TinyOS Scheduler, Boot sequence And concurrency model

I. TinyOS Boot sequence

When the system first starts the system goes thru some initialization steps that can be summarized as follows

1. **Scheduler initialization**

   This first step is to initialize TinyOS scheduler. It is used to manage tasks (Discussed later) although not all components need tasks this will insure that everything is working correctly. The basic TinyOS scheduler uses a FIFO algorithm, which can be replaced with other algorithms.

2. **Component initialization**

   The second step is to initialize the platformC component this component has different implementation for different platforms. Duties performed in this stage are to initialize different components (hardware and software) and putting the platform in a known state.

3. **Signal that the boot process has completed**

   At this stage the event booted from the interface Boot is signaled. This allows the programmer to know that all components are initialized and working. All of your code should start from here.

4. **Run the scheduler**

   After knowing that all components are initialized the system enters its core scheduling loop. As long as there are tasks in the queue the scheduler keeps working other wise the system goes to low power mode.

II. TinyOS concurrency model

We have four types of calls in TinyOS.
1. Commands (already discussed).
2. Events or Back calls (already discussed).
3. Tasks.
4. C procedures.

Most calls in TinyOS are non-preemptive meaning that they do not interrupt each other. However there are those cases were code is asynchronous (Preemptive) which can interrupt each others.
As rule of thump most hardware interrupt handlers are preemptive.

Tasks

Tasks are a type of deferred procedure calls (DPC) which means that they will not be executed immediately but will be posted to a queue which will execute them in FIFO order. Tasks are always void and take no parameters since they execute within the scope of a single component. Tasks also are non-preemptive with respect to each other. In other words if a task is started it will continue executing until it finishes without any interruptions from other tasks.
So one might ask what they are good for. Tasks are generally used to execute a procedure that can wait. In other words it must not execute immediately. Consider the following scenario.

```
Event message_t*  Receive.receive (message_t* buff, void* payload, uint8_t
len)
{
    // do something
    // do lengthy calculations
return buff;
}
```

In the above code the execution of the lengthy calculation code means that other received packets could be dropped since the receive event is still busy. Now consider the following scenario.

```
task void lengthy_calc_task();
Event message_t*  Receive.receive (message_t* buff, void* payload, uint8_t
len)
{
    // do something
post lengthy_calc_task();
return buff;
}
task void lengthy_calc_task(){
//do the lengthy calculations
 }
```

In the above example the calculation is deferred for a later time allowing the receive event to return and receive more packets.

How to use tasks?

Using tasks is simple. First you have to declare the task as follows

```
task void someTask(){
//do something }
```

Tasks headers can also be declared as in C functions

```
 task void someTask();
```

What's lift is to post the task. This can be done anywhere using

```
post someTask();
```

C Procedures

Those are procedures that are exactly like the ones in C. they can return types and takes parameters. They also can preempt each other.

The printf library for TinyOS

TinyOS starting from version 2.0.2 offers a printf library that prints to the serial port and can be read thru the PC using the Java printf client.
To use the printf library you have to include the printf.h header file in your code as follows

```
#include “printf.h”
```

Later on you can simply use the printf anywhere in your code with the same syntax as in C for example

```
uint16_t a = 8;
printf(“I’m writing from TOS my variable is %d”, a);
printfflush();
```

when you use printf the bytes you print are placed in a buffer. The data is only sent when you issue a printfflush() call. The default buffer size is 12 bytes and can be changed.

There is one more modification that has to be done for the makefile file to tell the compiler where to find the printf.h file. You should add the following line to the makefile file.

```
CFLAGS += -I$(TOSDIR)/lib/printf
```

For example assume the Blink application. Then the makefile would be like this

```
COMPONENT=BlinkAppC
CFLAGS += -I$(TOSDIR)/lib/printf
include $(MAKERULES)
```

Now to see the output of your program, go to cygwin window and type

```
java net.tinyos.tools.PrintfClient -comm serial@comX:telosb
```

Where X is the com port the mote is connected to.

Exercise 1

Modify the application in exercise 1 from the previous lab to do the sending in a task.

Exercise 2

Modify the second exercise of the previous lab to print the received data thru the printf library.

Exercise 3

Modify the second exercise of the previous lab to send the received message thru the serial port.
Hint look at tutorial 4 http://docs.tinyos.net/index.php/Mote-PC_serial_communication_and_SerialForwarder

For more information see the following links

TEP 106: http://www.tinyos.net/tinyos-2.x/doc/html/tep106.html
TEP 107: http://www.tinyos.net/tinyos-2.x/doc/html/tep107.html
Tutorial 6: http://docs.tinyos.net/index.php/Boot_Sequence
Tutorial 15: http://docs.tinyos.net/index.php/The_TinyOS_printf_Library
TinyOS Programming Sections 4.4 and 4.5