Mobile Software Development
Framework: TinyOS, J2ME

10/2/2012

Y. Richard Yang
Admin.

- Homework 2 posted
- Start to think about project
Recap: OFDM

- Basic idea: use multiple subcarriers to reduce symbol rate per carrier
- Problem: inter-carrier-interference (ICI)
  - Idea: orthogonal subcarriers to avoid ICI: chose each subcarrier frequency so that an integral number of cycles in a symbol period
- Problem: implementation complexity
  - Idea: iFFT as an efficient algorithm for modulation on multiple subcarriers
- Problem: ISI
  - Idea: cyclic prefix (guard interval)
Recap: Summary of Wireless PHY

Transmitter:
- Scramble
- DQPSK Mod
- Direct Sequence
  Spread Spectrum
- Symbol Wave
  Shaping
  To RF

Receiver:
- Decimation
- Despreading
- DQPSK Demod
- Descramble
  To MAC

(a) IEEE 802.11b 2Mbps

Transmitter:
- Scramble
- Convolutional
  encoder
- Interleaving
- QAM Mod
- IFFT
- GI Addition
- Symbol Wave
  Shaping
  To RF

Receiver:
- Decimation
- Remove GI
- FFT
- Demod +
  Interleaving
- Viterbi
  decoding
- Descramble
  To MAC

(b) IEEE 802.11a/g 24Mbps

Functional blocks in their PHY components. These functional blocks are pipelined with one another. Data are streamed through these blocks sequentially, but with different data types and sizes. As illustrated in Figure 1, different blocks may consume or produce different types of data in different rates arranged in small data blocks.

For example, in 802.11b, the scrambler may consume and produce one bit, while DQPSK modulation maps each two-bit data block onto a complex symbol which uses two 16-bit numbers to represent the in-phase and quadrature (I/Q) components.

Each PHY block performs a fixed amount of computation on every transmitted or received bit. When the data rate is high, e.g., 11Mbps for 802.11b and 54Mbps for 802.11a/g, PHY processing blocks consume a significant amount of computational power. Based on the model in [19], we estimate that a direct implementation of 802.11b may require 10Gops while 802.11a/g needs at least 40Gops. These requirements are very demanding for software processing in GPPs.

PHY processing blocks directly operate on the digital waveforms after modulation on the transmitter side and before demodulation on the receiver side. Therefore, high-throughput interfaces are needed to connect these processing blocks as well as to connect the PHY and radio front-end. The required throughput linearly scales with the bandwidth of the baseband signal. For example, the channel bandwidth is 20MHz in 802.11a. It requires a data rate of at least 20M complex samples per second to represent the waveform [14]. These complex samples normally require 16-bit quantization for both I and Q components to provide sufficient fidelity, translating into 32 bits per sample, or 640Mbps for the full 20MHz channel. Over-sampling, a technique widely used for better performance [12], doubles the requirement to 1.28Gbps to move data between the RF front-end and PHY blocks for one 802.11a channel.

2.2 Wireless MAC

The wireless channel is a resource shared by all transceivers operating on the same spectrum. As simultaneously transmitting neighbors may interfere with each other, various MAC protocols have been developed to coordinate their transmissions in wireless networks to avoid collisions.

Most modern MAC protocols, such as 802.11, require timely responses to critical events. For example, 802.11 adopts a CSMA (Carrier-Sense Multiple Access) MAC protocol to coordinate transmissions [7]. Transmitters are required to sense the channel before starting their transmission, and channel access is only allowed when no energy is sensed, i.e., the channel is free. The latency between sense and access should be as small as possible. Otherwise, the sensing result could be outdated and inaccurate. Another example is the link-layer retransmission mechanisms in wireless protocols, which may require an immediate acknowledgement (ACK) to be returned in a limited time window.

Commercial standards like IEEE 802.11 mandate a response latency within tens of microseconds, which is challenging to achieve in software on a general purpose PC with a general purpose OS.

2.3 Software Radio Requirements

Given the above discussion, we summarize the requirements for implementing a software radio system on a general PC platform:

- High system throughput.
- The interfaces between the radio front-end and PHY as well as between some PHY processing blocks must possess sufficiently high...
Recap: GNURadio

- A software development toolkit that provides signal processing blocks to implement software-defined radio systems.

- Some key ideas
  - Hybrid software system (Python/C++)
  - Composite pattern to build a hierarchy of blocks
  - Internal scheduler to orchestrate data flows among blocks
  - Define gr_block as a reusable base so that defining a new block is typically simple: overwrite general_work and forecast
Recap: TinyOS

- A free and open source component based operating system and platform targeting wireless sensor networks (WSNs)

- Some design features
  - Problem: small footprint
    - Idea: TinyOS: Generate customized OS + application: support one app at a time but flexible reprogramming
A TinyOS consists of one or more **components/modules** linked together
- software components motivated by hardware component

Each component specifies:
- it provides some **interfaces**
  - allows other components to control it
- also uses some interfaces
  - control other components
## Interface

- An interface declares:
  - a set of functions called **commands** that the provider must implement
  - another set of functions called **events** that the interface user must implement

![Diagram]

- A uses interfaces I1 and I2
- B provides I1
- C provides I2
- C provides I3
**Interface: Examples**

**StdControl.nc**

```plaintext
interface StdControl {
    command result_t init();
    command result_t start();
    command result_t stop();
}
```

**Timer.nc**

```plaintext
interface Timer {
    command result_t start(
        char type,
        uint32_t interval);
    command result_t stop();
    event result_t fired();
}
```

**ADC.nc**

```plaintext
interface ADC {
    async command result_t getData();
    async command result_t getContinuousData();
    event result_t dataReady(uint16_t data);
}
```
Example Application

- A simple TinyOS application which periodically reads in the light intensity value, computes a moving average, displays it on the LED
module SenseTaskMM {
    provides {
        interface StdControl;
    }
    uses {
        interface Timer;
        interface ADC;
        interface StdControl
            as ADCControl;
        interface Leds;
    }
}

A uses interfaces I1 and I2

I1
commands

I2
events

B provides I1

C provides I2

C provides I3
Module: Implementation

- Define
  - commands and event handlers
  - frame (storage)
    - statically allocated, fixed size to know memory requirement and avoid overhead of dynamic allocation

- See SenseTaskM.nc
Explicit Linking of Components

Two types of components:

- **modules**: individual components

- **configurations**: assemble components together, connecting interfaces (objects) used by components to interfaces (objects) provided by others
  - See SenseTask.nc
TinyOS Execution Model

SenseTask

Timer

ADC

ADCControl

LED

start/stop fired getData dataReady init/start/stop

xxxOn/Off()
TinyOS Execution Model

- Concurrency model: only two threads
  - long running tasks that can be interrupted by hardware event handlers

- Tasks are posted to a FIFO queue
  - Each task is atomic with respect to other tasks
  - run to completion, but can be preempted by events
  - the task scheduler is a simple FIFO scheduler

- Tasks perform the primary computation work
  - commands and event handlers post tasks
  - call lower level commands
  - signal higher level events
  - schedule other tasks within a component
Running tinyOS Program

- make mica
- ncc -o main.exe -target=mica SenseTask.nc
- avr-objcopy --output-target=srec main.exe main.srec
- Use uisp to install
A More Complete Sample Application

- Sensor network monitoring
  - monitor temperature and light conditions
  - periodically transmit measurements to a base station
  - sensors can forward data for other sensors that are out of range of the base station
  - dynamically determine the correct routing topology for the network
Internal Component Graph

Ad hoc Routing Application

Active Messages

Radio Packet

UART Packet

Temp

I2C

Light

RFM

Clocks

application

packet

byte

bit

bit

byte

packet

Application

Radio byte

UART

I2C

Light

Clocks

SW

HW
Message Send Transition

- Total propagation delay up the 5 layer radio communication stack is about 80 instructions

Timing diagram of event propagation
Evaluation: Storage

- Scheduler only occupies 178 bytes

- Complete application only requires 3 KB of instruction memory and 226 bytes of data (less than 50% of the 512 bytes available)

- Only processor_init, TinyOS scheduler, and C runtime are required

<table>
<thead>
<tr>
<th>Component Name</th>
<th>Code Size (bytes)</th>
<th>Data Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Routing</td>
<td>88</td>
<td>0</td>
</tr>
<tr>
<td>AM_dispatch</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>AM_temperature</td>
<td>78</td>
<td>32</td>
</tr>
<tr>
<td>AM_light</td>
<td>146</td>
<td>8</td>
</tr>
<tr>
<td>AM</td>
<td>356</td>
<td>40</td>
</tr>
<tr>
<td>RADIO_packet</td>
<td>334</td>
<td>40</td>
</tr>
<tr>
<td>RADIO_byte</td>
<td>810</td>
<td>8</td>
</tr>
<tr>
<td>RFM</td>
<td>310</td>
<td>1</td>
</tr>
<tr>
<td>Light</td>
<td>84</td>
<td>1</td>
</tr>
<tr>
<td>Temp</td>
<td>64</td>
<td>1</td>
</tr>
<tr>
<td>UART</td>
<td>196</td>
<td>1</td>
</tr>
<tr>
<td>UART_packet</td>
<td>314</td>
<td>40</td>
</tr>
<tr>
<td>I2C</td>
<td>198</td>
<td>8</td>
</tr>
<tr>
<td>Processor_init</td>
<td>172</td>
<td>30</td>
</tr>
<tr>
<td>TinyOS scheduler</td>
<td>178</td>
<td>16</td>
</tr>
<tr>
<td>C runtime</td>
<td>82</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>3450</strong></td>
<td><strong>226</strong></td>
</tr>
</tbody>
</table>
Evaluation: Timing

<table>
<thead>
<tr>
<th>Operations</th>
<th>Cost (cycles)</th>
<th>Time (µs)</th>
<th>Normalized to byte copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte copy</td>
<td>8</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Post an Event</td>
<td>10</td>
<td>2.5</td>
<td>1.25</td>
</tr>
<tr>
<td>Call a Command</td>
<td>46</td>
<td>11.5</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>51</td>
<td>12.75</td>
<td>6</td>
</tr>
<tr>
<td>Post a task to scheduler</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Context switch overhead</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interrupt (hardware cost)</td>
<td>9</td>
<td>2.25</td>
<td>1</td>
</tr>
<tr>
<td>Interrupt (software cost)</td>
<td>71</td>
<td>17.75</td>
<td>9</td>
</tr>
</tbody>
</table>
Components
- provide commands and require callback hooks for event-driven programming

Configurations
- Link components

TinyOS
- an app (configuration) at a time, linking only necessary components

Two threads exec
- one for event
- one for task

ADC.nc

```c
interface ADC {
    async command result_t getdata();
    async command result_t getContinuousData();
    event result_t dataReady(uint 16_t data);
}
```

configuration SenseTask {
    // this module does not provide any interfaces
} implementation {
    components Main, SenseTaskM, LedsC, TimerC, DemoSensorC as Sensor;
    Main.StdControl -> TimerC;
    Main.StdControl -> Sensor;
    Main.StdControl -> SenseTaskM;
    SenseTaskM.Timer -> TimerC.Timer[unique("Timer")];
    SenseTaskM.ADC -> Sensor;
    SenseTaskM.Leds -> LedsC;
}
Discussion: Compare TinyOS/GNURadio

■ What are some similar software concepts?

■ What are some differences?
Discussion

- Can we use GNURadio/TinyOS for writing mobile applications for mobile phones, or in other words, what are missing?
Java2 Micro Edition (J2ME)
Outline

- Admin and recap
- Mobile/wireless development framework
  - GNURadio
  - TinyOS
  - J2ME
Java Platforms

- **Java2 is divided into three platforms**
  - **J2EE (Java2 Enterprise Edition)**
    - business applications
  - **J2SE (Java2 Standard Edition)**
    - general applications
  - **J2ME (Java2 Micro Edition)**
    - small devices such as mobile phone, PDA, car navigation

- **Oracle’s claims on Java on mobile devices**
To accommodate heterogeneous mobile devices, define configurations and profiles

- A configuration provides fundamental services for a broad category of devices (e.g., lang, io, util)

- A profile supports higher-level services common to a more specific class of devices or market (e.g., life cycle, GUI)

- An optional package adds specialized services that are useful on devices of many kinds, but not necessary on all of them

http://developers.sun.com/techtopics/mobility/getstart/articles/survey/
J2ME

Servers &
enterprise
computers

Optional
Packages

Java 2
Platform,
Enterprise
Edition (J2EE)

Java VM

J2ME

Servers &
personal
computers

Optional
Packages

Java 2
Platform,
Standard
Edition (J2SE)

Java VM

High-end PDAs
TV set-top boxes
Embedded devices

Optional
Packages

Personal Profile

Personal Basis Profile

Foundation Profile

MIDP

Mobile phones &
entry-level
PDAs

Optional
Packages

Optional
Packages

Java Card
Card VM

Smart
cards

Java Platform,
Micro Edition (Java ME)

Upto 2M mem
32 bit proc

128-512K mem
16-32 bit proc
Example J2ME Configurations

- **Connected Limited Device Configuration (CLDC)**
  - 160 KB to 512 KB of total memory available
  - 16-bit or 32-bit processor
  - low power consumption and often operating with battery power
  - connectivity with limited bandwidth
  - examples: cell phones, certain PDAs

- **Connected Device Configuration (CDC)**
  - 2 MB or more memory for Java platform
  - 32-bit processor
  - high bandwidth network connection, most often using TCP/IP
  - examples: set-top boxes, certain PDAs
CLDC Available Packages

- java.lang
- java.util
- java.io
- javax.microedition.io
CLDC Classes

- Boolean
- Byte
- Character
- Class
- Integer
- Long
- Math
- Object
- Runnable
- Runtime
- Short
- String
- StringBuffer
- System
- Thread
- Throwable

java.lang

- Calendar
- Date
- Enumeration
- Hashtable
- Random
- Stack
- TimeZone
- Vector

java.util

- ByteArrayOutputStream
- ByteArrayInputStream
- DataOutput
- DataInput
- DataInputStream
- DataOutputStream
- InputStream
- InputStreamReader
- OutputStream
- OutputStreamWriter
- PrintStream
- Reader
- Writer

java.io
Example J2ME Profiles

- **Mobile Information Device Profile (MIDP)**
  - GUI, multimedia and game functionality, end-to-end security, and greater networked connectivity
  - mobile phones and entry level PDAs

- **Foundation Profile**
  - set of Java APIs that support resource-constrained devices without a standards-based GUI system

- **Personal Profile**
  - Full set of AWT APIs, including support for applets and Xlets
  - CDC + Foundation Profile + Personal Profile for high-end PDA

- ...
Mobile Phone Framework

MIDP 2.0

MIDlet Suites, OEM Applications

WMA (JSR-120)

MMAP (JSR-135) Conditionally included

Optional Packages

Vendor-specific Classes (OEM)

Native AMS/OTA

Native Browser

Native (Phone) Apps

Native OEM Apps

CLDC 1.0 or 1.1

Native Operating System

Wireless Device
MIDP Hardware

- Memory (added to CLDC memory)
  - 128 KB non-volatile for MIDP components
  - 8 KB non-volatile for application persistent data
  - 32 KB volatile for KVM

- Display
  - Screen 96x54
  - Display depth 1-bit
  - Pixel shape (aspect ratio) 1:1
MIDP Hardware

- **Input (one or more)**
  - one-handed keyboard (ITU-T phone keypad)
  - two-handed keyboard (QWERTY keyboard)
  - or touch screen

- **Networking**
  - two-way
  - wireless
  - possibly intermittent
  - limited bandwidth
MIDP Packages

- java.io
- java.lang
- java.util
- javax.microedition.io
- javax.microedition.lcdui
- javax.microedition.rms
- javax.microedition.midlet
- javax.microedition.lcdui.game
- javax.microedition.media
- javax.microedition.media.control
- javax.microedition.pki

version 1.0

addition in version 2.0
MIDP Technology Stack

CLDC = KVM + J2ME Core APIs in this example

KVM

DSP chip (e.g., ARM)

J2ME core APIs

Mobile Information Device Profile

Your MIDlet

Yellow Pages, train schedules and ticketing, games...

UI, HTTP networking...

Threads, no Floats...

32-bit RISC, 256K ROM, 256K Flash, 64K RAM
MIDlet

- GUI based

- Each MIDP has one instance of Display
  - `Display.getDisplay(this)` to get the manager
  - At any instance of time at most one Displayable object can be shown on the display device and interact with user
    - `display.setCurrent(<Displayable object>)`
An MIDP application is called a MIDlet
- similar to the J2SE applet

A MIDlet moves from state to state in the lifecycle, as indicated
- start - acquire resources and start executing
- pause - release resources and become quiescent (wait)
- destroy - release all resources, destroy threads, and end all activity
Each MIDP has one instance of Display

- `Display.getDisplay(this)` to get the manager

- At any instance of time at most one `Displayable` object can be shown on the display device and interact with user
  - `display.setCurrent(<Displayable object>)`
MIDP Visual Display

- Displayable
  - Canvas
    - GameCanvas
  - Screen
    - Alert, List, TextBox, Form

- Form can contain multiple form items for organization
  - Labels, Image Items, String Items, Text Fields, Date Fields, Gauges, Choice Groups
Displayable objects can declare commands and declare a command listener:

- `addCommand(Command cmd)`
- `addCommandListener()`

Command (<label>, <type>, <priority>)

- **Type:** BACK, CANCEL, EXIT, HELP, ITEM, OK, SCREEN, and STOP
```java
public class HelloWorldMIDlet extends MIDlet implements CommandListener {

    private Command exitCommand;
    private Display display;
    private TextBox t;

    public HelloWorldMIDlet() {
        display = Display.getDisplay(this);
        exitCommand = new Command("Exit", Command.EXIT, 2);
        t = new TextBox("CS434", "Hello World!", 256, 0);
        t.addCommand(exitCommand);
        t.setCommandListener(this);
    }

    public void startApp() { display.setCurrent(t); }
    public void pauseApp() { }
    public void destroyApp(boolean unconditional) { }
    public void commandAction(Command c, Displayable s) {
        if (c == exitCommand) {
            destroyApp(false);
            notifyDestroyed();
        }
    }
}
```
MIDP: Persistent State

- Record store defined in javax.microedition.rms

- Record store identified by name:
  - static String[] listRecordStores();
  
  recordStore = RecordStore.openRecordStore("scores", true);
  
  recordId = addRecord(byte[] data, int offset, int numBytes);
  
  getRecord(int recordId);
Summary: J2ME

- Scale down a popular programming environment to ease learning
- Use virtual machines to mask device heterogeneity
- Use configuration/profiling to handle device heterogeneity and avoid using lowest common denominator
- MIDLet to manage app life cycle
- Displayable to visual display, commands and provides command listener
- Introduce persistent record store
Discussion on J2ME

- What designs of J2ME do you like and hence expect newer frameworks (e.g., IOS, Android) may have too?

- What features do you think are missing in J2ME?