Sensor Network Application Development
ZIGBEE CONCEPTS 1

Cruise Summerschool
Johannes Kepler University
November 5 - 7, 2007, Linz / Austria
Dipl.-Ing. Andreas Riener
riener@pervasive.jku.at
Overview

Structure of this lesson

ZigBee-Application development

- Extended Informations about Interfaces, Configurations, Modules, etc.
- Tasks, Interrupts and Synchronization
- Single-Hop communication
- Time synchronisation
### nesC: Interfaces and Configurations

**interface: Timer.nc**

```c
interface Timer {
    // Start the timer.
    command result_t start(char type, uint32_t interval);

    // Stop the timer.
    command result_t stop();

    // The signal generated by
    // the timer when it fires.
    event result_t fired();
}
```

**configuration: TimerC.nc**

```c
configuration TimerC {
    provides interface Timer[uint8_t id];
    provides interface StdControl;
}

implementation {
    components TimerM, ClockC, NoLeds, HPLPowerManagementM;

    TimerM.Leds -> NoLeds.Leds;
    TimerM.Clock -> ClockC;
    TimerM.PowerManagement ->
        HPLPowerManagementM;
    StdControl = TimerM;
    Timer = TimerM.Timer;
}
```

- **Component TimerM uses** the interface Leds provided by the component NoLeds.
  Abbreviation on distinct names: TimerM.Leds -> NoLeds;

- **TimerC provides interface Timer**: has to implement commands, has to signal events.

- **Connect the provided interface Timer** to the Timer interface of Component TimerM.
  Abbreviation on distinct names: Timer = TimerM;
nesC: Modules

```plaintext
module: TimerM.nc

provides interface Timer[uint8_t id];
provides interface StdControl;
uses {
  interface Leds;
  interface Clock;
  interface PowerManagement;
}

implementation {
  //...
  command result_t Timer.start[uint8_t id](char type, uint32_t interval) {
    //do something ...
  }

  task void signalOneTimer() {
    // some code ...
    signal Timer.fired[itimer]();
  }

  default event result_t Timer.fired[uint8_t id]() {
    return SUCCESS;
  }
}

Provided interfaces:
- implement commands
- signal events

Used interfaces:
- implement events
- call commands

Implement a command.

Signal an event.

Provided interface Timer: Optionally, a default event that does nothing can be implemented.
```
### nesC: Application

**configuration: Sense.nc**

```plaintext
configuration Sense {
}
implementation {
    components Main, SenseM, LedsC, TimerC, Photo as Sensor;
    Main.StdControl -> Sensor;
    Main.StdControl -> TimerC;
    Main.StdControl -> SenseM;
    SenseM.ADC -> Sensor;
    SenseM.ADCControl -> Sensor;
    SenseM.Leds -> LedsC;
    SenseM.Timer -> TimerC.Timer[unique("Timer")];
}
```

**module: SenseM.nc**

```plaintext
module SenseM {
    provides {
        interface StdControl;
    }
    uses {
        interface Timer;
        interface ADC;
        interface StdControl as ADCControl;
        interface Leds;
    }
}
implementation {
    // implement StdControl commands ...
    result_t StdControl.start() {
        return call Timer.start(
            TIMER_REPEAT, 500);
    } 
    result_t StdControl.stop() {
        return call Timer.stop();
    } 
    // Timer fired event
    event result_t Timer.fired() {
        return call ADC.getData();
    } 
    // handle ADC data ready events ...
}
```

**Implements events of used interfaces (e.g. Timer.fired()).**

**The module can call commands of used interfaces. (e.g. Timer.start() and Timer.stop()).**

**Module SenseM uses interface Timer of component TimerC (it provides multiple Timers, so an index has to be specified; unique(string) returns a unique id number).**

**SenseM uses the Timer interface of some component specified in the configuration (here TimerC).**
nesC: Interface, Configuration, Module Chain

conf.: TimerC.nc
configuration TimerC {
  provides interface
  Timer[uint8_t id];
} implementation {
  components TimerM, ...
  Timer = TimerM;
}

interface: Timer.nc
interface Timer {
  command ...
  event ...
}

conf.: Sense.nc
configuration Sense {} implementation {
  components SenseM, TimerC, ...
  SenseM.Timer ->
  TimerC.Timer[
    unique("Timer")];
}

module: TimerM.nc
module TimerM {
  provides Timer, ...
  uses ...
} implementation {
...
}

module: SenseM.nc
module SenseM {
  provides ...
  uses interface Timer;
} implementation {
...
}
nesC: Interface and Component Renaming (as)

**module: exampleM.nc**

```nesC
module exampleM {
    provides interface StdControl;
    uses {
        interface Leds;
        interface StdControl as CommControl;
        interface StdControl as SensorControl;
        interface SendMsg as Send;
    }
}
```

Default declaration of interface.

If you use a single interface to control different components, you have to give them custom names.

You can also use custom names to clarify or simplify your code.

**configuration: example.nc**

```nesC
configuration example ()
implementation {
    components Main, exampleM as app, LedsC, Photo as Sensor, GenericComm as Comm;
    Main.StdControl -> app;
    app.Leds -> LedsC;
    app.CommControl -> Comm;
    app.SensorControl -> Sensor;
    app.Send -> Comm.SendMsg[AM_AMSG];
}
```

Components can be renamed too.

This is equal to:

```
Main.StdControl -> app.StdControl;
```

The target interface name can be omitted if it is non-ambiguous.

This expression is valid: CommControl is actually a StdControl interface, and GenericComm provides exactly one StdControl.
ZigBee-Concepts

Tasks, Interrupts and Synchronization
**nesC: Program Flow**

- **Task is started**
  e.g. by a TimerEvent

- **Task is running**
  This may take some time

- **Another event occurs and interrupts the current task**
  e.g. an incoming RF packet

- **Event handling**
  To avoid long interruptions, store the event data and enqueue a task to handle it later

- **Proceed with last task**

- **Check for waiting tasks and execute them in FIFO order**
nesC: Tasks and Interrupts

Task queue (FIFO)
- Tasks do not interrupt each other
- No synchronization between tasks necessary

Interrupts
- Some events may occur at any time (e.g. dataReady events from Sensors)
- Marked with the 'async' keyword
- Interrupt currently running tasks (and may even interrupt other 'async' events)
- Have to lock any data they access and share with other events or tasks
- If a command is called from or an event is signaled from an 'async'-section, it is also 'async'
- To lock variables (exclusive access), use the 'atomic' keyword
- The compiler warns you if you try to access variables shared with 'async' sections and do not lock them
- To avoid compiler warnings in case you are sure there is no danger involved, use the 'norace' keyword
### nesC: Tasks and Synchronization

```nesC
module: TimerM.nc

task void signalOneTimer() {
    uint8_t itimer = dequeue();
    if (itimer < NUM_TIMERS)
        signal Timer.fired[itimer]();
}

task void HandleFire() {
    uint16_t int_out;
    atomic {
        int_out = interval_outstanding;
        interval_outstanding = 0;
    }
    // ...
    post signalOneTimer();
    // ...
    enqueue(i);
    // ...
}

async event result_t Clock.fire() {
    atomic {
        if (interval_outstanding == 0)
            post HandleFire();
    }
    return SUCCESS;
}
```

- Tasks do not interrupt each other and usually do not need any synchronization. This also applies to any signaled events or called commands.
- If variables are shared with an 'async' section, synchronization is necessary! Use 'atomic'.
- `interval_outstanding` is locked. Other tasks/events may not access it until this 'atomic' section ends.
- The keyword 'post' is used to enqueue a task into the task-queue.
- 'async' event: May interrupt currently running tasks.
- `interval_outstanding` is locked.
- Event handling is done by its own task.
nesC: Synchronization with 'norace'

module: exampleM.nc

implementation {
  bool bufferInUse;
  norace uint8_t buffer[32];

  result_t doSomething() {
    bool localBIU;
    atomic {
      localBIU = bufferInUse;
      bufferInUse = true;
    }
    if (localBIU) return FAIL;
    // do something with buffer
    atomic bufferInUse = false;
    return SUCCESS;
  }

  async event result_t Clock.fire() {
    bool localBIU;
    atomic {
      localBIU = bufferInUse;
      bufferInUse = true;
    }
    if (localBIU) return FAIL;
    // do something with buffer
    atomic bufferInUse = false;
    return SUCCESS;
  }
}

Usually all operations on the buffer would have to be in an 'atomic' area, but we secured the buffer with the flag-variable bufferInUse. In this case a race condition is impossible. To avoid irrelevant compiler messages, we declare the buffer as 'norace'.

But: 'bufferInUse' still needs to be locked!
ZigBee-Concepts

Sensors and Analog-Digital-Converters (ADC)
XBow Sensor Board MTS300/310

Sensors

- Dual-Axis Accelerometer (ADXL202)
- Dual-Axis Magnetometer
- Light
- Temperature
- Acoustic
- Sounder (4-KHz Buzzer)

All sensor data is read via the ADC (Analog-Digital-Converter) interface:

```cpp
interface ADC {
    // Initiates an ADC conversion on a given port.
    async command result_t getData();

    // Initiates a series of ADC conversions. Each return from
    // <code>dataReady()</code> initiates the next conversion.
    async command result_t getContinuousData();

    // Indicates a sample has been recorded by the ADC as the result
    // of a <code>getData()</code> command.
    async event result_t dataReady(uint16_t data);
}
```
**XBow Sensor Example**

**makefile**

```makefile
COMPONENT=example
PLATFORMS=pc micaz

XBOWROOT=$(TOSROOT)/contrib/xbow/tos
PFLAGS= -I$(XBOWROOT)/platform/micaz -I$(XBOWROOT)/sensorboards/mts310

include $(TOSROOT)/tools/make/Makerules
```

**application configuration: example.nc**

```makefile
configuration example {}
implementation {
    components Main, exampleM, Photo as Sensor;
    Main.StdControl -> Sensor;
    Main.StdControl -> exampleM;
    exampleM.ADC -> Sensor;
}
```

For more details on the provided interfaces, take a look on the components in `<tinyos-dir>/contrib/xbow/tos/sensorboards/mts310`

MTS310 Sensor Components: Accel (Accelerator), Mag (Magnetometer), MicC (Microphone), Photo (Light), Temp (Temperature)
XBow Sensor Example

```
module exampleM {
  provides interface StdControl;
  uses interface ADC;
}
implementation {
  //Initialize the component
  command result_t StdControl.init() {
    return SUCCESS;
  }
  //Start the component
  command result_t StdControl.start() {
    return call ADC.getData();
  }
  //Stop the component
  command result_t StdControl.stop() {
    return SUCCESS;
  }
  // ADC data ready event handler
  async event result_t ADC.dataReady(uint16_t data) {
    if (data > 0x200) dbg(DBG_USR1, "High signal");
    return SUCCESS;
  }
}
```

Use an Analog-Digital-Converter Interface

Request a data sample from the connected sensor

Data is ready: 10-bit data sample. Careful: dataReady event is asynchronous!
ZigBee-Concepts

Single-Hop Communication
Single-Hop Communication

**GenericComm**: Single component for both RF (zigbee) and UART (serial port) transmissions

- **Input interface ReceiveMsg**

```c
interface: ReceiveMsg.nc

includes AM;
interface ReceiveMsg {
    event TOS_MsgPtr receive(TOS_MsgPtr m);
}
```

- **Output interface SendMsg**

```c
interface: SendMsg.nc

includes AM;
interface SendMsg {
    command result_t send(uint16_t address, uint8_t length, TOS_MsgPtr msg);
    event result_t sendDone(TOS_MsgPtr msg, result_t success);
}
```
Special target addresses and basic message header in AM.h

```c
#header: AM.h

enum {
    TOS_BCAST_ADDR  = 0xffff,
    TOS_UART_ADDR   = 0x007e,
};

typedef struct TOS_Msg {
    // The following fields are
    // transmitted/received on the radio.
    uint16_t addr;
    uint8_t type;
    uint8_t group;
    uint8_t length;
    int8_t data[TOSH_DATA_LENGTH];
    uint16_t crc;
    // ...
} TOS_Msg;

typedef TOS_Msg *TOS_MsgPtr;
// ...

Broadcast a message (RF only):
SendMsg.Send(TOS_BCAST_ADDR, length, msg);

Send to serial port:
SendMsg.Send(TOS_UART_ADDR, length, msg);

Send to node 2 (RF):
SendMsg.Send(2, length, msg);

The local node address is:
TOS_LOCAL_ADDRESS
```
**Single-Hop Communication: Example**

Define message type

```c
typedef struct ExampleMsg {
    uint16_t value;
} ExampleMsg;

groupid: AM_EXAMPLEMSG = 1
```

Application configuration

```c
#include ExampleMsg.h

configuration: example.nc

includes ExampleMsg;
configuration example {
implementation {
    components Main, exampleM, GenericComm as Comm;

    Main.StdControl -> Comm;
    Main.StdControl -> exampleM;

    exampleM.SendMsg -> Comm.SendMsg[AM_EXAMPLEMSG];
    exampleM.ReceiveMsg -> Comm.ReceiveMsg[AM_EXAMPLEMSG];
}
```

The component can now send and receive messages of group AM_EXAMPLEMSG.

Data part of our message

Message group id
Single-Hop Communication: Example

Application module implementation

```
module: exampleM.nc

includes ExampleMsg;
module exampleM {
  provides interface StdControl;
  uses {
    interface SendMsg;
    interface ReceiveMsg;
  }
}
implementation {
  TOS_Msg nextMsg; // the application has to provide the send-buffer
  task void sendData() {
    ExampleMsg *message = (ExampleMsg *)nextMsg.data;
    message->value = 0;
    call SendMsg.send(TOS_BCAST_ADDR, sizeof(ExampleMsg), &nextMsg);
  }
  event result_t SendMsg.sendDone(TOS_MsgPtr msg, result_t success) {
    return SUCCESS;
  }
  event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr m) {
    ExampleMsg *message = (ExampleMsg *)m->data;
    dbg(DBG_USR1, "%i received\n", (int)message->value);
    return m;
  }
}
```

Both interfaces are bound to messages of group AM_EXAMPLEMSG

We send only messages of type ExampleMsg in this group, so we can safely cast incoming messages to this type.
ZigBee-Concepts

Time Synchronization (with Reference Broadcast)
Time Synchronization in a Wireless Network

i) A synchronisation- (sync-) signal is broadcasted by a sender

ii) The receiving nodes compare their timestamps of the event

**Advantage**
An already existing broadcast-signal can be used as the reference

**Disadvantage**
All nodes have to be in the same broadcast domain
Time Synchronization in a Wireless Network

Error in Reference Broadcast Synchronization
\[ e = (R_A - R_B) + (P_{SA} - P_{SB}) \]

R … Time to receive a packet
P … Propagation time between 2 nodes
S,A,B … S (Sender), A and B (Nodes)

The receive time is usually very low and the propagation time is close to zero
(propagation of RF signals is near light speed)

The time to prepare and send a packet does not add to the error!
- This is a great improvement in wireless networks, where the MAC scheme may prevent immediate access to the network
Sensor Network Application Development
ZIGBEE CONCEPTS 1

Cruise Summerschool
Johannes Kepler University
November 5 - 7, 2007, Linz / Austria
Dipl.-Ing. Andreas Riener