5.1 Operating System: Tiny OS

- Tiny OS aims at supporting sensor network applications on resource-constrained hardware platforms
- To ensure that an application code has an extremely small footprint.
- Tiny OS chooses to have no file system, supports only static memory allocation, implements a simple task model, and provides minimal device and networking abstractions.
- To a certain extent, each Tiny OS application is built into the operating system.
- Like many operating systems, Tiny OS organizes components into layers.
- Intuitively, the lower a layer is, the "closer" it is to the hardware; the higher a layer is, the "closer" it is to the application.
- In addition to the layers, Tiny OS has a unique component architecture and provides as a library a set of system software components.
- A component specification is independent of the component implementation.
- Although most components encapsulate software functionalities, some are just thin wrappers around hardware.

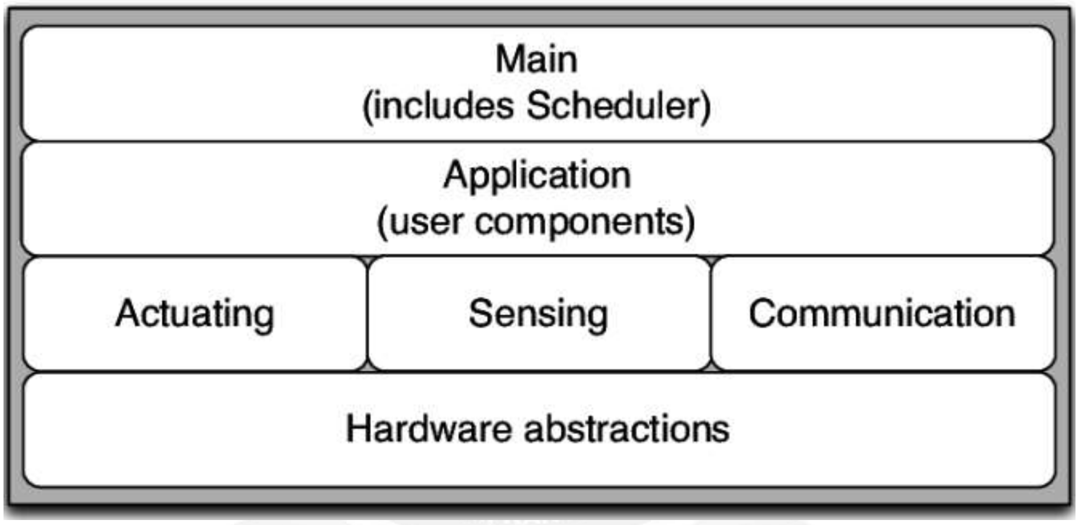

Fig: 5.1.1 Tiny OS Architecture

- An application, typically developed in the nesC language covered in the next section, wires these components together with other application-specific components.
- Let us consider a Tiny OS application example—Field Monitor, where all nodes in a sensor field periodically send their temperature and photosensor readings to a base station via an ad hoc routing mechanism.
- A diagram of the Field Monitor application is shown in above Figure, where blocks represent Tiny OS components and arrows represent function calls among them.
- The directions of the arrows are from callers to callees
- Hardware abstraction components are the lowest-level components.
- They are actually the mapping of physical hardware such as I/O devices, a radio transceiver, and sensors. Each component is mapped to a certain hardware abstraction.
- Synthetic hardware components are used to map the behaviour of advanced hardware and often sit on the hardware abstraction components.
- Tinyos designs a hardware abstract component called the Radio-Frequency Module(RFM) for the radio transceiver and a synthetic hardware component called radio byte, which handles data into or out of the underlying RFM.
- Higher-level components encapsulate software functionality, but with a similar abstraction.

- They provide commands, signal events and have internal handlers task threads and state variables.

Advantages of TinyOS

1. It requires very little code and a small amount of data.
2. Events are propagated quickly and the rate of posting a task and switching the corresponding context is very high
3. It enjoys efficient modularity.

NesC language.

- nesC is an extension of C to support and reflect the design of TinyOS v1.0 and above.
- It provides a set of language constructs and restrictions to implement TinyOS components and applications.

Component Interface

- A component in nesC has an interface specification and an implementation.
- To reflect the layered structure of TinyOS, interfaces of a nesC component are classified as provides or uses interfaces.
- A provides interface is a set of method calls exposed to the upper layers, while a uses interface is a set of method calls hiding the lower layer components.
- Methods in the interfaces can be grouped and named.
- For example, the interface specification of the Timer component in above Figure is listed in below Figure.
- The interface, again, independent of the implementation, is called Timer Module.
- Although they have the same method call semantics, nesC distinguishes the directions of the interface calls between layers as event calls and command calls.
- An event call is a method call from a lower layer component to a higher layer component, while a command is the opposite.
- Note that one needs to know both the type of the interface (provides or uses) and the direction of the method call (event or command) to know exactly whether an interface method is implemented by the component or is required by the component.
- The separation of interface type definitions from how they are used in the components promotes the reusability of standard interfaces.
- A component can provide and use the same interface type, so that it can act as a filter interposed between a client and a service.
- A component may even use or provide the same interface multiple times.
- In these cases, the component must give each interface instance a separate name, as shown in the Clock interface in Figure.

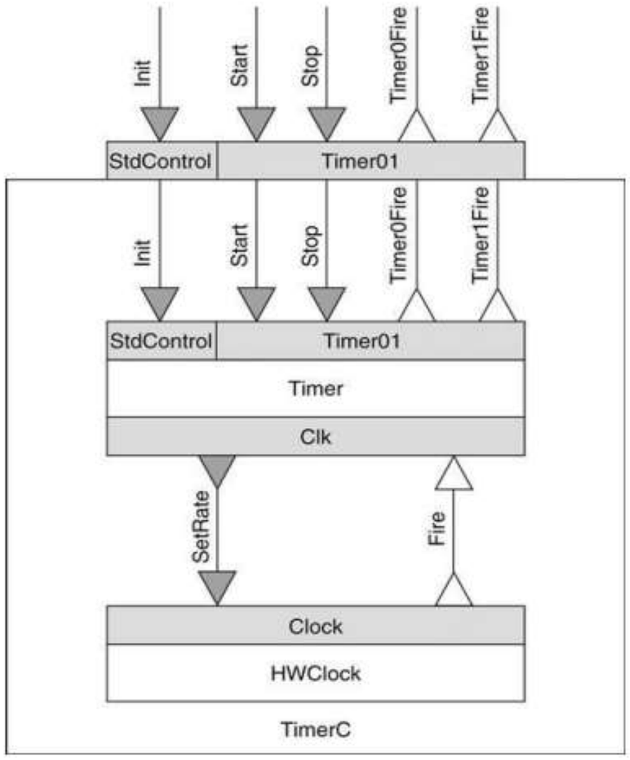

Fig: The TimerC configuration implemented by connecting Timer with HWClock.

Component Implementation

- There are two types of components in nesC depending on how they are implemented: modules and configurations.
- Modules are implemented by application code (written in a C-like syntax). Configurations are implemented by connecting interfaces of existing components.
- The implementation part of a module is written in C-like code.
- A command or an event bar in an interface foo is referred as foo.bar.
- A keyword call indicates the invocation of a command.
- A keyword signal indicates the triggering by an event. For example, the above Figure shows part of the implementation of the Timer component, whose interface is defined in Figure.
- In a sense, this implementation is very much like an object in object-oriented programming without any constructors.
- Configuration is another kind of implementation of components, obtained by connecting existing components.
- Suppose we want to connect the Timer component and a hardware clock wrapper, called HWClock, to provide a timer service, called Timer C.
- A conceptual diagram of how the components are connected and Figure a. shows the corresponding nesC code.
- In the implementation section of the configuration, the code first includes the two components and then specifies that the interface StdControl of the TimerC component is the StdControl interface of the TimerModule; similarly for the Timer01 interface.
- The connection between the Clock interfaces is specified using the -> operator.
- Essentially, this interface is hidden from upper layers.
- nesC also supports the creation of several instances of a component by declaring abstract components with optional parameters.
- Abstract components are created at compile time in configurations.

- TinyOS does not support dynamic memory allocation, so all components are statically constructed at compile time.
- A complete application is always a configuration rather than a module.
- An application must contain the Main module, which links the code to the scheduler at run time.
- The Main has a single StdControl interface, which is the ultimate source of initialization of all components.

Concurrency and Atomicity

- The language nesC directly reflects the TinyOS execution model through the notion of command and event contexts.
- The SenseAndSend component is intended to be built on top of the Timer , an analog-to-digital conversion (ADC) component, which can provide sensor readings, and a communication component, which can send a packet.
- When responding to a timer0Fire event, the SenseAndSend component invokes the ADC to poll a sensor reading. Since polling a sensor reading can take a long time, a split-phase operation is implemented for getting sensor readings.
- The call to ADC.getData() returns immediately, and the completion of the operation is signaled by an ADC.dataReady() event.
- A busy flag is used to explicitly reject new requests while the ADC is fulfilling an existing request. The ADC.getData() method sets the flag to true, while the ADC.dataReady() method sets it back to false.
- Sending the sensor reading to the next-hop neighbor via wireless communication is also a long operation.
- To make sure that it does not block the processing of the ADC.dataReady() event, a separate task is posted to the scheduler.
- A task is a method defined using the task keyword. In order to simplify the data structures inside the scheduler, a task cannot have arguments.
- Thus the sensor reading to be sent is put into a sensor Reading variable.
- There is one source of race condition in the Sense And Send, which is the updating of the busy flag.
- To prevent some state from being updated by both scheduled tasks and event-triggered interrupt handlers, nesC provides language facilities to limit the race conditions among these operations.
- In nesC, code can be classified into two types:
    - Asynchronous code (AC): Code that is reachable from at least one interrupt handler.
    - Synchronous code (SC): Code that is only reachable from tasks.
- Because the execution of TinyOS tasks are non pre-emptive and interrupt handlers pre-empt tasks, an SC is always atomic with respect to other SCs.
- However, any update to shared state from AC, or from SC that is also updated from AC, is a potential race condition.
- To reinstate atomicity of updating shared state, nesC provides a keyword atomic to indicate that the execution of a block of statements should not be preempted.
- This construction can be efficiently implemented by turning OFF hardware interrupts.
- To prevent blocking the interrupts for too long and affecting the responsiveness of the node, nesC does not allow method calls in atomic blocks.
- In fact, nesC has a compiler rule to enforce the accessing of shared variables to maintain the racefree condition
- If a variable x is accessed by an AC, then any access of x outside of an atomic statement is a compile-time error.
- This rule may be too rigid in reality.
- When a programmer knows for sure that a data race is not going to occur, or does not care if it occurs, then a no race declaration of the variable can prevent the compiler from checking the race condition on that variable.
- Thus, to correctly handle concurrency, nesC programmers need to have a clear idea of what is synchronous code and what is asynchronous code.
- However, since the semantics is hidden away in the layered structure of TinyOS, it is sometimes not obvious to the programmers where to add atomic blocks.