reprogrammed by the variable timer. Such an interval is set to the earliest upcoming timeout among the time quantum of currently running thread.

Scheduling operation does not occur as frequent as passing messages between handlers in the event-driven system. In most WSN applications the *context switching* overhead is only a moderate issue.

(3) Event-aware Thread Scheduling

The thread scheduling is based on a priority-aware real-time scheduling interface to enable the kernel's dynamic priority management. Three policies are used to schedule RETOS

threads: SCHED_RR, SCHED_FIFO, and SCHED_OTHER.

An event-aware thread scheduling is used to increase the event response time of threads. To handle an important event, the scheduler directly boosts the priority of the thread that handles such a specific event. When an event occurs, the priority-boosted thread will be able to swiftly preempt other threads.

6.4.3 Loadable Kernel Module

Dynamic application loading is supported in RETOS. A memory relocation mechanism is used to support dynamic application loading. Memory relocation cannot be supported by a PIC (position independent code) approach.

Memory relocation mechanism is shown in Figure 6.11. A RETOS file format consists of a generic portion and a hardware-dependant section. It has compiled codes. If a sensor uses RETOS, its microcontroller needs to support different addressing features, such as relocation type and relative memory-accessing instructions. Therefore, such a file format has hardware-specific information to aid the memory relocation for the corresponding hardware.

Place Figure 6.11 here.

Figure 6.11 Simple RETOS relocation mechanism. Reading from left to right, this mechanism produces the smallest footprint of the OS possible (both in the NV storage and RAM) to free space\resources on the system. [Hojung07]\

Problems & Exercises

6.1 Compared to traditional OS (such as Microsoft Windows), what special characteristics does TinyOS have?

6.2 Explain TinyOS architecture.

6.3 What enhancements does LA-TinyOS make on the basis of TinyOS?

6.4 Explain SOS module interaction principle.

6.5 What benefits does RETOS have when using module relocation?

Event	Timer ID	Graceful Length	Tasks	Hops	Adaption Functions
"A"	1	2000	dataTask	2	enterl() / leavel()
"B"	2	1000	getMax	1	Null / Null

Table 6.1

Timer ID	Type	Status	Period	Time-to-Expired
0	ONE SHOT	on	300	240
1	REPEAT	on	1500	360
2	REPEAT	off	500	450

Table 6.2