3. The Berkeley motes environment

3.1. Hardware

3.1.1. Overview

Researches at UC Berkeley have developed small sensor devices , called motes [Cro1] ,and an operating system that is especially suited to running on them , called TinyOS [Cro3, Cro4].

In detail there are two kind of motes developed: Mica2 (see Figure 3.1.1-1) and Mica2dot (see Figure 3.1.1-2) ; they are quite similar functionality but they deeply differ in form-factor .

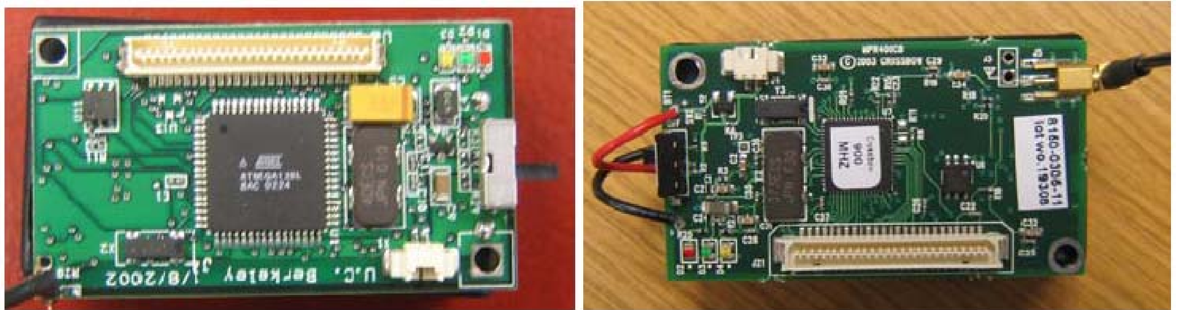

Figure 3.1.1-1

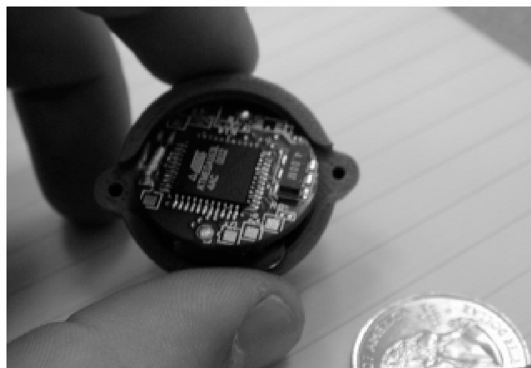

Figure 3.1.1-2

Mica2 and Mica2dot motes both have a 4Mhz , 8bit Atmel microprocessor. Their RFM ChipCon radios run at 19.2 Kbits/second over a single shared CSMA/CA (carrier-sense multiple access collision avoidance) channel. Like all wireless radio ChipCon radio is half-duplex, which means that they cannot detect collisions

because they cannot listen to their own traffic. Instead, they try to avoid collisions by listening to the channel before transmitting and backing off for a random time period when it is in use (for more detail see MAC layer in 3.2).

Motes have an external 32kHz clock that the TinyOS operating system can synchronize with neighboring motes to approximately +/- 1 ms to ensure that neighbors are powered up and listening when there is information to be exchanged between them.

Both generation of Mica motes (mica2 and mica2dot) are equipped with 512KB of non-volatile flash memory that can be used for logging and data collection.

Motes hardware has a 51-pin connector that allows expansion boards to be added. Typically a sensor board is placed in the connector which adds a suite of sensors to the device.

Table 3.1.1-1 summarizes hardware characteristics of Mica2 and Mica2dot nodes:

Mote Type	Renee	Mica	Mica2	Mica2Dot
Microcontroller				
Type	Atmega163	Atmega128	Atmega128	Atmega128
CPU Clock (Mhz)	4	4	8	4
Program Memory (KB)	16	128	128	128
RAM (KB)	1	4	4	4
Non-volatile Storage				
Size (KB)	32	512		
Radio Communication				
Radio	RFM TR1000		Chipcom CC1000	
Frequency	916		916 / 433	
Transmit Power Control	Programmable resistor potentiometer.		Programmable via CC1000 registers.	
Encoding	SecDed (Software)		Manchester (Hardware)	

Table 3.1.1-1

In the motes, the AVR interfaces with four hardware blocks (Radio, LEDS, Flash

Memory and Sensor board / Programming interface). The general hardware organization is presented in Figure 3.1.1-3:

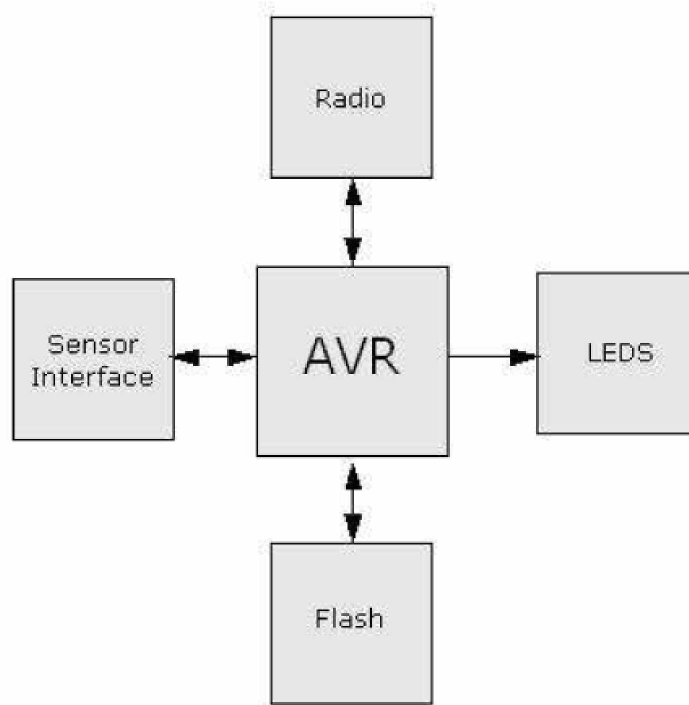

Figure 3.1.1-3

3.1.2. Processor

The microcontroller unit (MCU) is responsible for control of the sensors and the execution of communication protocols and signal processing algorithms on the gathered sensor data.

MCU supports various operating. Central unit of a sensor node is a low-power microcontroller that controls all functional parts of the node. Software for such a microcontroller has to be resource-aware on the one hand. On the other hand, several Quality-of-Service (QoS) aspects have to be met by sensor node software, such as latency, processing time for data fusion or compression, or flexibility regarding routing algorithms or MAC techniques.

Conventional software development for microcontrollers usually covers hardware abstraction layer, operating system and protocols and application layer. Often software for microcontrollers is limited to an application specific monolithic

software block optimized for performance and resource usage. Microcontrollers are often developed and programmed for a specific, well-defined task. This limitation of the application domain leads to high performance embedded systems even with strict resource constraints. Accordingly if the application domain of an embedded system changes often the whole microcontroller is replaced instead of writing and downloading a new program.

For sensor nodes, application specific microcontrollers are preferred instead of general purpose microprocessors. This is because of the small size and the low energy consumption of those controllers. However, requirements concerning a sensor node exceed the main characteristics of a conventional microcontroller and its software. The main reason for this is the dynamic character of a sensor node's task. Sensor nodes can adopt different tasks such as data acquisition, data forwarding, or information processing.

The processor within the Mica2 is an Atmel Atmega128 AVR. AVR is an 8-Bit Harvard architecture, with separate instruction and data memory.

AVR micro controllers provide several sleep modes. The purpose of these modes is to provide a way of suspending program execution when necessary, thereby reducing power consumption.

3.1.3. Leds

Three Programmable LEDs are connected to the AVR in the Mica2 motes. These may be used for status and output of digital values.

3.1.4. Flash Memory

In order to allow permanent storage and data logging in the motes, a 512KB Serial Flash memory chip is attached to one of the AVR's UART ports. If installed in conjunction with a simple co-processor, this secondary memory could be also used for over-the-air reprogramming of the main microcontroller.

3.1.5. Radio

Typical communication distances for low power wireless radios such as those used in mica2 and mica2dot motes range from a few meters to around hundred meters depending on transmission power and environmental conditions. Such short ranges mean that almost all real sensor network deployments must make use of multi-hop communication where intermediate nodes relay information for their peers.

In the case of mica2 and mica2dot the radio uses a Manchester encoding providing a delivered bandwidth of 19.2 kbps.

Manchester encoding is used to avoid synchronization errors which commonly occur in demodulation phase-locked-loop circuits when long sequences of 1's or 0's are received.

The sensor node's radio enables wireless communication with neighboring nodes and the outside world.

The Mica2 and Mica2dot use a low-power, single-chip UHF transceiver from Chipcom as its radio component. The CC1000 is designed for very low power and very low voltage wireless applications. The circuit is mainly intended for the ISM (Industrial, Scientific and Medical) and SRD (Short Range Device) frequency bands at 315, 433, 868 and 915 MHz, but can easily be programmed for operation at other frequencies in the 300- 1000 MHz range. The main operating parameters of CC1000 can be programmed via a serial bus, thus making CC1000 a very easy to use transceiver. CC1000 is configured via a simple 3-wire interface. There are 36 8-bit configuration registers, each addressed by a 7-bit address. A Read/Write bit initiates a read or write operation.

A full configuration of CC1000 requires sending 29 data frames of 16 bits each (7 address bits, R/W bit and 8 data bits). All registers are also readable. Data is transferred to and from the AVR microcontroller via a dedicated SPI (Serial Peripheral Interface) Bus, and the Radio generates one interrupt every 8 bits when in receive mode.

In general radio can operate in four distinct modes of operation: Transmit , Receive , Idle , Sleep (Off). So it is very important to completely shut down the

radio rather than transitioning to Idle mode when it is not transmitting or receiving data.

So features of mica2 radio include:
- Frequency selectable from 300-1000 Mhz;
- FSK modulation with data rates up to 19.2 Kbps;
- Hardware based Manchester encoding;
- Integrated bit synchronizer;
- -110 dBm sensitivity;
- selectable power states;
- digital control interface using special function register;

The radio module as default setting makes the following operational state after being issued `StdControl.init()` and `StdControl.start()`:
- set default frequency channel;
- set 19.2 kbps data rate;
- set high sensitivity mode (longest settling time);
- set 0 dBm transmit power;
- turn on radio in receiving mode;

The CC1000 uses a digital frequency synthesizer to select a particular send/receive channel. Specific control registers are programmed with values according to the channel and FSK separation used. Because of the nature of the synthesizer it is only capable of reproducing discrete frequencies in the operating range of the device.
The TinyOS stack and related tools take the guesswork out of tuning the CC1000 for the mica2 series motes. The stack will automatically compute the nearest channel for a given frequency and program the necessary register values (manual tuning) or use pre-determined values from a preset table (preset tuning).
The control path function `CC1000Control.TuneManual()` takes a desired frequency in Hz , computes the optimal achievable frequency , determines the necessary control register values , programs the CC1000 and calibrates the device. It returns the frequency, in Hz, of the actual channel. Defining the compiler flag

`CC1K_DEF_FREQ=x` sets the default frequency for the device when compiling the application. Using manual tuning does not affect the modem control registers (default data rate is 38.4 Kbaud/19.2 Kbps).

The control path function `CC1000Control.TunePreset()` takes a given index in this table, sets the register values and calibrates the device.

The data path provides a method of altering the duty cycle state of the radio to meet power constraints to set and determine the present duty cycle mode for both the receiving and transmitting side of the data path.

3.1.6. Sensing Hardware

Sensor transducers translate physical phenomena to electrical signals and can be classified as either analog or digital devices depending on the type of output they produce.

There are several sources of power consumption in a sensor , including:

- signal sampling and conversion of physical signals to electrical ones
- signal conditioning
- analog-to-digital conversion

While the modular design of the motes allows a wide range of analog and digital sensors to be attached to the Sensor Node, the reference sensor board for the mica platform is the "Mica Sensorboard " (see Figure 3.1.6-1).

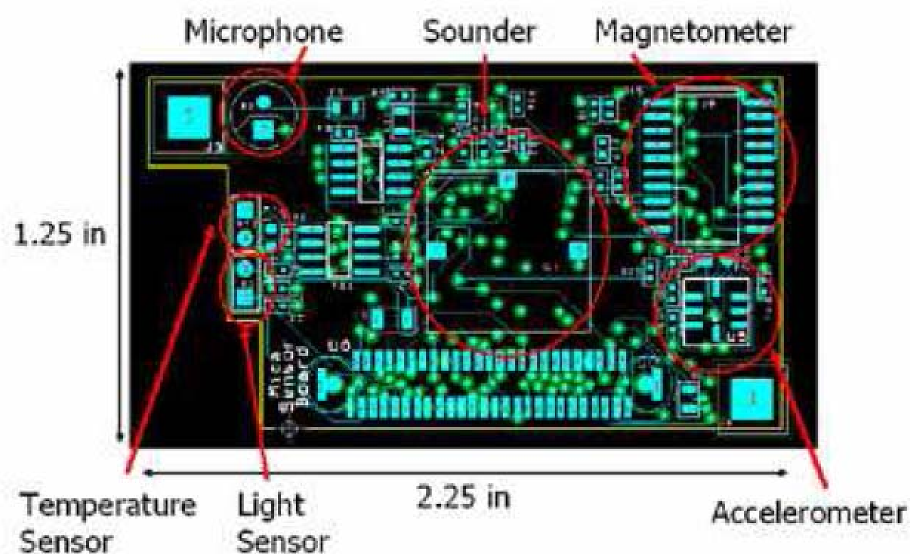

Figure 3.1.6-1

A variety of sensors have been interfaced with the motes [Cro2] ; a partial list includes sensors for light , surface and ambient temperature , acceleration , magnetic field , voltage , current (DC and AC) , sound volume , ultrasound (Figure 3.1.6-2), barometric pressure (Figure 3.1.6-3) , humidity , and solar radiation.

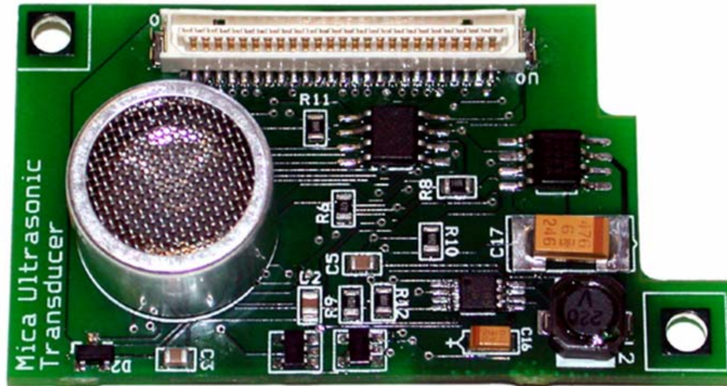

Figure 3.1.6-2

In addition to the above sensors, the board is capable of generating acoustic output, using its 4 kHz single tone buzzer. Optional hardware support to detect the generated tone on a receiving node is provided by an active bandpass filter and a LMC567 tone decoder from National Semiconductor, which has built in phase lock loop and adjustable threshold detection.

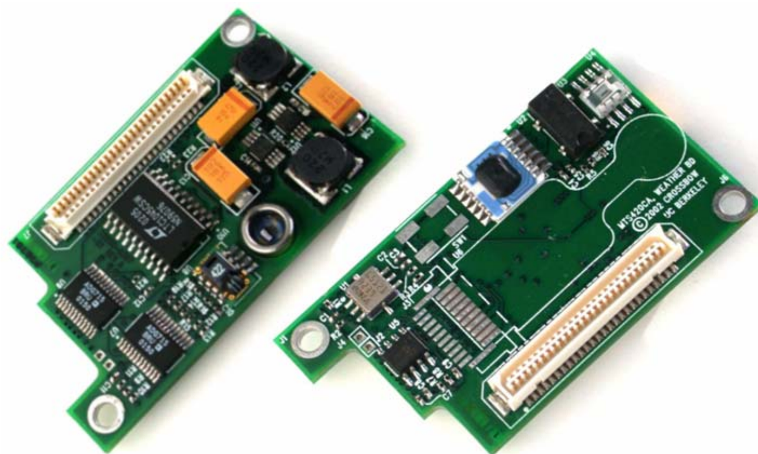

Figure 3.1.6-3

All modules in the sensor board can be power cycled independently, and are power isolated from the Mica's processor through an analog switch.

It is very important the cost of energy required to fetch samples from sensors.
The variations among sensors are dramatic, both in terms of power usage and time to obtain a sample. Some devices, such as pressure and humidity sensors , require as long as a second to capture a reading , which means that the per-sample energy costs are very high.
Other devices, such as the passive thermistor, whose resistance varies with ambient temperature , require only a few microseconds to sample , and thus contribute only a negligible amount to the total energy consumption of the motes.

There are a variety of sensor boards available. The sensor boards allow for a range of different sensing modalities as well as interface to external sensor via prototyping areas or screw terminals. The following Table 3.1.6-1 lists the currently available sensor boards for each mote family.

Part Number	Motes Supported	Sensors and Features
MTS101CA	MICA, MICA2	Light, Temperature, Prototype Area
MTS300CA	MICA, MICA2	Light, Temperature, Acoustic, and Sounder
MTS310CA	MICA, MICA2	Light, Temperature, Acoustic, Sounder, 2-Axis Accelerometer (ADXL202), and 2-Axis Magnetometer
MDA300CA	MICA2	Light, Humidity, General Purpose Interface for External Sensors
MDA500CA	MICA2DOT	General Purpose Interface

Table 3.1.6-1

In the measurements we refer there were used Crossbow sensor board MTS310CA for mica2 motes that includes the following sensing modalities:

- *microphone*

The microphone circuit has two principal uses. The first use is for acoustic ranging. The second use is for general acoustic recording and measurement. A novel application of the sounder and tone detector is acoustic ranging. In this application, a mote pulses the sounder and sends an RF packet via radio at the

same time. A second mote listens for the RF packet and notes the time of arrival by resetting a timer/counter on its processor. It then increments a counter until the tone detector detects the sounder. The counter value is the Time-of-Flight of the sound wave between the two motes. The Time-of-Flight value can be converted into an approximate distance between motes. Using groups of Motes with Sounders and Microphones, a crude localization and positioning system can be built

- *sounder*: the sounder is a simple 4 kHz fixed frequency piezoelectric resonator.
- *light and temperature*
- *2-Axis accelerometer* : The sensor can be used for tilt detection, movement, vibration, and/or seismic measurement
- *2-Axis Magnetometer*: magnetometer can measure the Earth’s field and other small magnetic fields. A useful application is vehicle detection. Successful test have detected disturbances from automobiles at a radius of 4,57 meters.

3.2. Software

3.2.1. Traditional OS Architectures

In traditional OS architectures we have Large memory & storage requirement , unnecessary and overkill functionality (address space isolation, complex I/O subsystem) , relative high system overhead (e.g, context switch) and these architectures require complex and power consuming hardware support.

Architecture must be:

- extremely small footprint;
- extremely low system overhead;
- extremely low system overhead;

So we don't have kernel but direct hardware manipulation , no process management but only one process on the fly , no virtual memory but single linear physical address space , no dynamic memory allocation bit memory assigned at compile time , no software signal or exception but function call instead

3.2.2. Introduction to TinyOS

TinyOS is a component-based operating system for sensor networks developed at UC Berkeley [Cro3]. TinyOS can be seen as an advanced software framework which has a large user community due to its open source character . The framework contains numerous pre-built sensor applications and algorithms for example multi-hop ad-hoc routing and supports different sensor node platforms. Programmers experienced with the C programming language can easily develop TinyOS applications written in a proprietary language called NesC

The design of TinyOS is based on the specific sensor network characteristics: small physical size, low-power consumption, concurrency-intensive operation, multiple flows, limited physical parallelism and controller hierarchy, diversity in design and usage, and robust operation to facilitate the development of reliable distributed applications. The main intention of the TinyOS developers was "retaining energy, computational and storage constraints of sensor nodes by managing the hardware capabilities effectively, while supporting concurrency-intensive operation in a manner that achieves efficient modularity and

robustness”. Therefore, TinyOS is optimized in terms of memory usage and energy efficiency. It provides defined interfaces between the components which reside in neighboring layers. A layered model is shown in Figure 3.2.2-1:

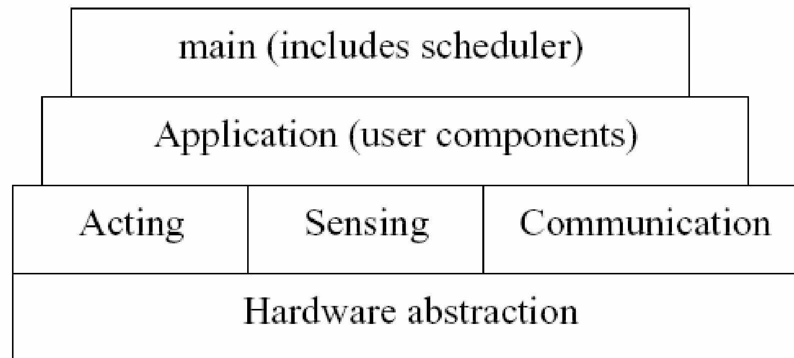

Figure 3.2.2-1

.

TinyOS utilizes an *event model* instead of a stack-based threaded approach, which would require more stack space and multi-tasking support for context switching, to handle high levels of concurrency in a very small amount of memory space. Event based approaches are the favorite solution to achieve high performance in concurrency intensive applications. Additionally, the event-based-approach uses CPU resources more efficiently and therefore takes care of the most precious resource, the energy. An event is serviced by an event handler. More complex event handling can be done by a task. The event handler is responsible for posting the task to the task scheduler. Event and task scheduling is performed by a *two-level scheduling structure*. This kind of scheduling provides that *events,* associated with a small amount of processing, can be performed immediately, while longer running *tasks* can be interrupted by events. *Tasks* are handled rapidly, however no blocking or polling is permitted.

3.2.3. *TinyOS design*

In order to achieve the necessary levels of concurrency, TinyOS uses a state machine based programming model as opposed to a thread based programming model. By making each component or service a state machine, we are able to make very efficient use of CPU and memory resources. Instead of having to allocate multiple stacks for each running application or service, we are able to share a single execution context amongst multiple sate machines. Each component uses events and commands to quickly transition from state to state. Logically, these state transitions are thought of a instantaneous, requiring very few CPU operations. Each component is temporarily allocated the execution context for the duration of these state changes. It has been added to this model the notion of tasks, which allow components to request the CPU execution context in order to perform long-running computations. These tasks get scheduled at a later date and run to completion. While they execute atomically with respect to other tasks, they can be periodically interrupted by higher priority events. Currently it is used a simple FIFO queue for scheduling, however an alternative scheduling mechanism could be easily added.

A secondary advantage of choosing to structure this programming model after finite state machines is that it propagates the hardware abstractions into software. Just as hardware bases state machine responds to changes on its I/O pins, our components respond to events and commands on their interfaces.

TinyOS consists of the tiny scheduler and a graph of components (Figure 3.2.3-1). *Components* satisfy the demand for modular software architectures. Every component consists of four interrelated parts: a command handler, an event handler, an encapsulated fixed-size and statically allocated frame, and a bundle of simple tasks. The frame represents the internal state of the component. Tasks, commands and handlers execute in the context of the frame and operate on its state. In addition, the component declares the commands it uses and the events it signals. Through this declaration, modular component graphs can be composed. The composition process creates layers of components. Higher layer components issue commands to lower level components and these signal events to higher level components. To provide an abstract definition of the interaction of two

components via commands and events, the bi-directional interface is introduced in TinyOS

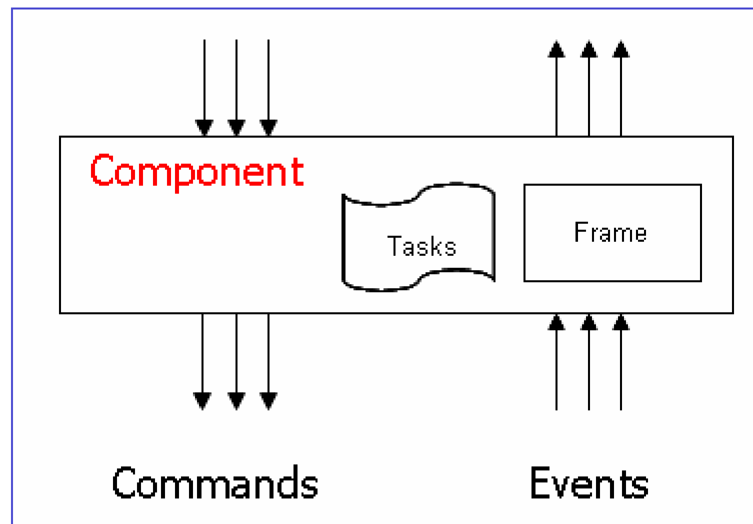

Figure 3.2.3-1

Commands are non-blocking requests made to lower layer components. A command provides feedback to its caller by returning status information. Typically, the *command handler* puts the command parameters into the frame and posts a task into the task queue for execution. The acknowledgment whether the command was successful, can be signaled by an event.
Commands deposit request parameters into the frame , are non-blocking , need to return status so postpone time consuming work by posting a task and can call lower level commando.

Event handlers are invoked by *events* of lower layer components, or when directly connected to the hardware, by interrupts. Similar to commands, the frame will be modified and tasks are posted. Both, commands and tasks, perform a small fixed amount of work similar to interrupt service routines.
Events can call commands, signal events, post tasks, can not be signaled by commands; events preempt tasks, not vice-versa , interrupt trigger the lowest level events and deposit the information into the frame.

Tasks perform the primary work. They are atomic, run to completion, and can only be preempted by events. Tasks are queued in a FIFO *task scheduler* to perform an immediate return of event or command handling routines. Due to the FIFO scheduling, tasks are executed sequentially and should be short. Alternatively to the FIFO task scheduler, priority-based or deadline-based schedulers can be implemented into the TinyOS framework.

Tasks perform computationally intensive work and handle multiple data flows

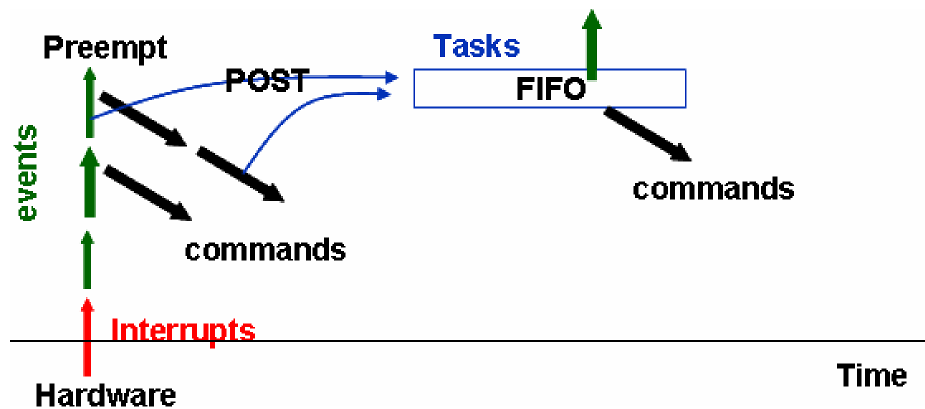

Figure 3.2.3-2

3.2.4. Communication

Communication stack in TinyOS is shown in Figure 3.2.4-1:

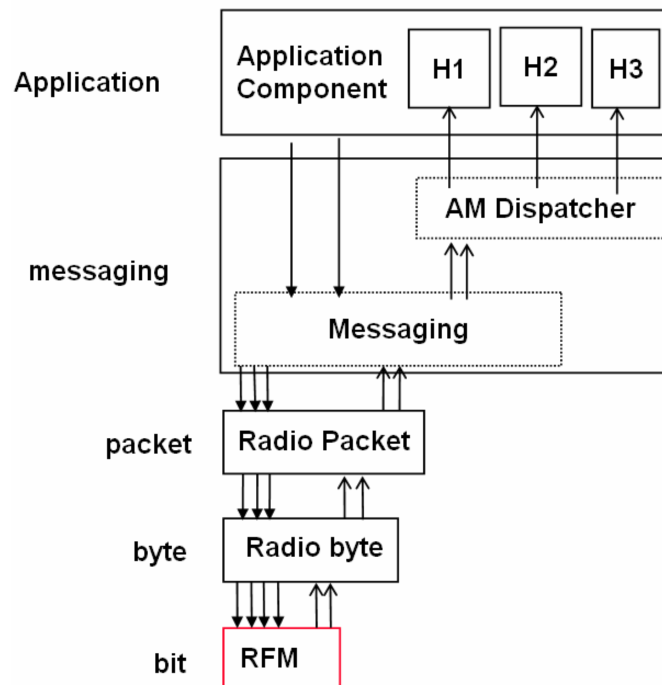

Figure 3.2.4-1

In details we take a look at each level:

- *RFM*

At this level we can set Operation Mode (transmitting or receiving) , set sampling rate , receive one bit , transmit one Bit , notify TX/RX is finished and shut down RFM.

- *Radio Byte*

This level is responsible of bit encoding (Manchester), error detection and correction, signal strength and to detect whether current channel is free to transmit, otherwise wait for random of clock ticks.

- *Radio Packet*

At this level there is 16-bit CRC check (drops packet if fails) and redundancy transmit.

- *Messaging*

The activities are packaging (dividing , combining) , routing , and support for special addresses (broadcast or UART interface)

- *AM dispatcher*

It's one byte message type used to direct packet to handlers. This layer follows a typical implementation:

```
if(msg.type == 0)   val =  Handler0(data);
if(msg.type == 1)    val = Handler1(data);
….
if(msg.type == 255) val = Handler255(data);
```

User can redefine handler names (e.g. `#define Handler5  NULL_FUNC`)

- *Application*

At this level we have content-based routing, consensus algorithm, location service, tracking and sensor data processing.

A simple profiling if we want to send 60 data bytes , we need to invoke (Figure 3.2.4-2) :

- messaging layer 1 times;
- packet layer > 2 times;
- byte layer > 60 times ;
- RFM > 480 times;

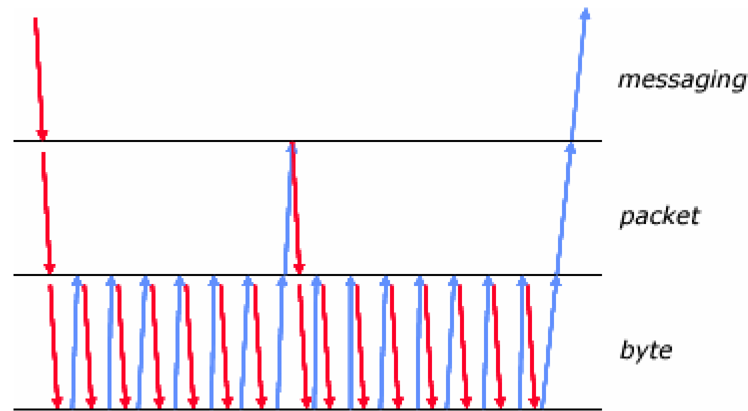

Figure 3.2.4-2

3.2.5. Active Message

In TinyOS legacy legacy communication (TCP/IP, sockets, routing protocols like OSPF) can't be used because traditional communications uses intensive bandwidth and are centered on "stop and wait" semantics.

In fact with socket/TCP/IP too much memory is used for buffering and threads ; furthermore data are buffered in network stack until application threads read it and application threads blocked until data is available.

With WSNs there it the need of real time constraints and low processing overhead.

Active message is a layer responsible of :

- Integrating communication and computation
- Matching communication primitives to hardware capabilities
- Provides a distributed event model where networked nodes send events to each other
- Closely fits the event-based model of TinyOS

Message contains a user-level handler to be invoked on arrival and data payload passed as argument. Message handlers are executed quickly to prevent network congestion and provide adequate performance. Event-centric nature enables network communication to overlap with sensor-interaction.

Active Message and TinyOS form "Tiny Active Messages" that support three basic primitives : best effort message transmission , addressing and dispatch.

With Active Message every message contains the name of an event handler ; the sender declares buffer storage in a frame , names a handler , requests transmission and does completion signal. On the other side receiver's event handler is fired automatically in a target node.

So there is no blocked or waiting threads on the receiver and we have a single buffering.

A typical send message code could be:

```
char TOS_COMMAND(INT_TO_RFM_OUTPUT)(int val){
     int_to_led_msg* message =
(int_to_led_msg*)VAR(msg).data;
     if (!VAR(pending)) {
       message->val = val;
       if
(TOS_COMMAND(INT_TO_RFM_SUB_SEND_MSG)(TOS_MSG_BCAST,
                  AM_MSG(INT_READING), &VAR(msg))) {
         VAR(pending) = 1;
         return 1;
       }
     }
     return 0;
}
```

Initially there is application access to message buffer (`VAR(msg.data`) through a cast to a defined format (`int_to_led_msg*`). Then there is a check (`if (!VAR(pending) )`) , it's build the message and request transmission through a destination identifier (`TOS_MSG_BCAST`) and a handler identifier (`AM_MSG(INT_READING)`). After that the state is marked busy (`VAR(pending)=1`)

3.2.6. NesC

The basic concepts behind nesC are:

• *Separation of construction and composition*: programs are built out of components, which are assembled ("wired") to form whole programs. Components define two scopes, one for their specification (containing the names of their interface instances) and one for their implementation.

Components have internal concurrency in the form of tasks. Threads of control may pass into a component through its interfaces. These threads are rooted either in a task or a hardware interrupt.

• *Specification of component behavior in terms of set of interfaces*. Interfaces may be provided or used by the component. The provided interfaces are intended to represent the functionality that the component provides to its user, the used interfaces represent the functionality the component needs to perform its job.

• *Interfaces are bi-directiona*l: they specify a set of functions to be implemented by the interface's provider (commands) and a set to be implemented by the interface's user (events). This allows a single interface to represent a complex interaction between components (e.g., registration of interest in some event, followed by a callback when that event happens). This is critical because all lengthy commands in TinyOS (e.g. send packet) are non-blocking; their completion is signaled through an event (send done). By specifying interfaces, a component cannot call the send command unless it provides an implementation of the sendDone event. Typically commands call downwards, i.e., from application components to those closer to the hardware, while events call upwards. Certain primitive events are bound to hardware interrupts.

• *Components are statically linked to each other via their interfaces*. nesC is designed under the expectation that code will be generated by whole-program compilers. This allows for better code generation and analysis. An example of this is nesC's compile-time data race detector.

• The concurrency model of nesC is based on run-to-completion tasks, and interrupt handlers which may interrupt tasks and each other. The nesC compiler signals the potential data races caused by the interrupt handlers.

As it was said the nesC model is formed by interfaces and components.

An interface could be used or can be provided ; components are modules or configurations.

An application is a graph of components (Figure 3.2.6-1).

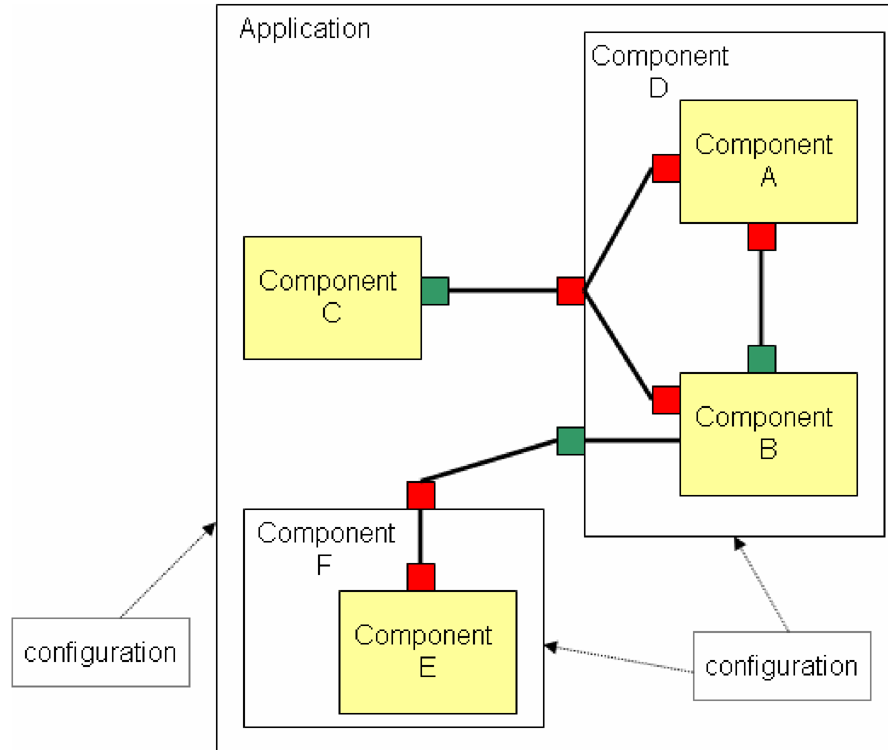

Figure 3.2.6-1

Interfaces are used for grouping functionality like split-phase operation (send , sendDone) or standard control interface (init , start , stop). They describe bi-directional interaction ; interface provider must implement commands while interface user must implement events.

For instance the interface `clock.nc` can contains:

```
interface Clock {
   command result_t setRate (char interval, char
scale);
   event result_t fired ();
}
```

Modules implement a component's specification with C code . An example is the following `Mycomp` module:

```
module MyComp {
   provides interface X;
   provides interface Y;
   uses interface Z;
}
implementation {
…// C code
}
```

Configurations implements a component by wiring together multiple components. Wiring means connect interfaces, commands, events together :

```
configuration MyComp {
   provides interface X;
   provides interface Y;
   uses interface Z;
}
implementation {
…// wiring code
}
```

Obviously connected elements must be compatible (interface-interface, command-command, event-event)

3.2.7. TinyOS application

In Figure 3.2.7-1 there is a representation of a complete application. The lowest layer of components directly correspond to the hardware of the system. They simply map the physical hardware into our software based component model. The user application sits at the top of the hierarchy issuing commands down into the lower level components and responding to events propagating up from the system components. During execution, all events are directly or indirectly triggered from the propagation of hardware events up through the component graph. This comes directly from the state machine based programming model, where state changes are the result of changes on the input pins.

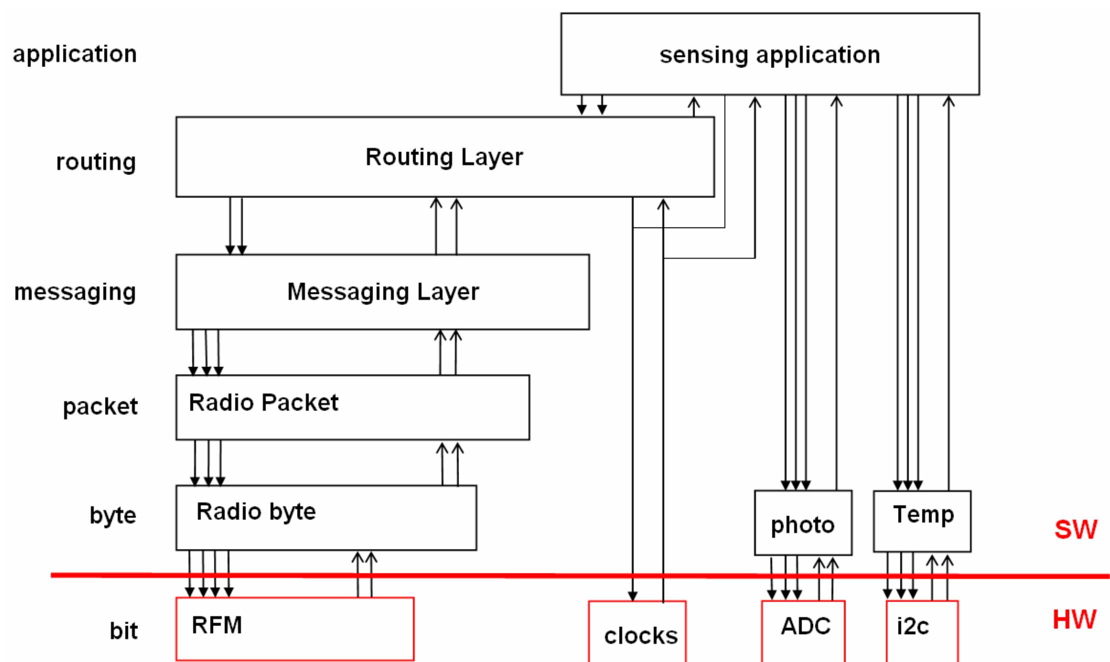

Figure 3.2.7-1

Now let's look at "Blink" application ; this application simply causes the red LED on the mote to turn on and off at 1 Hz.

Blink consists of two components: a module, called `BlinkM.nc`, and a configuration, called `Blink.nc`. All applications require a single top-level configuration, which is typically named after the application itself. In this case `Blink.nc` is the configuration for the Blink application and the source file that

the NesC compiler uses to generate the executable for the mote. `BlinkM.nc`, on he other hand, actually provides the implementation of the Blink application. `Blink.nc` is used to wire the `BlinkM.nc` module to other components that the Blink application requires. The reason for the distinction between modules and configurations is to allow a system designer to quickly "snap together" applications. For example, a designer could provide a configuration that simply wires together one or more modules, none of which she actually designed. Likewise, another developer can provide a new set of "library" modules that can be used in a range of applications.

The nesC compiler, ncc, compiles a nesC application when given the file containing the top-level configuration. Typical TinyOS applications come with a standard Makefile that allows platform selection and invokes ncc with appropriate options on the application's top-level configuration.

Let's look first at the module `Blink.nc`:

```
configuration Blink {
     implementation {
     components Main, BlinkM, SingleTimer, LedsC;
     Main.StdControl -> BlinkM.StdControl;
     Main.StdControl -> SingleTimer.StdControl;
     BlinkM.Timer -> SingleTimer.Timer;
     BlinkM.Leds -> LedsC;
     }
}
```

The first two lines,

```
configuration Blink {
}
```

simply state that this is a configuration called Blink. Within the empty braces here it is possible to specify uses and provides clauses, as with a module.

The actual configuration is implemented within the pair of curly bracket following key word implementation. The components line specifies the set of components that this configuration references, in this case `Main`, `BlinkM`, `SingleTimer`, and `LedsC`. The remainder of the implementation consists of connecting interfaces used by components to interfaces provided
by others.
`Main` is a component that is executed first in a TinyOS application. To be precise, the `Main.StdControl.init()` command is the first command executed in TinyOS followed by `Main.StdControl.start()`. Therefore, a TinyOS application must have Main component in its configuration. `StdControl` is a common interface used to initialise and start TinyOS
components.
The following two lines in Blink configuration:

```
Main.StdControl -> SingleTimer.StdControl;
Main.StdControl -> BlinkM.StdControl;
```

wire the `StdControl` interface in Main to the `StdControl` interface in both `BlinkM` and `SingleTimer.StdControl.init()` and `BlinkM.StdControl.init()` will be called by `Main.StdControl.init()`. The same rule applies to the `start()` and `stop()` commands.
The `BlinkM` module uses the interface Leds, so `Leds.init()` is called explicitly in `BlinkM.init()`.
nesC uses arrows to determine relationships between interfaces.
The line `BlinkM.Timer -> SingleTimer.Timer` is used to wire the Timer interface used by `BlinkM` to the Timer interface provided by `SingleTimer`. `BlinkM.Timer` on the left side of the arrow is referring to the interface called Timer (*/tos/interfaces/Timer.nc*), whereas `SingleTimer.Timer` on the right side of the arrow is refering to the implementation of Timer (*/tos/lib/SingleTimer.nc*).
Now let's look at the module `BlinkM.nc`:

```
module BlinkM {
     provides {
     interface StdControl;
     }
     uses {
          interface Timer;
          interface Leds;
     }
}
```

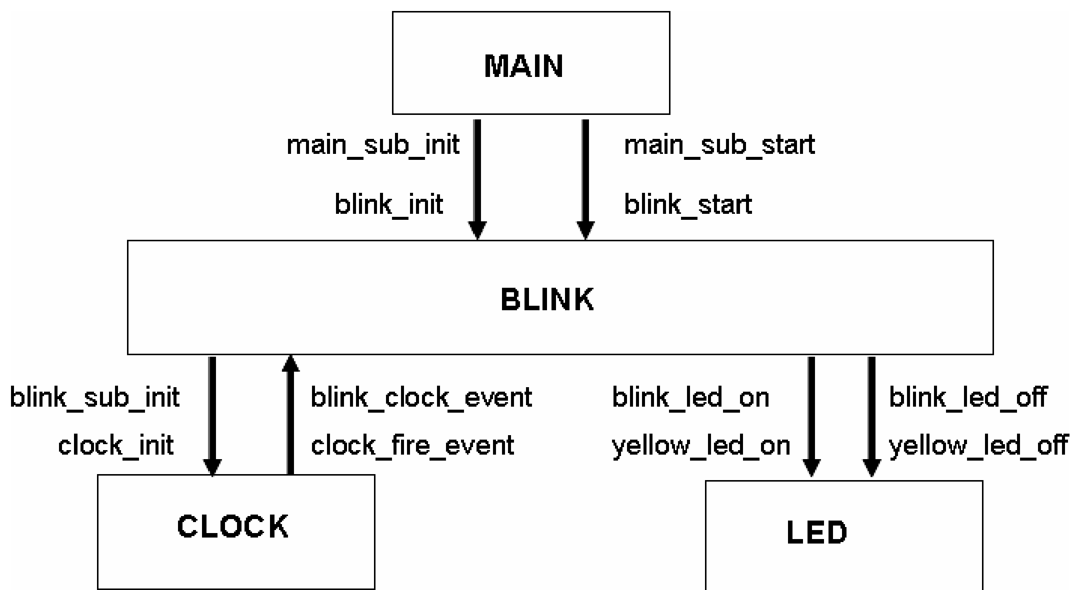

Figure 3.2.7-2

The first part of the code states that this is a module called BlinkM and declares the interfaces it povides and uses. The `BlinkM` module provides the interface `StdControl`. This means that `BlinkM` implements the `StdControl` interface. The `BlinkM` module also uses two interfaces: `Leds` and `Timer`. This means that `BlinkM` may call any command declared in the interfaces it uses and must also implement any events declared in those interfaces. The Leds interface defines several commands like `redOn()`, `redOff()`, and so forth, which turn the different LEDs (red, green, or yellow) on the mote on and off.
Because `BlinkM` uses the Leds interface, it can invoke any of these commands. However Leds is just an interface: the implementation is specified in the

`Blink.nc` configuration file. An event is a function that the implementation of an interface will signal when a certain event takes place. In this case, the `fired()` event is signaled when the specified interval has passed. This is an example of a bi - directional interface: an interface not only provides commands that can be called by users of the interface, but also signals events that call handlers in the user. A module that uses an interface must implement the events that this interface uses.

Let's look at the rest of `BlinkM.nc` to see how this all fits together:

```
implementation {
	command result_t StdControl.init() {
		call Leds.init();
		return SUCCESS;
		}
	command result_t StdControl.start() {
		return call Timer.start(TIMER_REPEAT, 1000) ;
		}
	command result_t StdControl.stop() {
		return call Timer.stop();
		}
	event result_t Timer.fired(){
		call Leds.redToggle();
		return SUCCESS;
		}
}
```

As we see the `BlinkM` module implements the `StdControl.init()`, `StdControl.start()`, and `StdControl.stop()` commands, since it provides the StdControl interface. It also implements the `Timer.fired()` event, which is necessary since BlinkM must implement any event from an interface it uses. The `init()` command in the implemented StdControl interface simply initialises the Leds subcomponent with the call to `Leds.init()`. The `start()` command invokes `Timer.start()` to create a repeat timer that

expires every 1000 ms. `stop()` terminates the timer. Each time `Timer.fired()` event is triggered, the `Leds.redToggle()` toggles the red LED.

3.2.8. TinyOS MAC layer

Wireless sensor networks impose additional challenges on Media Access Control (MAC) mechanisms. In both wired and wireless networks media access has been at the core of effective communication. Since WSNs are a new domain of wireless application, traditional methods (such as 802.11 or Ethernet MAC) rarely minimize power consumption or provide enough control to the application [Woo].
In shared medium networks, one of the fundamental tasks of a MAC layer is to avoid collisions between two interfering nodes. It allocates the channel to the nodes efficiently, so that each node can communicate with a bounded waiting time and with as little overhead as possible. The important attributes for traditional MAC are fairness, latency, throughput and bandwidth utilization.
In contrast, the important attributes of a MAC protocol for WASN are energy efficiency and scalability towards size and topology change. The major sources of energy wastage are [Pol03b]:

- *Collision*: collision results in corruption of a packet and subsequent retransmission leading to increased energy consumption as well as latency.
- *Idle listening & overhearing*: listening for either possible packets or packets destined for other nodes leads to wastage of energy. Idle listening consumes significant energy comparable to actually receiving a packet.
- *Control packets overhead*: increased control overhead leads to increased energy usage in direct proportion.

The mica2 series of sensor motes uses the ChipCon model CC1000 single-chip RF transceiver.
Figure 3.2.8-1 shows the overall component and configuration architecture for the CC1000 stack.

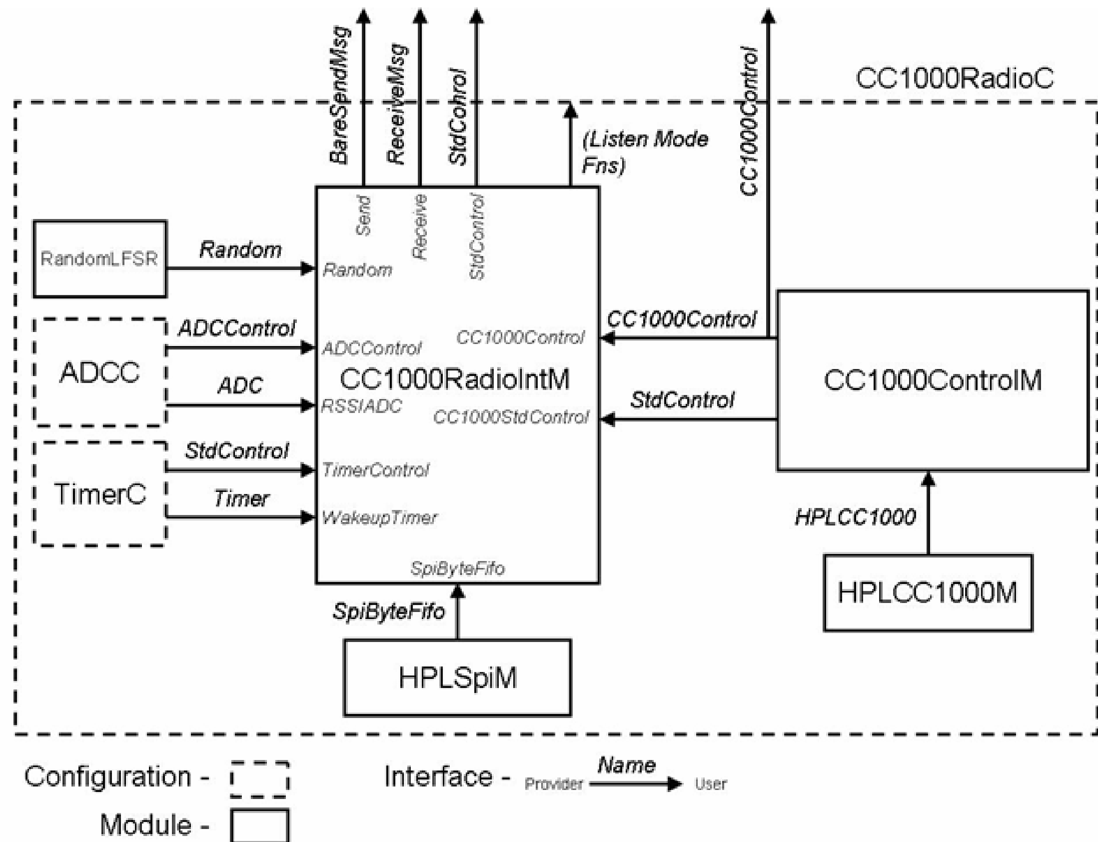

Figure 3.2.8-1

`CC1000RadioIntM` module is the data path module for the CC1000. This module provides transmit and receive data movement using CSMA/CA based contention avoidance schemes. In the receive mode , the module accepts bytes of data from the radio via the SPI interface and performs the necessary preamble detection , synchronization and CRC calculation/checks. When a packet has been received , it posts a task which signals a receive event. While the stack computes and checks the CRC , it does not drop the packet based on a bad CRC. Rather it sets the `crc` field of the `TOS_Msg` struct to be '1' if the CRC is valid.

To transmit the stack checks to see if the channel is clear by searching for a preamble AND monitoring received signal strength (via the ADC). When the channel is clear , it toggles the radio and pushes out the preamble , sync and payload bytes. The stack also implements low power listening modes and uses the Timer components to trigger RX and TX periods.

`CC1000ControlM` is the control path module for CC1000 operation. This module provides all of the management functions of the radio including tuning ,

toggling between TX/RX operation, selecting power states , and reading special I/O pins. It provides these features via the `CC1000Control` interface. While most of the functionality of this stack is used by the data path , it is also exported outside the radio stack for applications that may need to take direct control of the stack

If we look at the entire radio stack (see figure Figure 3.2.8-2) we see several other components:

- `SpiByteFifo`: provides a byte-level abstraction to the radio. In essence , it uses the Serial Peripheral Interface (SPI) of the ATMega processor to shift out bits to the radio when sending and shift in bits from the radio when receiving
- `RandomLFSR` : returns a 16 bit random number. It is used in `CC1000RadioIntM` module to determine the lenght of the backoff state in radio clock ticks.

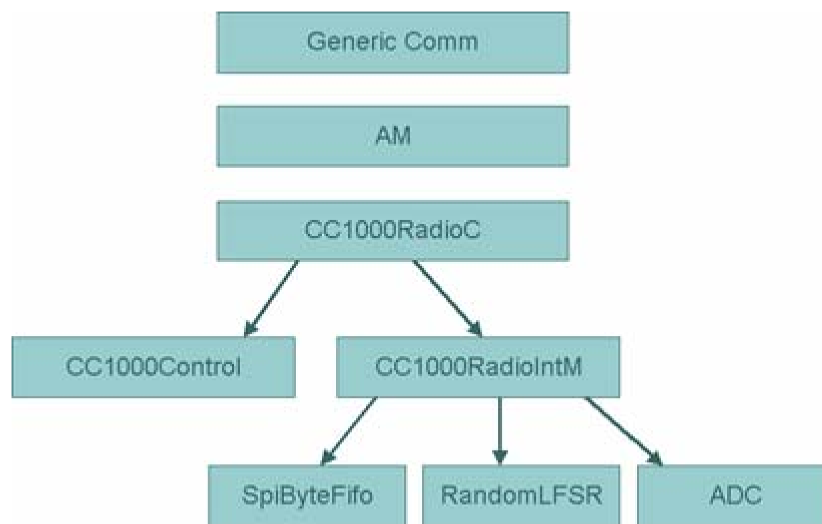

Figure 3.2.8-2

The network stack is initialized by calling init() in `CC1000RadioIntM`. In turn , `RandomLFSR` and `SpiByteFifo` is initialized. RandomLFSR also initializes the seed from the ID of the mote for the random number generator. `CC1000RadioIntM` sets its `MACdelay` field to -1 and sets the radio hardware receiving. While the entire network stack is idle it shifts in the bit received into a

buffer and checks for the preamble. Preamble/start symbol will be discussed in further detail below.

To understand the behavior of MAC layer it's important to discuss about TinyOS message's structure :

```
typedef struct TOS_Msg
{
     uint16_t addr;
     uint8_t type;
     uint8_t group;
     uint8_t lenght;
     int8_t data[TOSH_DATA_LENGTH];
     uint16_t crc;
     uint16_t strength;
     uint16_t time;
} TOS_Msg;
```

It consists of an unsigned two byte field `addr`, followed by three unsigned single byte fields `type`, `group`, and `length` `addr` specifies a moteID or the broadcast address (0xffff). When the `CC1000RadioIntM` receives a packet, the packet is passed to the AM level. If `addr` is neither the broadcast address nor the address of the mote receiving the packet, the packet is dropped.
The group field specifies a channel for motes on a network. If a mote receives a packet sent by a mote with a different group field, the packet is dropped at the AM level. The default group is 0x7d. The `type` field specifies which handler to be called at the AM level when a packet is received. The `length` field specifies the length of the data portion of the `TOS Msg`. Packets have a maximum
payload of 29 bytes.
The next field in the `TOS Msg struct` is the `data` portion. It consists of an array of 29 bytes (as specified by TOSH DATA LENGTH). The unsigned two byte field crc follows. When sending, the

CRC is incrementally calculated as each byte of the packet is transmitted. The maximum length of a transmitted TOS Msg is 36 bytes (addr(2 bytes) + type(1 bytes) + group(1 bytes) + length(1bytes) + data(29 bytes) + crc(2 bytes = 36 bytes)).

The `strength` and `time` fields are not transmitted; they are meta-data about the packet. The last two fields of TOS Msg are the unsigned two byte `strength` and unsigned two byte `time` fields. When the network stack finishes sending a packet, it will return the `TOS MsgPtr` to the application that issued the send request.

The state-machine that CC1000RadioIntM implements is shown in Figure 3.2.8-3. Looking at the figure we can discuss the various states:

- *disabled state*: it's the initial state resulting of `init()` call. However after `start()` call we can switch to p*ower down state*.
- *power down stat*e : it's the state in which radio is off and processor is a low power consumption mode
- *idle state* : in this state radio is on and it's possible to begin a transmission or a reception. If timer fires because of `MacDelay=0` and there is something to transmit we switch to *pretx_state*. However if a preamble is detected (thus 17 special bytes) we switch to *sync_state* to receive synchronization byte.
- *pretx_state*: in this state we make an RSSI measure to monitor the channel. If channel is idle we switch to *tx_state* (in the sub state *tx_s_preamble*) for beginning transmission. Otherwise we return to *idle_state*.
- *sync_state*: in this state we expect synchronization byte. If received we go in *rx_state* else turn back to *idle state*.
- t*x_state*: this state is composed by six sub states for preamble, synchronization byte, data, crc and flush transmission. In particular flush transmission acts as ending sequence. When it's achieved *tx_s_done* state radio is put in receiving mode ,it is posted the task `PacketSent()` and we switched again to *idle_state*.
- *rx_state*: in this state we receive data byes checking the CRC. When finished it's posted the task `PacketReceived()` and we turn back to *idle_state*.

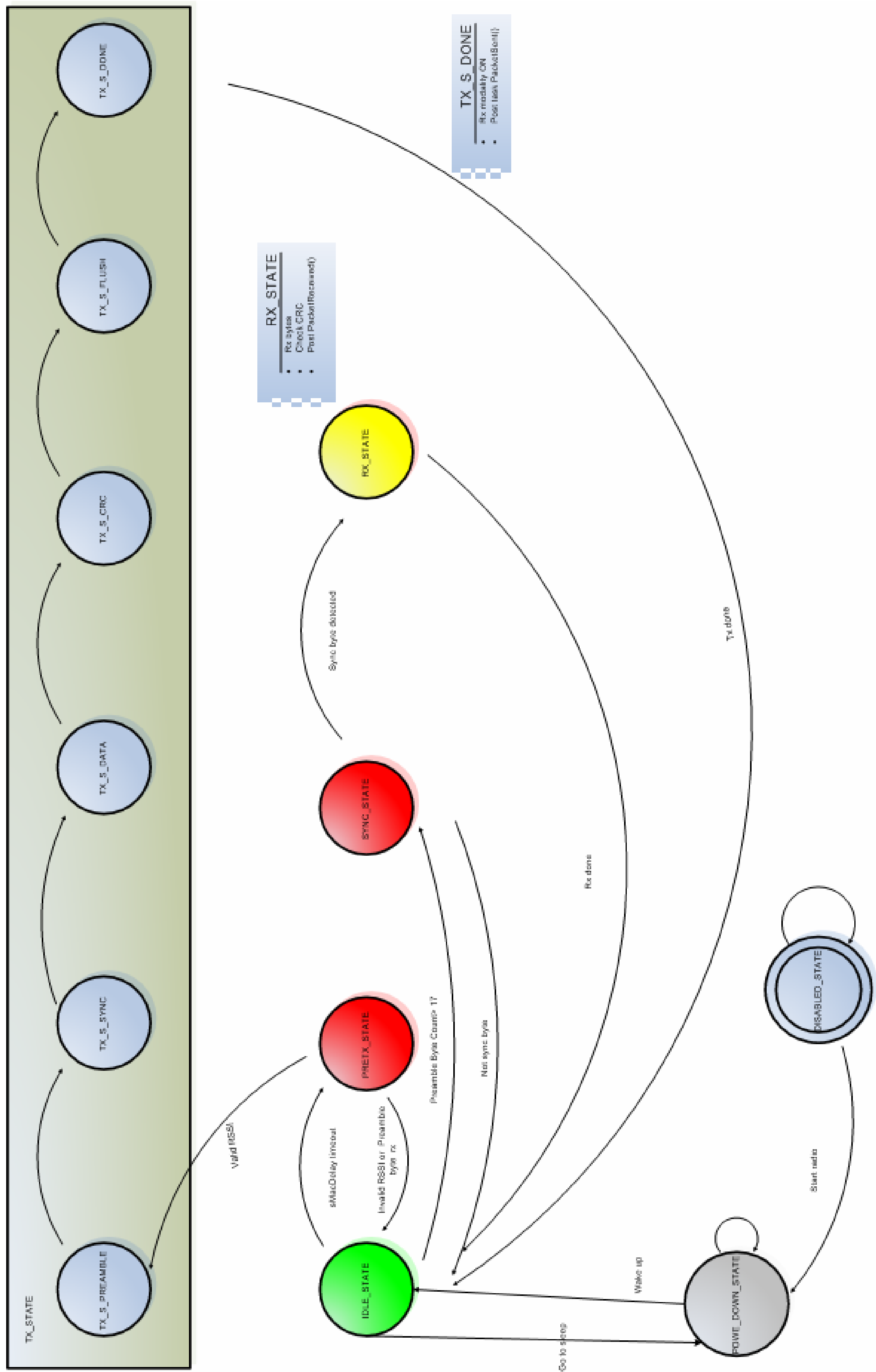

Figure 3.2.8-3

The noise floor on a wireless channel is estimated by examining the characteristics of the channel from the received signal strength indicator (RSSI). The noise floor is typically centered around a mean with some standard deviation. Noise floor estimation is further complicated by errors in the receiver packet detection. Noise floor estimation must be resilient to outliers and noisy signals.
RSSI output of the CC1000 offers a voltage which is inversely proportional to the sensed level of RSSI in the channel ; for this reason if RSSI measure is major than a threshold channel is considered idle (see Figure 3.2.8-4)

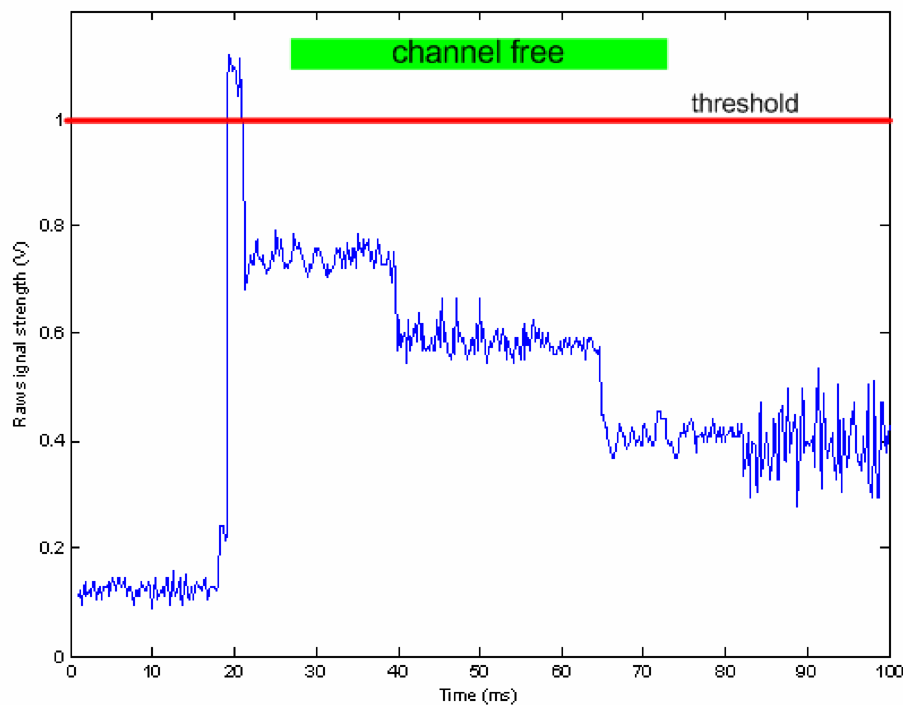

Figure 3.2.8-4

Of primary concern in achieving high channel utilization in CSMA schemes is the initial and congestion backoff algorithm. The *initial backoff* is the time a transmitter delays from the point that a packet is submitted for transmission to the first time status of channel is evaluated. The *congestion backoff* is the time a transmitter delays after checking the status of a channel and determining that the channel is busy. The initial backoff affects the maximum load that each node may offer to the channel. Longer initial backoffs result in less throughput per node and more nodes to saturate the channel. Long initial backoffs prevent against collision

when responding to synchronized transmissions (such as broadcast messages). Conventional schemes including Ethernet use and exponentially increasing congestion backoff but in wireless sensor networks primary concern is power consumption and secondary one is channel fairness. For example, bulk data transfers may not want any initial backoff, responses to broadcast messages requires a random backoff.

Now we consider in detail the backoff time compute. Backoff algorithm is very simple:

```
Initial backoff = MSG_DATA_SIZE+ randint(0,127)
Congestion backoff = MSG_DATA_SIZE * (randint(0,15)+1)
```

where `raindint(x,y)` is a function that computes a random integer between `x` and `y`

Since maximum message data size (`MSG_DATA_SIZE`) is 36 bytes and effective radio rata rate is 19.2 kbps we have :

```
initial backoff = rand(15ms,68.3ms)
congestion backoff = rand(12.08ms,193.3ms)
```

where `rand(x,y)` is a function that computes a random real number between `x` and `y`.

So MAC algorithm for a packet transmission is:

```
Schedule initial backoff = rand(15ms,68.3ms)
If sMacDelay=0
     If !receivedpreamble & RSSI > threshold =>transmit
Else schedule congestion backoff =rand(12.08ms,193.3ms)
```