Pratique de TinyOS

T.Hérault <herault@lri.fr>

Master 2 Recherche Réseaux et Télécommunications

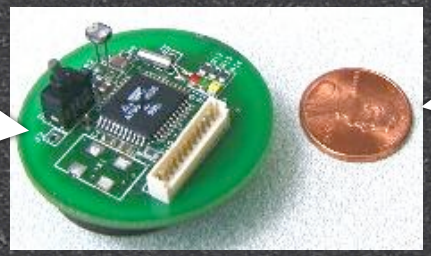
Introduction

Caractéristiques des Réseaux de Senseurs

- Petite taille et faible consommation électrique
- Fonctionnement à fort parallélisme
- Parallélisme matériel et hiérarchie de contrôle limités
- Diversité de l’utilisation et de la conception
- Robustesse des opérations et de la conception d’applications

Petite taille et faible consommation

MCU Berkeley (version XP.)



Pièce de 1 cents

La taille contraint (ou finira par contraindre)
- la capacité de calcul
- la quantité de mémoire
- les fonctionnalités de réseaux

Le système d'exploitation se doit
- d'économiser la consommation énergétique
- de tenir dans le minimum d'espace
- de tirer le meilleur parti possible du matériel

La loi de Moore n'est pas une porte de sortie à ces contraintes.

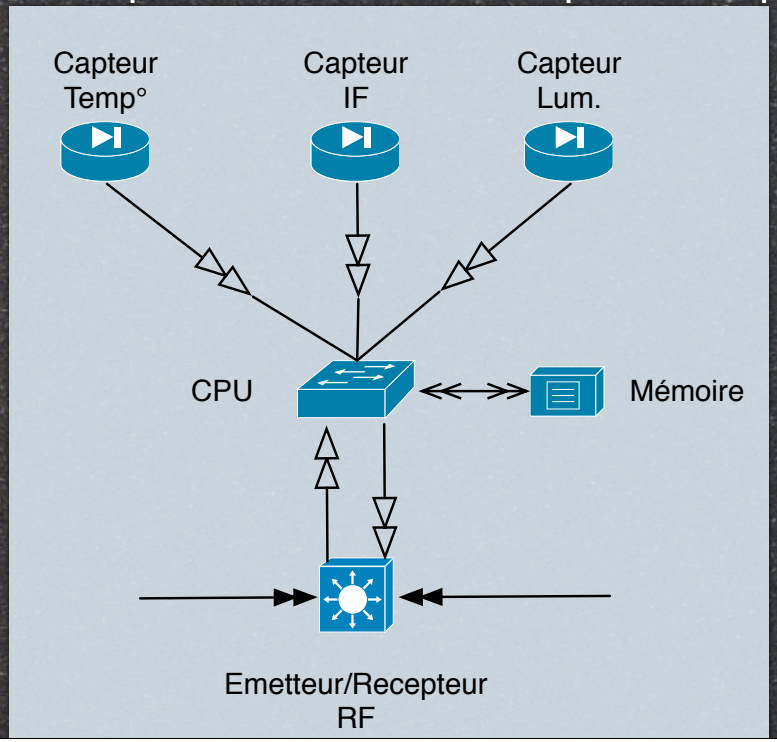
Fonctionnement à fort parallélisme

- ! Parallélisme des capteurs, certes, mais ce n'est pas celui qui nous intéresse ici !
- Parallélisme des opérations menées par chaque capteur

Pas (ou peu) de buffers

Un unique CPU généraliste

Un réseaux très actif



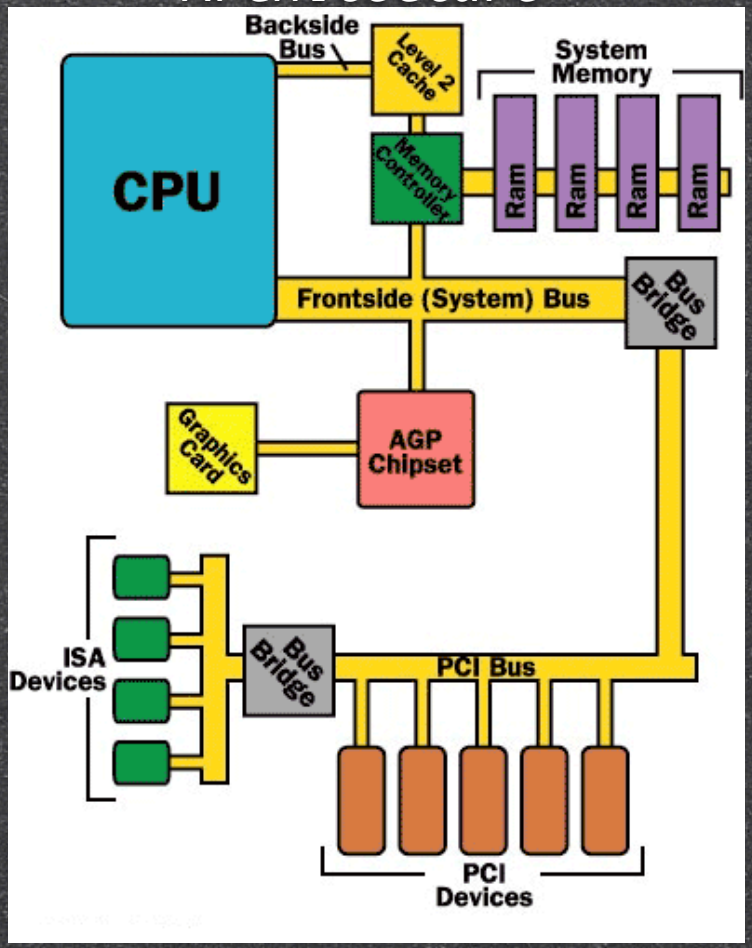
Tâches complexes (p.e. calcul de résultat synthétique)

Tâches temps-réel

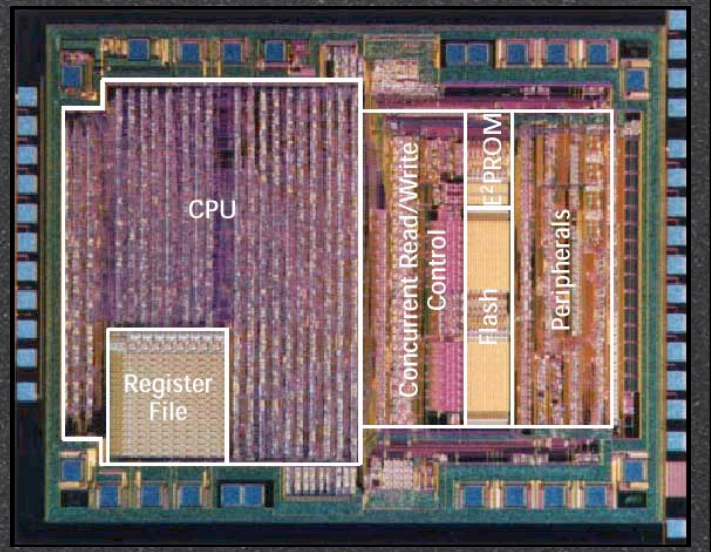
Interruptions fréquentes

Parallélisme matériel et hiérarchie de contrôle limités

Typical Pentium-Based Architecture

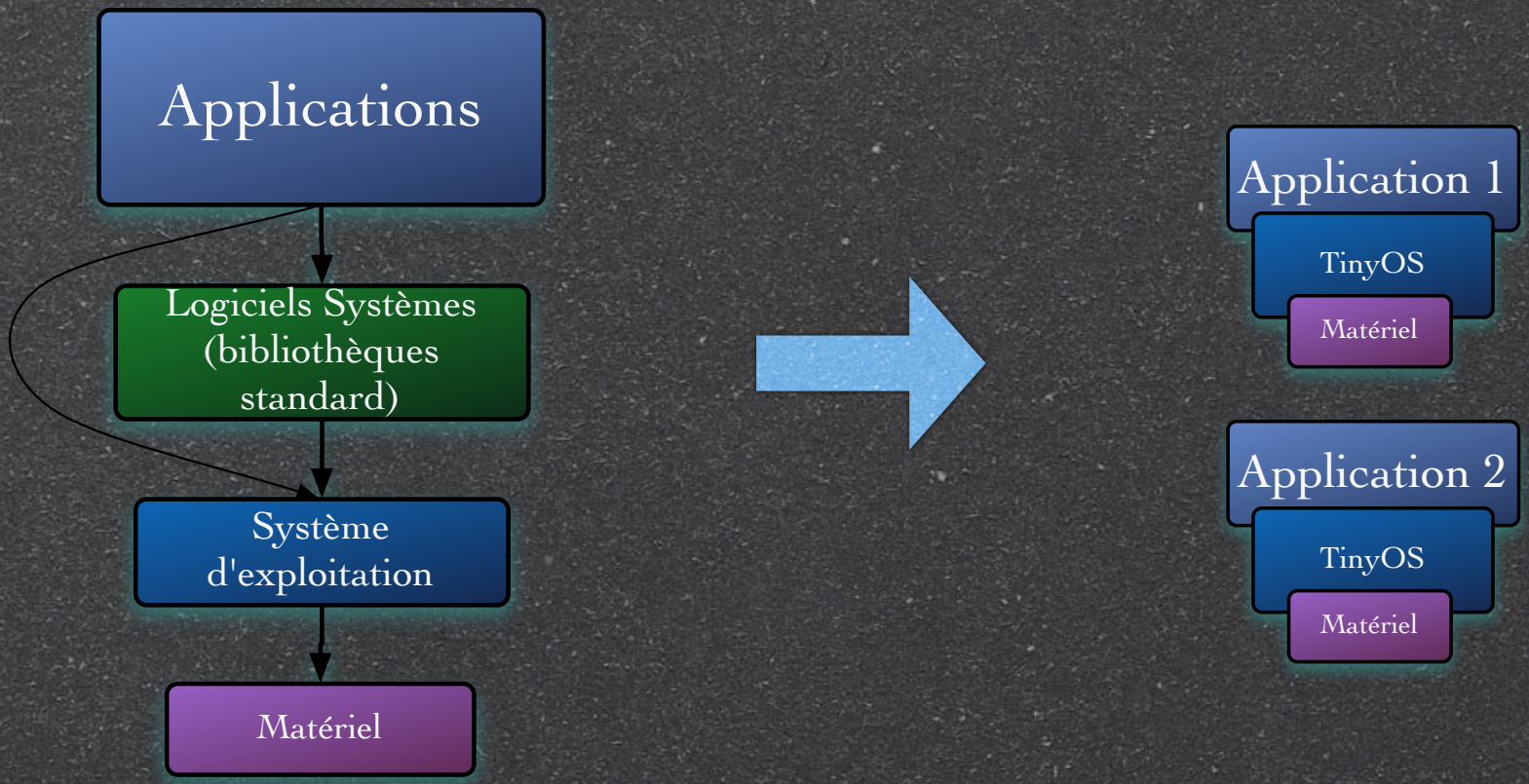


AT90S1200 (AVR) CPU

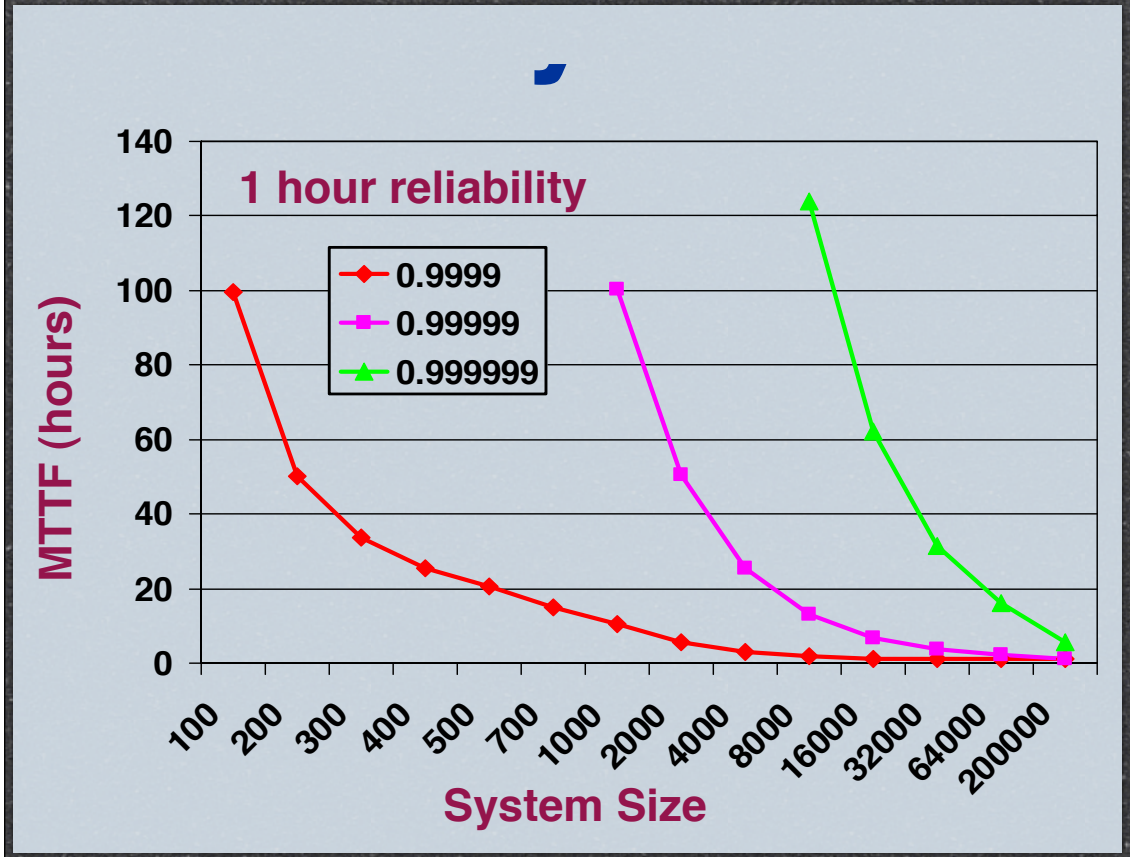


http://www.kreatives-chaos.com/images/127.jpg

Diversité de l'utilisation et de la conception



Robustesse des opérations et de la conception



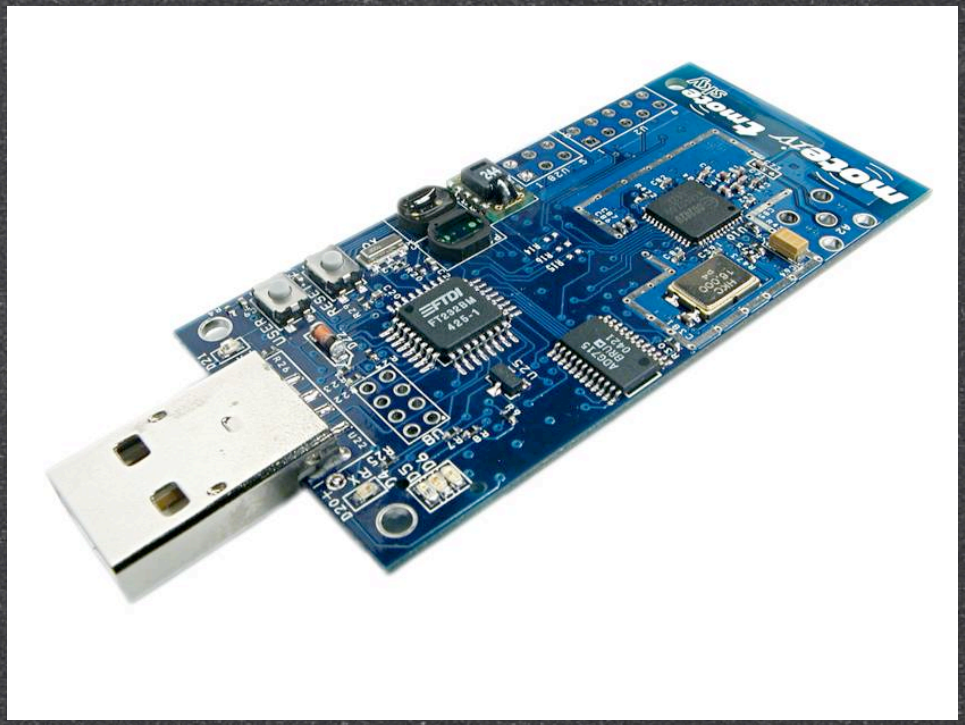
- Augmenter la fiabilité du système via des protocoles tolérants

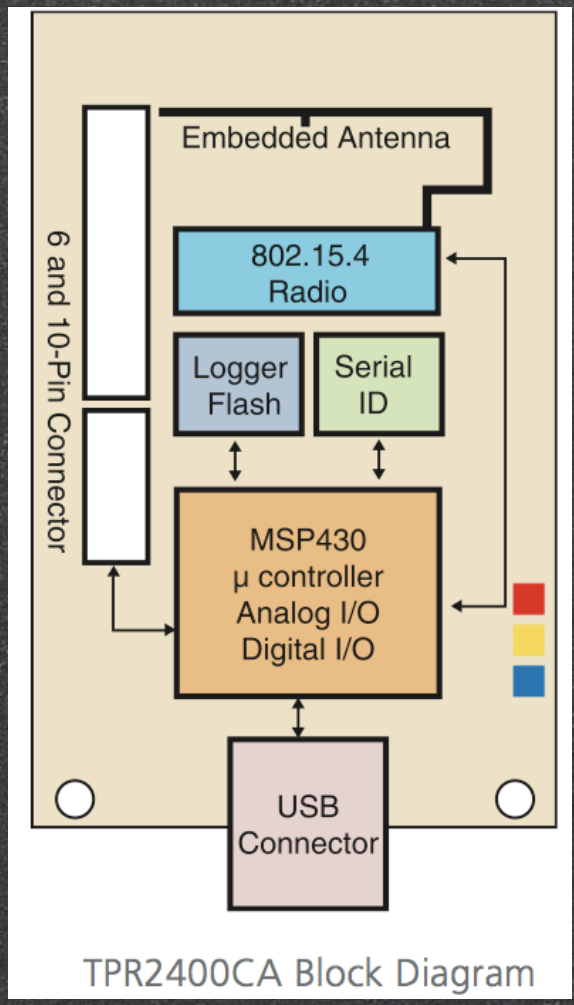
- Développer de nouvelles techniques plus adaptées que la réplication

- Augmenter la fiabilité des Capteurs ET des logiciels qui s'exécutent dessus.

Hardware

Telos B : matériel





TPR2400CA Block Diagram

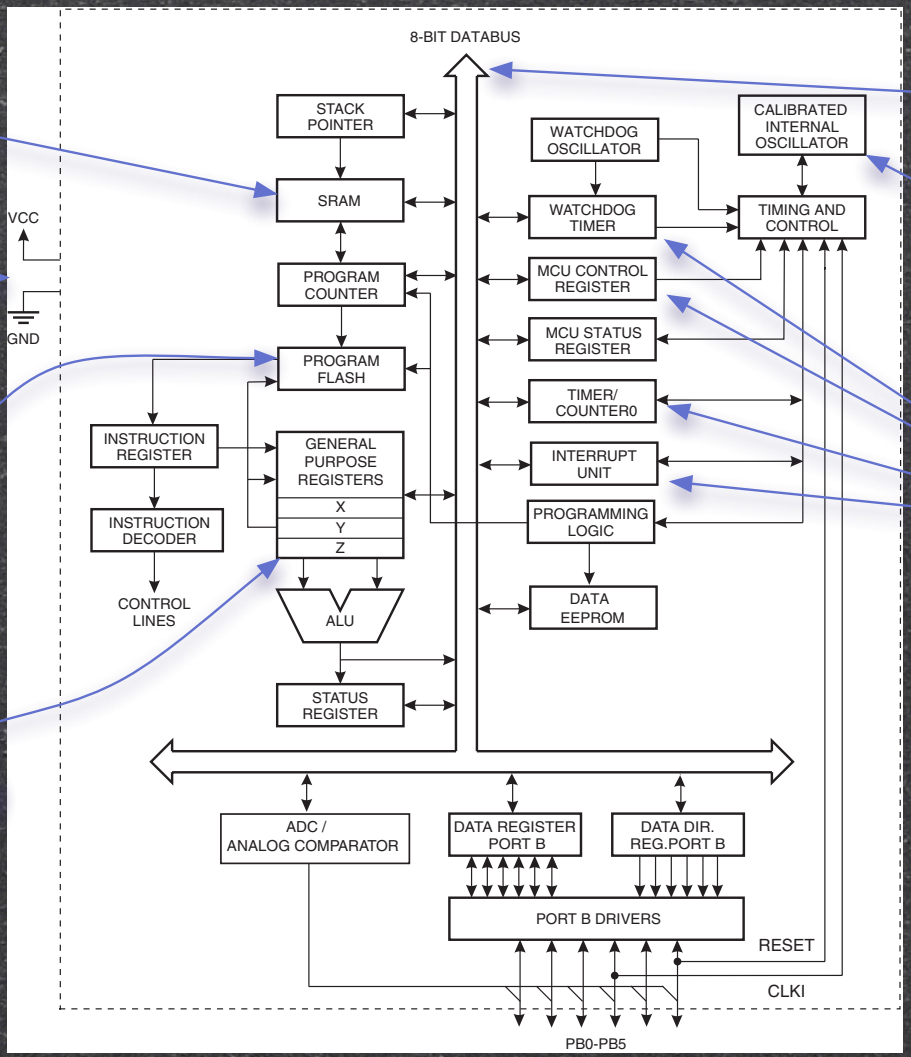
Processeur : ex. ATMEL ATtiny13

512 Octets SRAM (données)

3V

8KB flash (code) non modifiable (co-processeur pour effectuer cela)

32 registres généraux



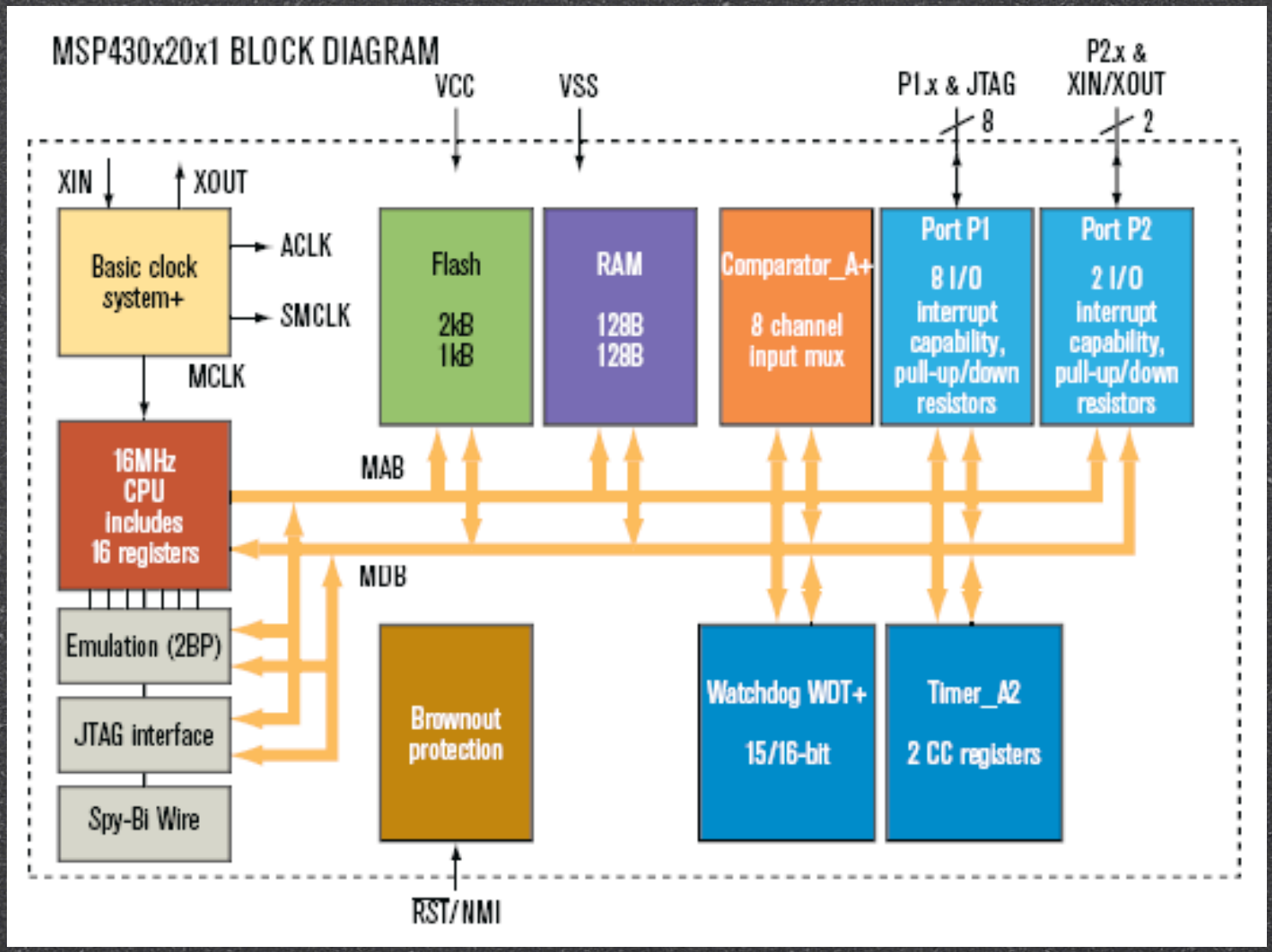
Architecture 8 bits (@ sur 16 bits)

4MHz

compteurs et chronomètres pour créer des interruptions

3 modes de sommeil:
- Idle (processeur off)
- power down (tout off sauf watchdog)
- power save (= power down + chronomètre asynchrone)

Processeur : MSP 430



CC420 RF Radio

- matériel Entrées/Sorties Asynchrone avec contraintes de temps réel.
- ON-OFF key jusqu’à 19.2Kbps
- Transmition XOR Réception XOR power-off.
- Pas de buffer disponible!
- Les valeurs transmises ne sont pas normalisées, et les effets de perturbation eléctromagnétiques sont transmis dans le signal.

Capteurs et port série

- Capteurs analogiques reliés à un numériseur
- Capteurs I2C
- UART série avec mode de transmission par octet
- Coprocesseur pour programmer le code du processeur, et pour stocker de la donnée supplémentaire.

Caractéristiques de consommation énergétique

Component	Active (mA)	Idle (mA)	Inactive (µA)
MCU core	5	2	1
MCU pins	1.5	-	-
LED	4.6 each	-	-
Photocell	0.3	-	-
Radio TX	12	-	5
Radio RX	4.5	-	5
Temp°	1	0.6	1.5
Co-proc	1	0.6	1
EEPROM	3	-	1

575mAh ->

Peak load = 30h

Idle = 200h

Inactive = > 1 year

Exemple de consommation énergétique

- La radio doit émettre toutes les 52µs
  - -> ~ 1µJ pour émettre 1 bit
  - -> ~ 0.5µJ pour recevoir 1 bit.
- Pendant ce temps, le CPU peut exécuter 208 cycles (~100 inst.)
  - -> Consommer jusqu'à 0.8µJ
  - -> dont une partie est dévolue au traitement de la comm.
- Le temps inutilisé est passé en Idle ou power-off
  - -> entre 8.8µJ et 14.4µJ pour émettre un octet
  - -> entre 4.8µJ et 10.4µJ pour recevoir un octet

Software: TinyOS

Microthreading Operating System

- Environment Requirements
  - Concurrency-intensive operation
  - efficient modularity and robustness
- Material
  - Small physical size
  - modest active power load
  - tiny inactive load

Similar to building efficient network interfaces :
- large number of concurrent flows
- juggle numerous outstanding events

-> Extremely efficient multithreading engine

The problem of scheduling multiple threads seen from the memory point of view

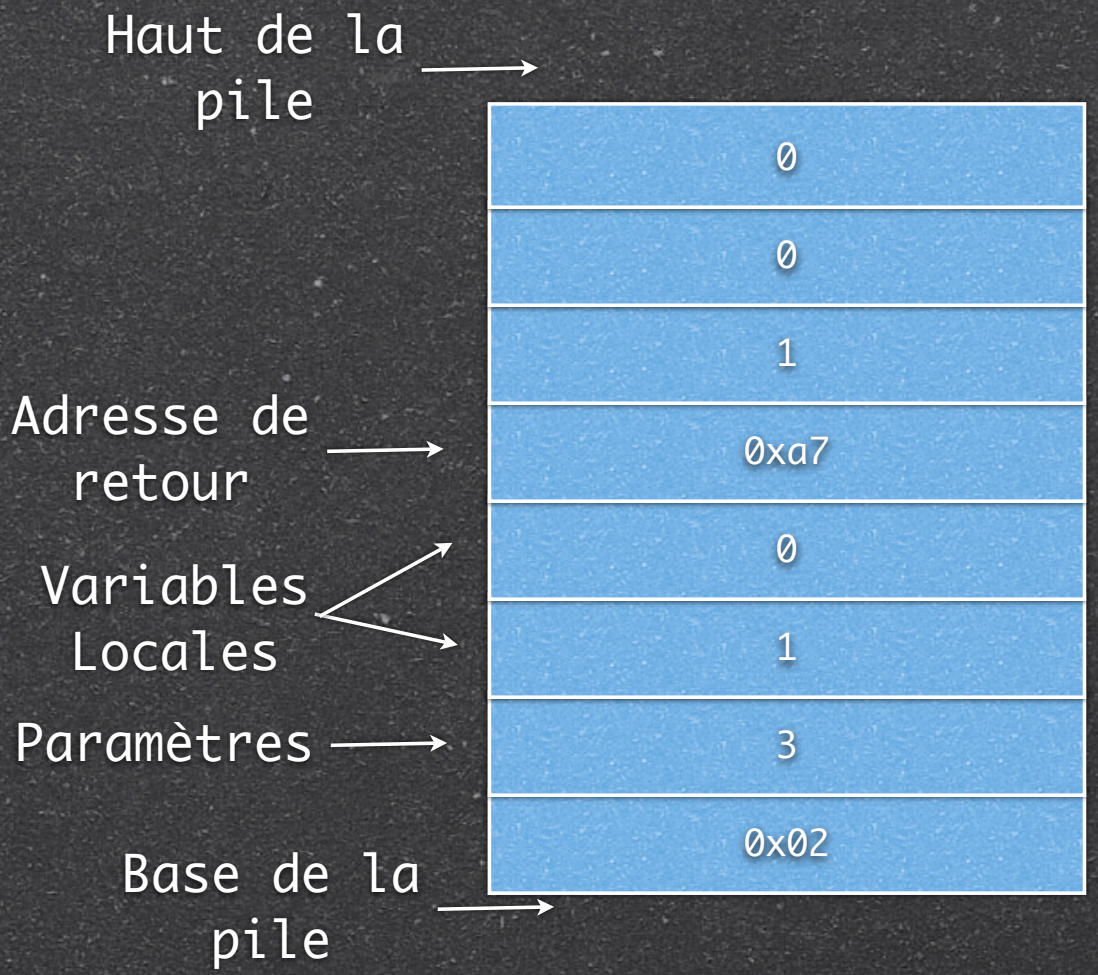
What needs to be saved when scheduling (classical) multiple threads?

- The program counter
- The value of the CPU registers
- The memory map in case of VM
- The Stack!

Each Thread Needs its own STACK

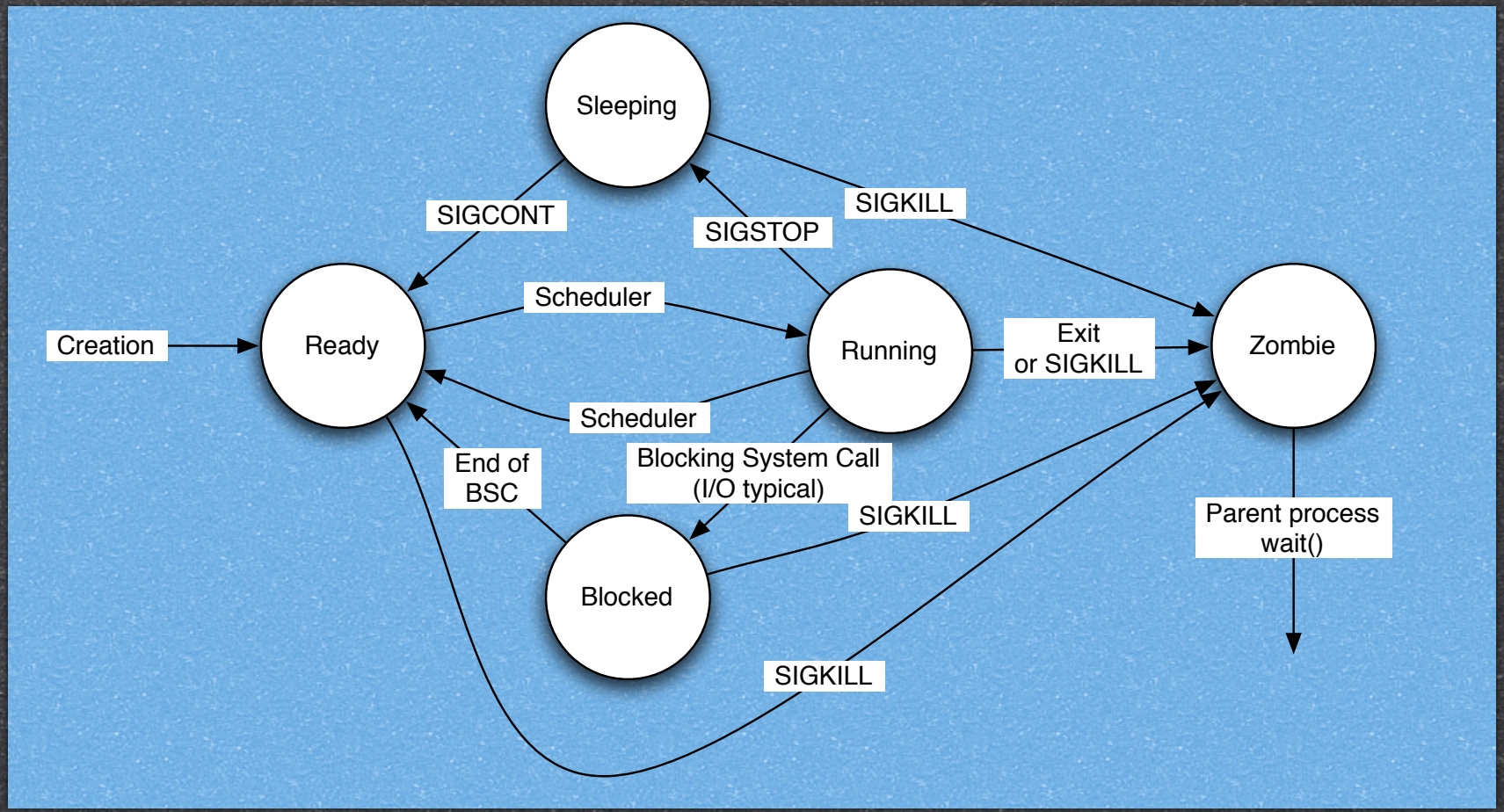
Stack usage for function calls

```
      int f(int a)
      {
       int l, r;
0xa1:  if(a == 1)
0xa2:   return 1;
0xa3:  if(a == 2)
0xa4:   return 1;
0xa5:  l = f(a-1);
0xa6:  r = f(a-2);
0xa7:  return l+r;
      }
```

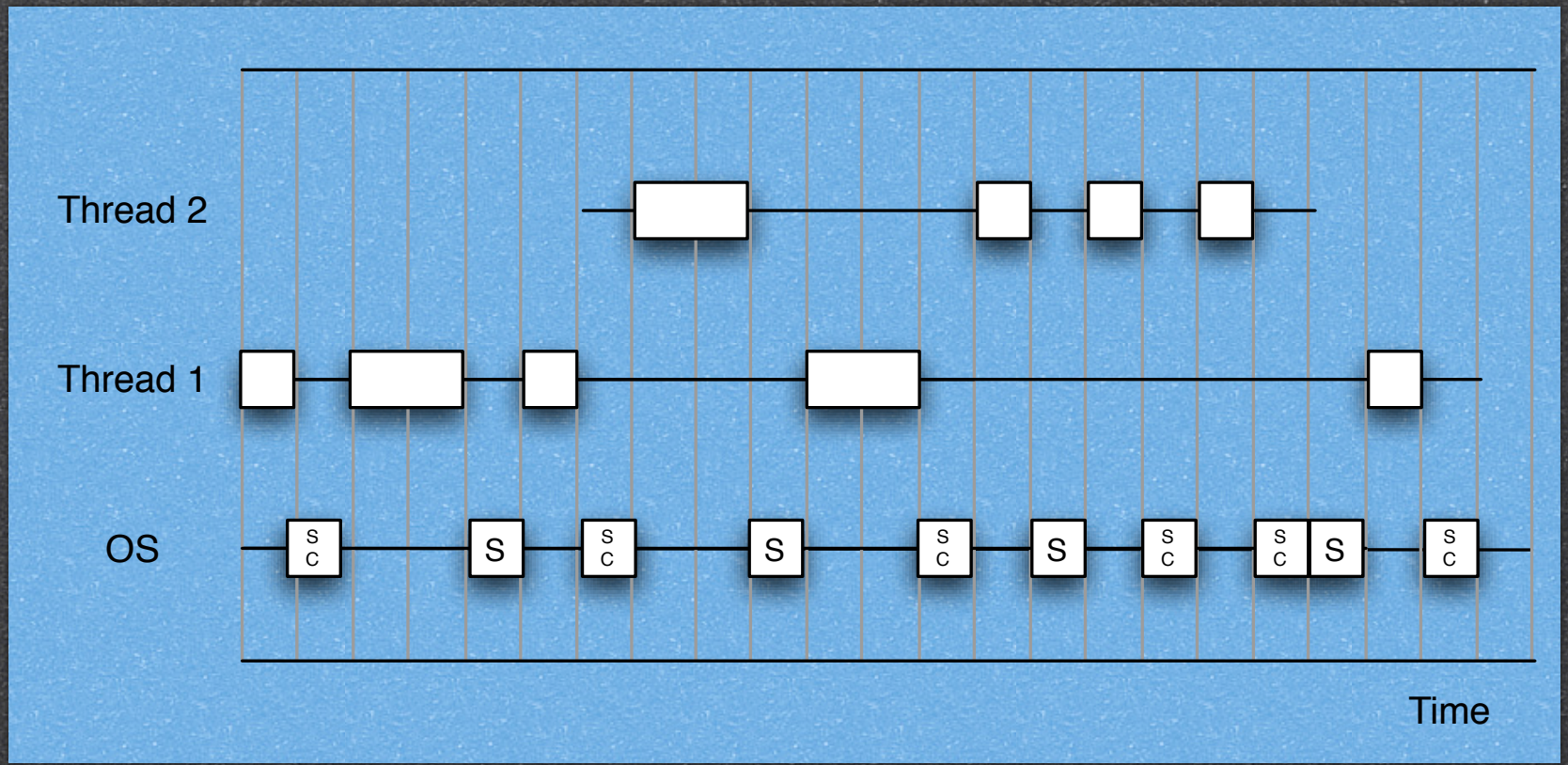


PC scheduling: preemptive

scheduling of threads



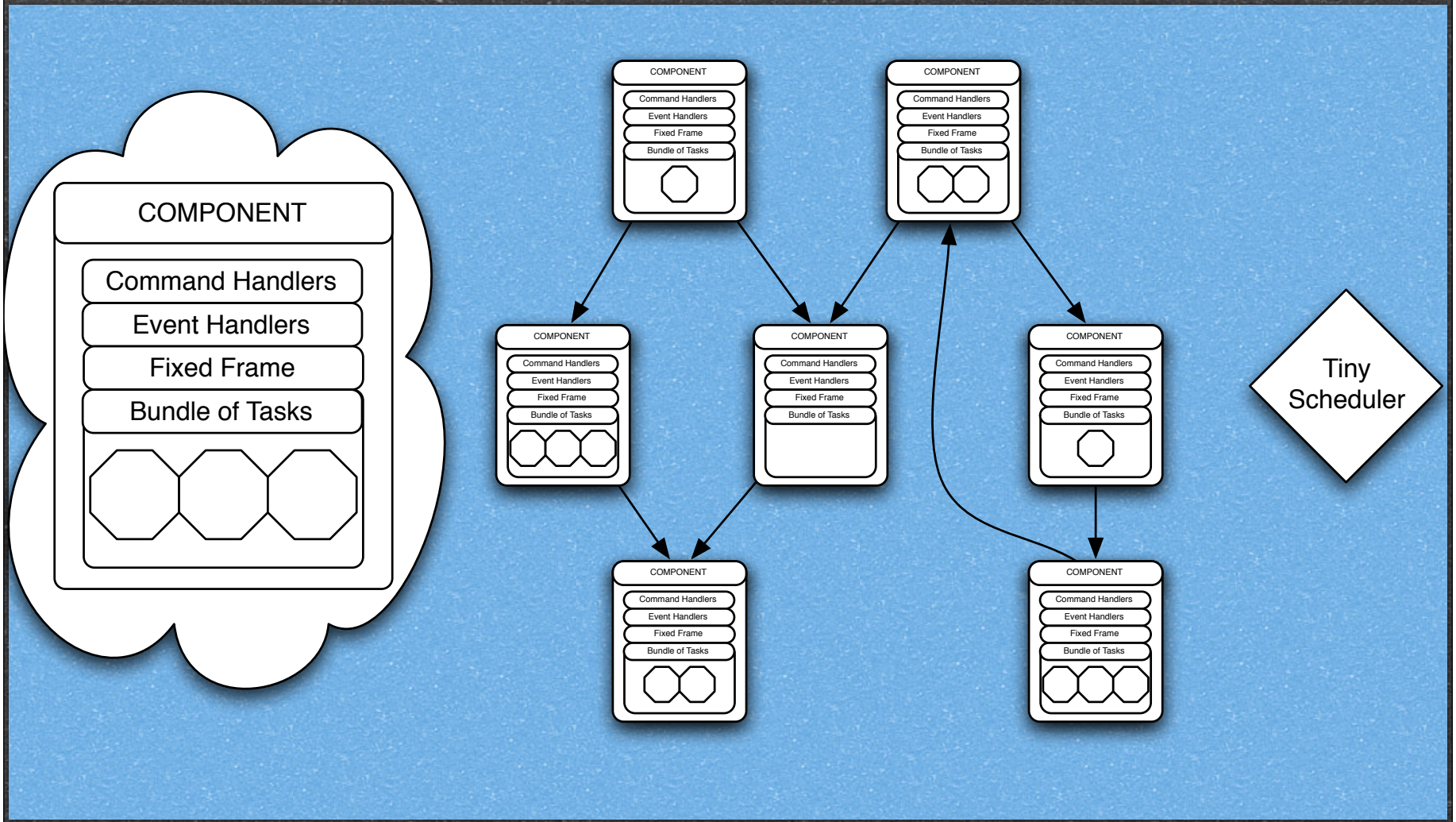
PC scheduling: preemptive, quantum-based scheduling of threads



Limitations of preemptive quantum based scheduling for tinyOS

- No or few control parallelism => core CPU must be scheduled at the frequency of the devices (RF: 1/50µs)
- Not enough memory to assign a variable stack per “thread”
  - -> Event-Based Scheduling. Helps to save power.
- Hardware events activate CPU preemptively
- When CPU is not activated by a hardware event it may do computation for TASKS.

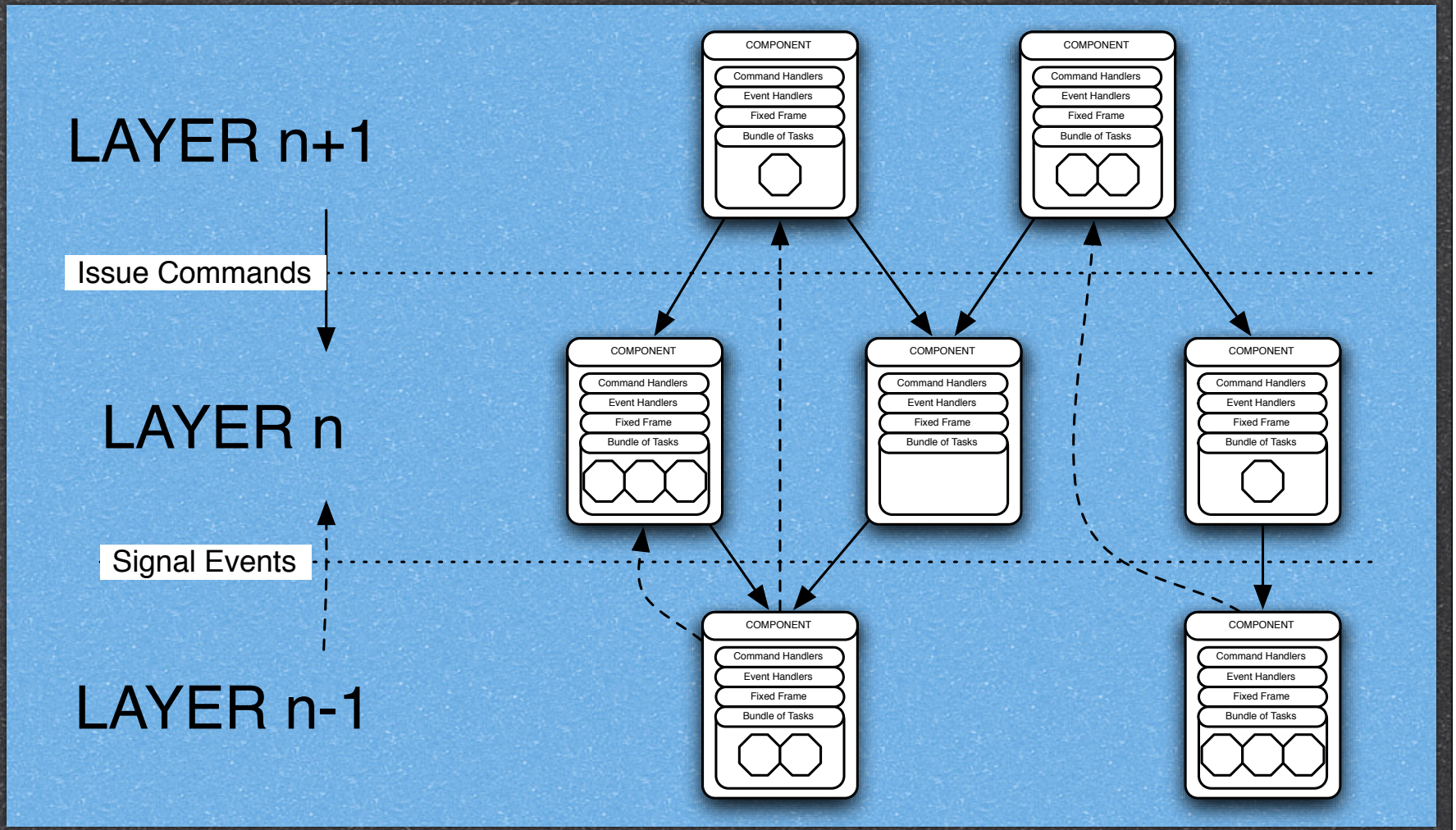
TinyOS Design



Components

- Command Handlers: provides the functionalities of the Component
- Event Handlers: defines to which events the Component can react
- Fixed Frame: defines the memory environment of the Component; is of fixed size (permits statical memory allocation)
- Bundle of tasks: Command and Event handlers should only do low-latency operations (see later). Complex computation must be delayed to tasks.

Components Composition



Physical Hardware

Commands

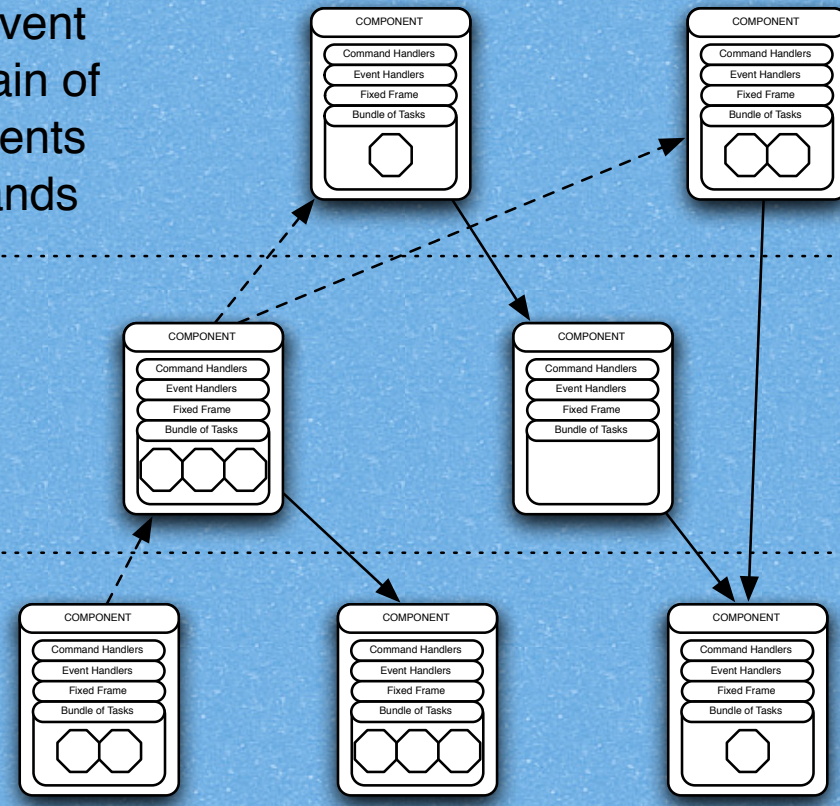
- NON Blocking requests made to Lower Level components
  - Typically deposit requests parameters into the frame of the components and conditionnaly posts a task for LATER execution
- May invoke LOWER commands
- but must NOT wait for long or indeterminate latency actions
- Must provide feedback to its caller by returning status

Events

- Invoked to deal with hardware events (possibly indirectly)
- Lowest level are the representation of the hardware events (external interrupt, timer or counter event)
  - Typically deposits information concerning the event into the frame, posts tasks, signal higher events or call LOWER level commands

Events / Commands : preventing cycles

Hardware Event trigger fountain of Software events and commands



Commands cannot Signal events

Events can call commands of LOWER layers only

Tasks

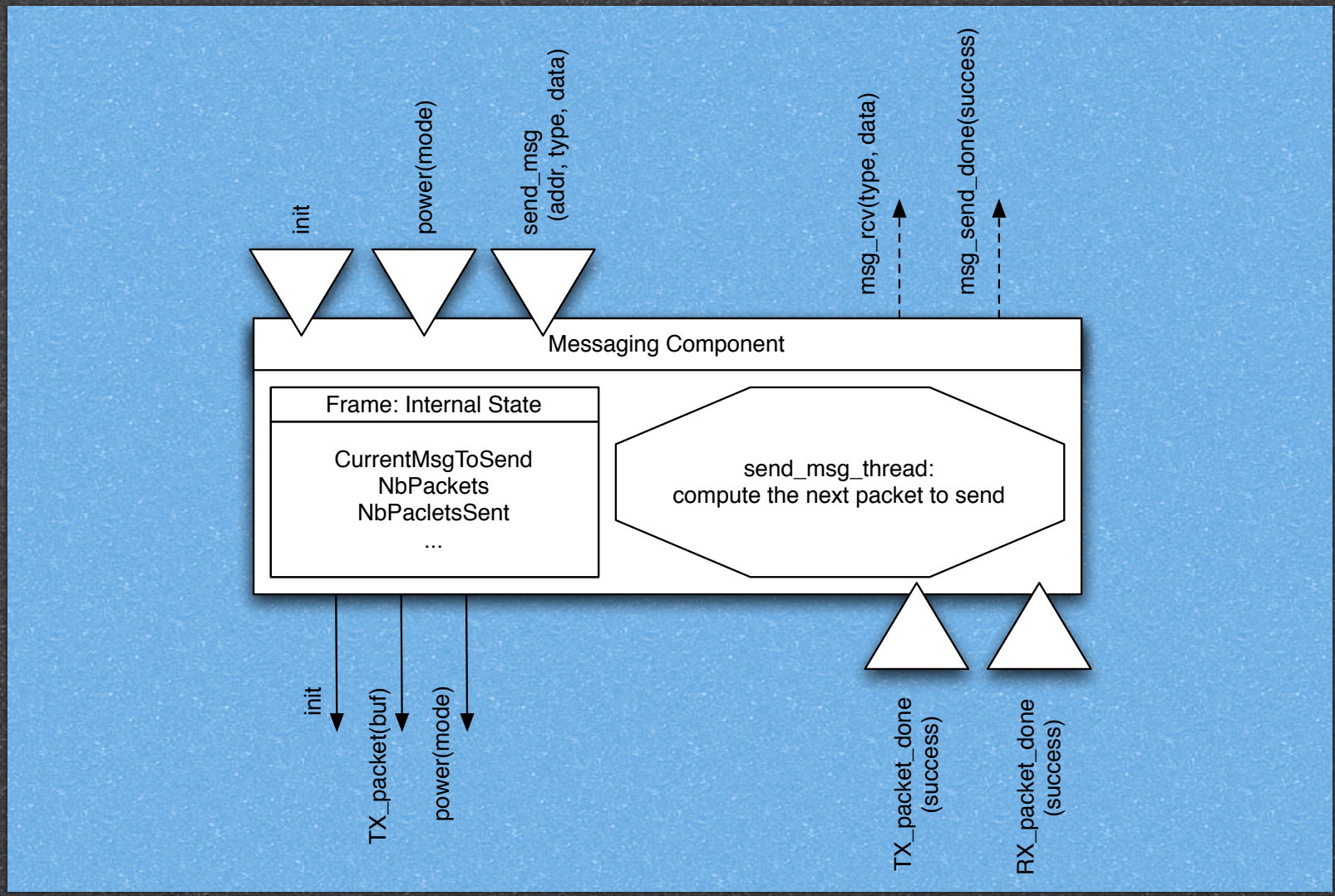
- Tasks perform the primary work
- Tasks are ATOMIC with respect to other Tasks
- Tasks RUN TO COMPLETION with respect to other Tasks
- Tasks can be preempted by Events Handlers only.
- Tasks can call LOWER level command or signal HIGHER Level events and post other Tasks of the same component.
- Tasks must NEVER block or spin-wait

- Events and commands ~ instantaneous state transition
- Tasks provide a way to incorporate arbitrary computation into the event driven model

µ-Scheduler

- The event and command handler has no use of stack since they work inside the fixed-size frame
- Tasks are run to completion
  - Thus, we need a single stack!
- The scheduler is a simple FIFO scheduler
  - Tasks are run one after the other, in the order of posts
  - more sophisticated schedulers could be used (priority/deadline based)
- The scheduler is power-aware: when the task queue is empty, the CPU is put to sleep; only peripherals are left operating

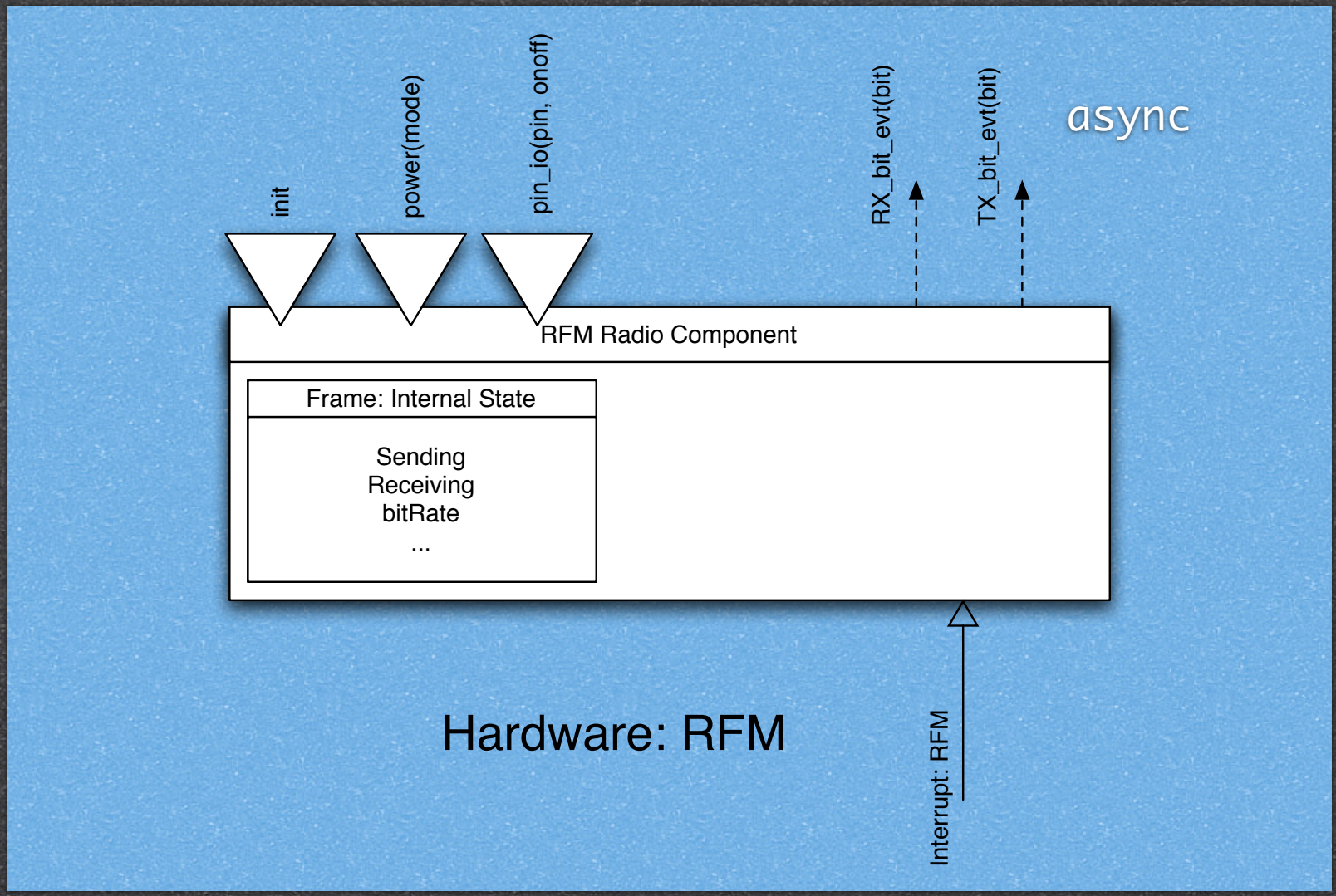
A Component example



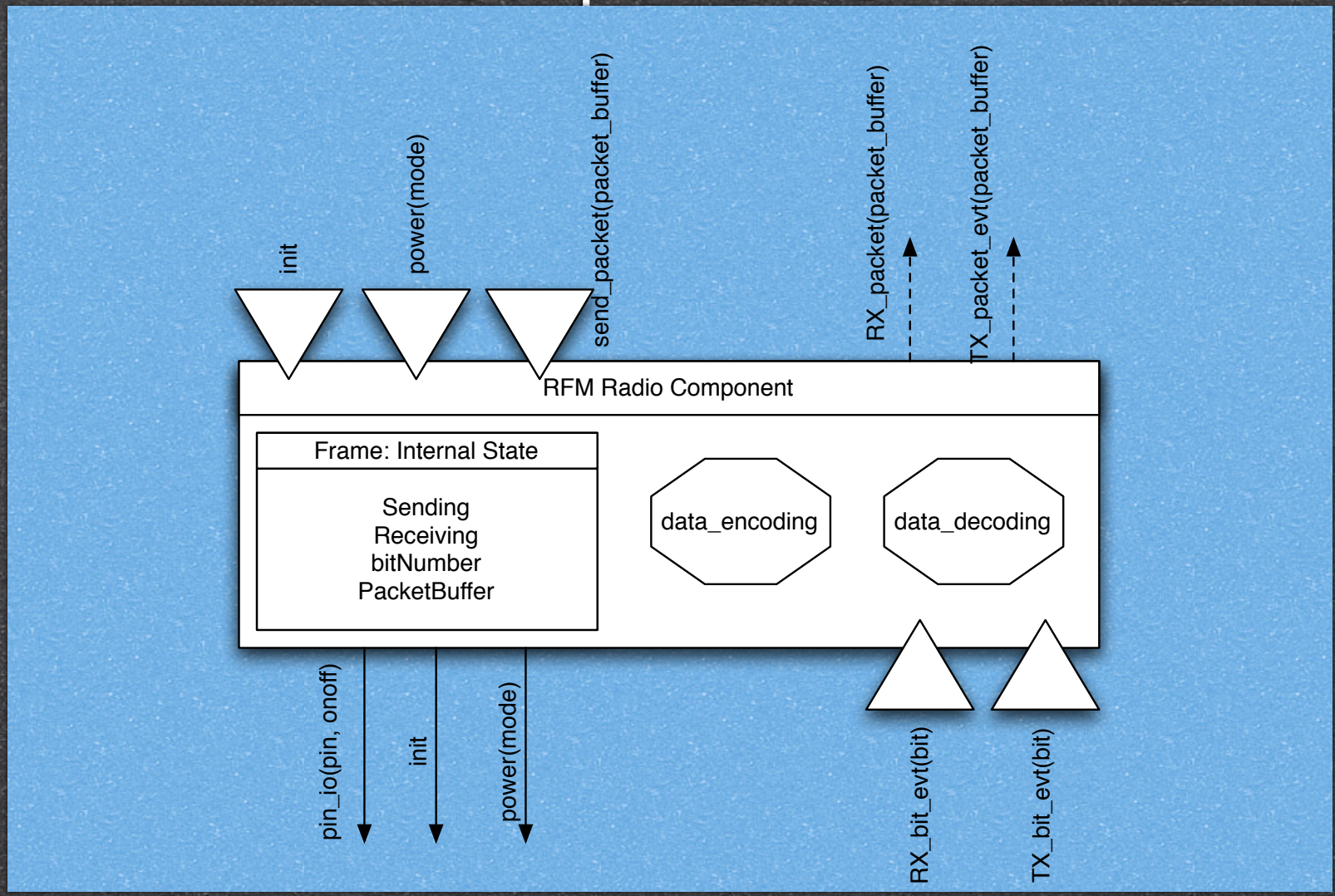
Components Types

- Hardware Abstractions (e.g. RFM radio component)
- Synthetic Hardware Components (e.g. Radio Byte Transmitter)
- High-Level Software Components (e.g. Messaging Module)

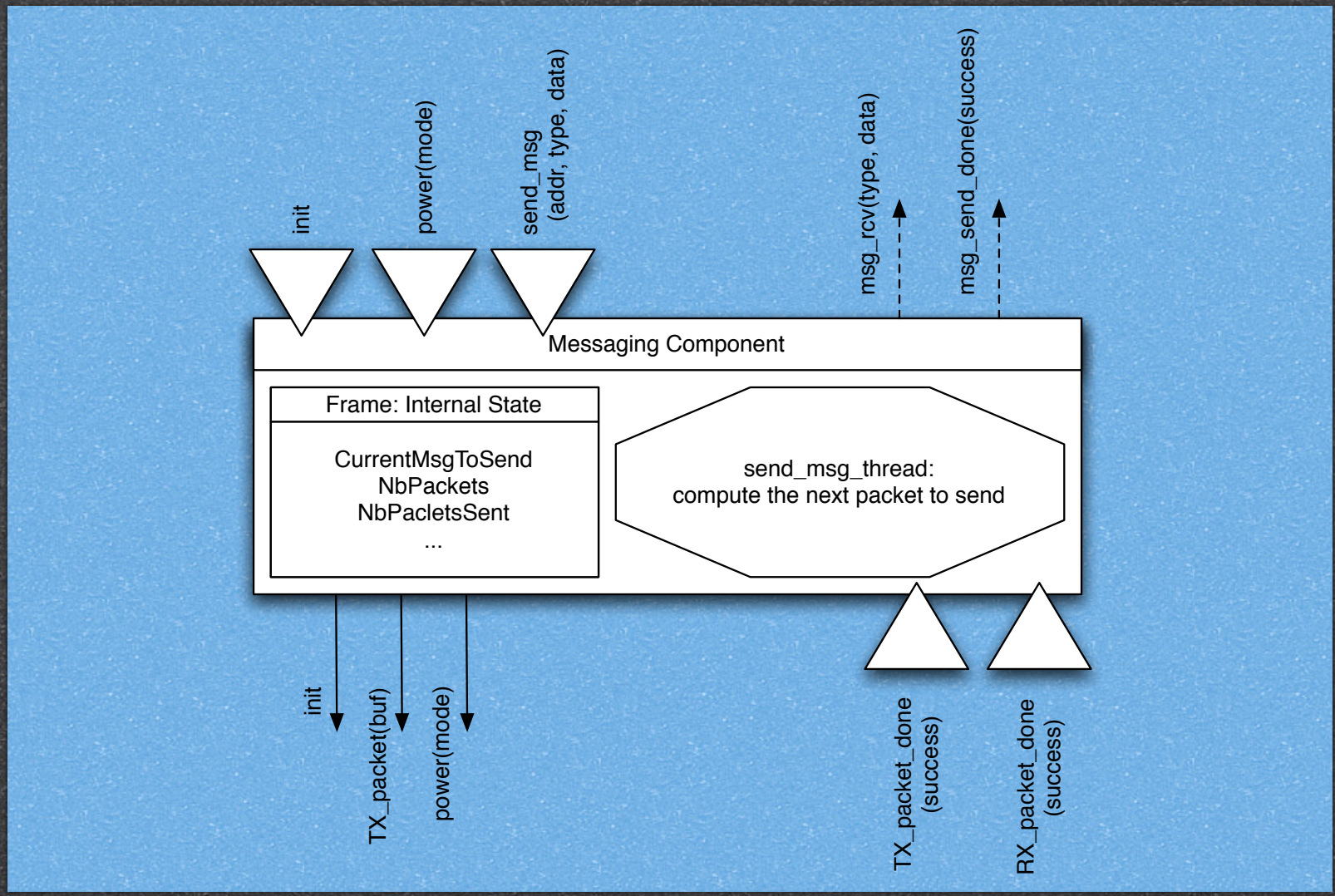
Hardware Abstractions



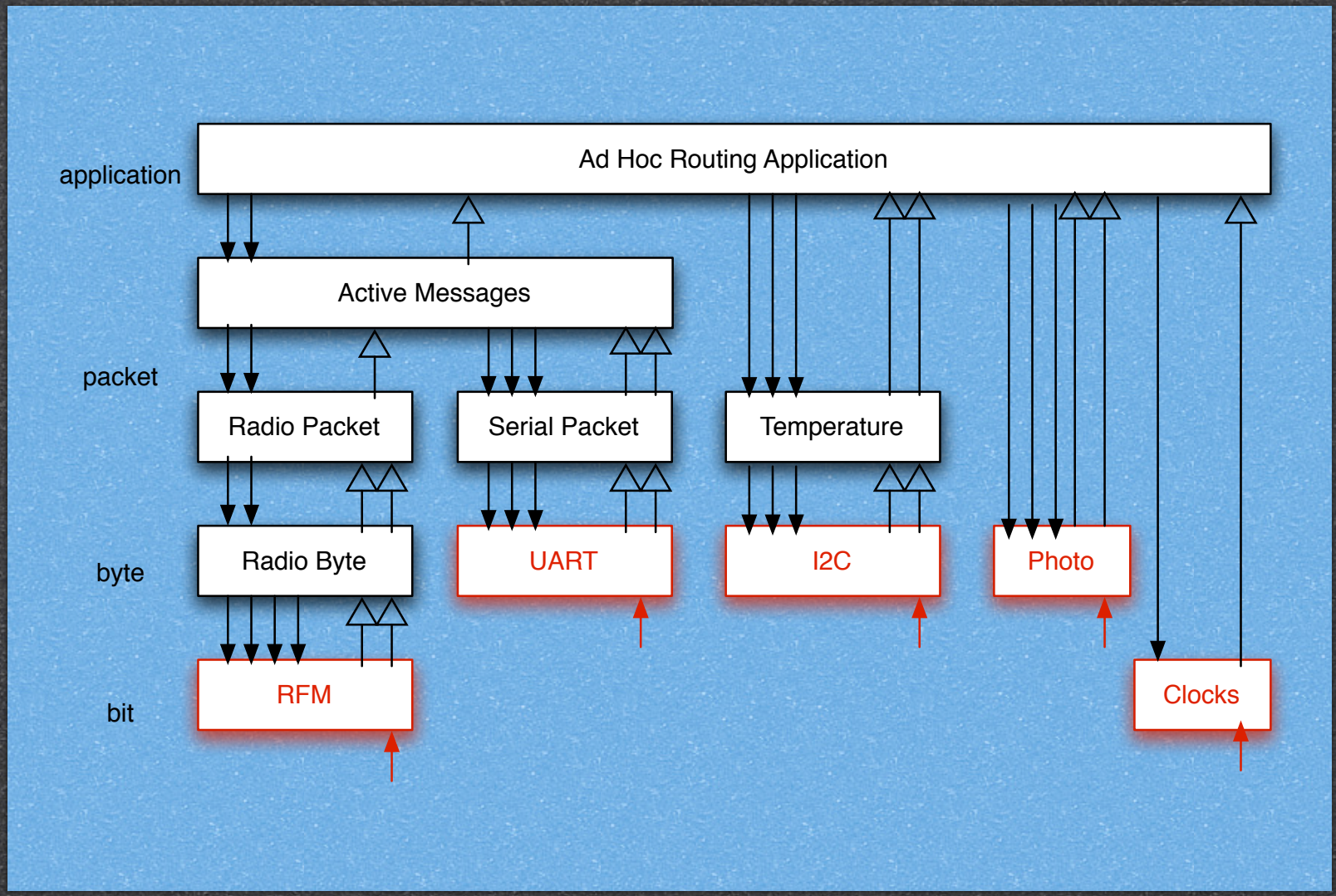
Synthetic Hardware Components



High-Level software



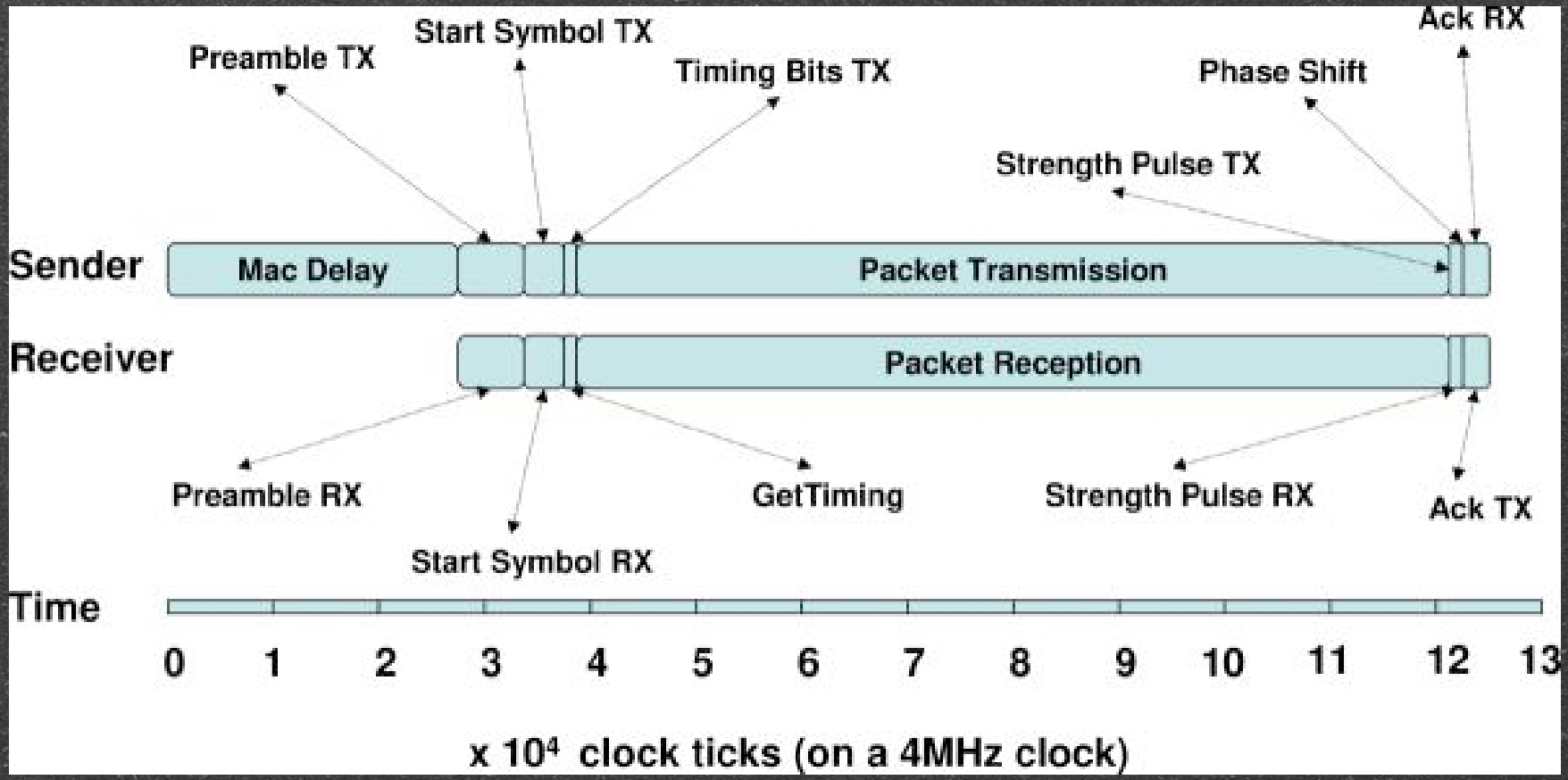
Whole-System



A note on the Network Layer

- Message Abstraction is the Active Messages
  - Each message holds a 8-bit integer
  - This integer defines which function should handle the message itself at the receiver.
- Messages hold a signal strength field filled-in by the receiver
- When the sender has sent a message, a sendDone event is signaled, with a “ackowledged” parameter
  - a non-acknowledged message is not necessarily non-received
  - an acknowledged message is not necessarily delivered to the destination

TinyOS Networking



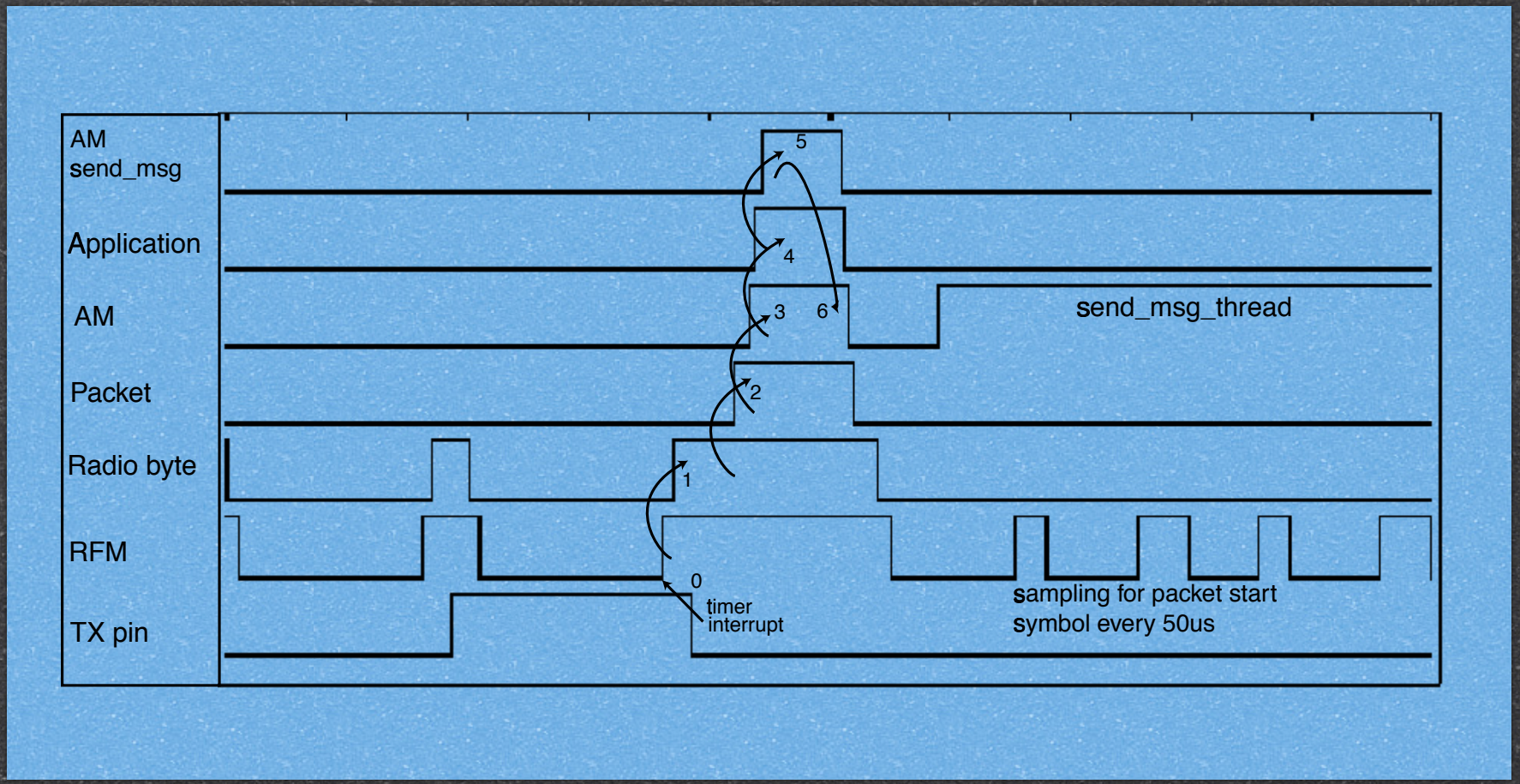
Evaluation
1- Size

Name	Code Size	Data Size
Multihop router	88	0
AM_dispatch	40	0
AM_temperature	78	32
AM_light	146	8
AM	356	40
Packet	334	40
Radio_Byte	810	8
RFM	310	1
Photo	84	1
Temperature	64	1
UART	196	1
UART_packet	314	40
I2C_bus	198	8
Processor_init	172	30
TinyOS Scheduler	178	16
C-Runtime	82	0
Total	3450	226

Evaluation
2- Concurrency

Operations	Cost (cycles)	Time (μs)	Normalized to byte copy
Byte copy	8	2	1
Post an Event	10	2.5	1.25
Call a Command	10	2.5	1.25
Post a task to scheduler	46	11.5	6
Context switch overhead	51	12.75	6
Interrupt (hardware cost)	9	2.25	1
Interrupt (software cost)	71	17.75	9

Evaluation
2&3 - Concurrency & Modularity



Evaluation
4- Limited physical parallelism and controller hierarchy

Components	Packet reception breakdown	Percent CPU Utilization	Energy (nJ/bit)
AM	0.05%	0.02%	0.33
Packet	1.12%	0.51%	7.58
Radio handler	26.87%	12.16%	182.38
Radio decode task	5.48%	2.48%	37.2
RFM	66.48%	30.08%	451.17
Radio Reception	-	-	1350
Idle	-	54.75%	-
Total	100.00%	100.00%	2028.66

Components	Packet transmission breakdown	Percent CPU Utilization	Energy (nJ/bit)
AM	0.03%	0.01%	0.18
Packet	3.33%	1.59%	23.89
Radio handler	35.32%	16.90%	253.55
Radio encode task	4.53%	2.17%	32.52
RFM	56.80%	27.18%	407.17
Radio Transmission	-	-	1800
Idle	-	52.14%	-
Total	100.00%	100.00%	4317.89

nesC

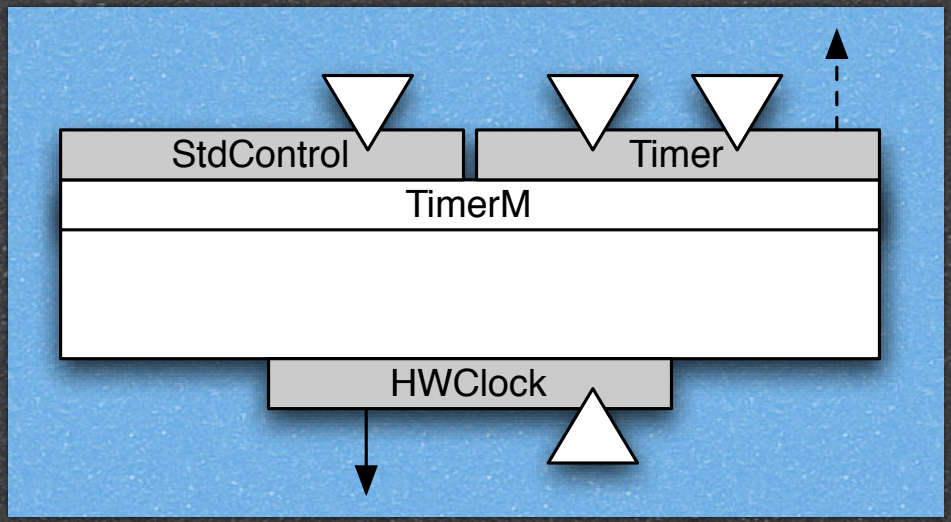
Programming Language of TinyOS

Overview

- nesC is an extension of C
- whole-program analysis
  - safety
  - performance (modularity through code)
- nesC is a static language
  - no dynamic memory allocation
  - call-graph fullly known at compile-time
- nesC supports and reflects TinyOS's design
  - Components
  - event-based concurrency

Example Component

```
module TimerM {
  provides {
    interface StdControl;
    interface Timer;
  }
  uses interface Clock as Clk;
} ...
```




```
interface StdControl {
  command result_t init();
}
```

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

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

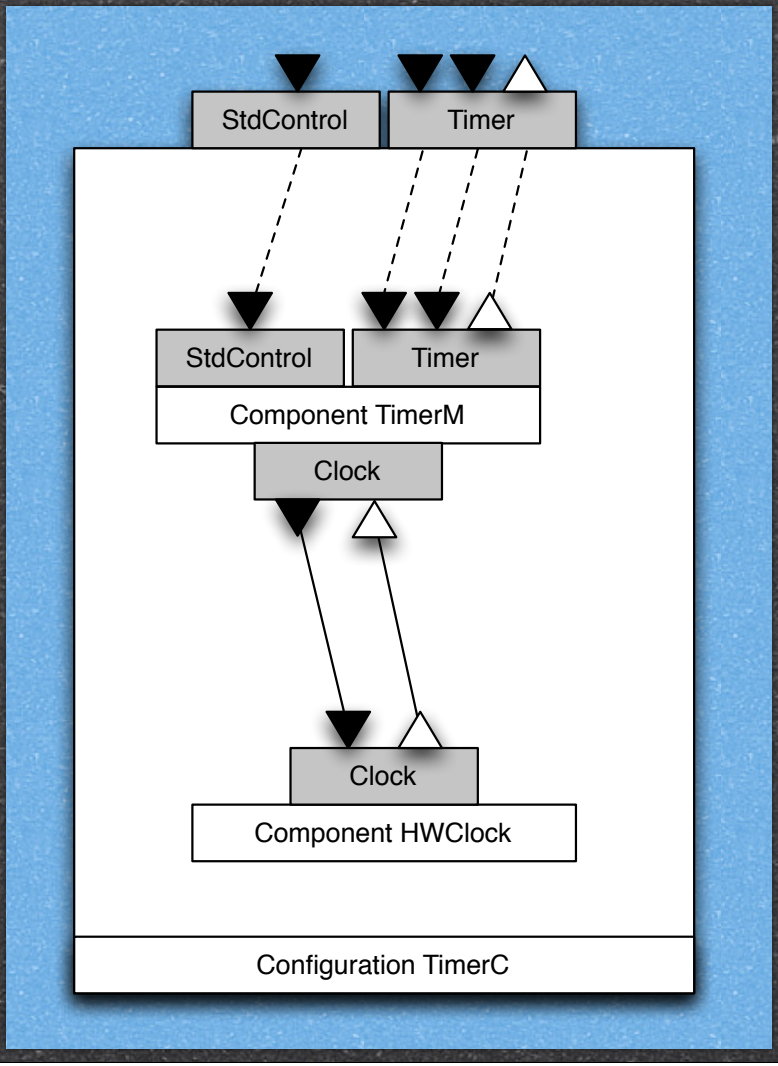
Types of Components

- Modules
  - provide application code
  - implement one or more interface
- Configurations
  - connect (“wire”) components together
  - interfaces used by components are connected to interfaces provided by others
- Every nesC application is described by a toplevel configuration

Nomenclature

- A command or event f in an interface i is named i.f
- A command call is like a regular function call prefixed with the keyword **`call`**
- An event signal is like a regular function call prefixed with the keyword **`signal`**
- A command definition is prefixed with the keyword **`command`**
- An event definition is prefixed with the keyword **`event`**

Building over TimerM: TimerC



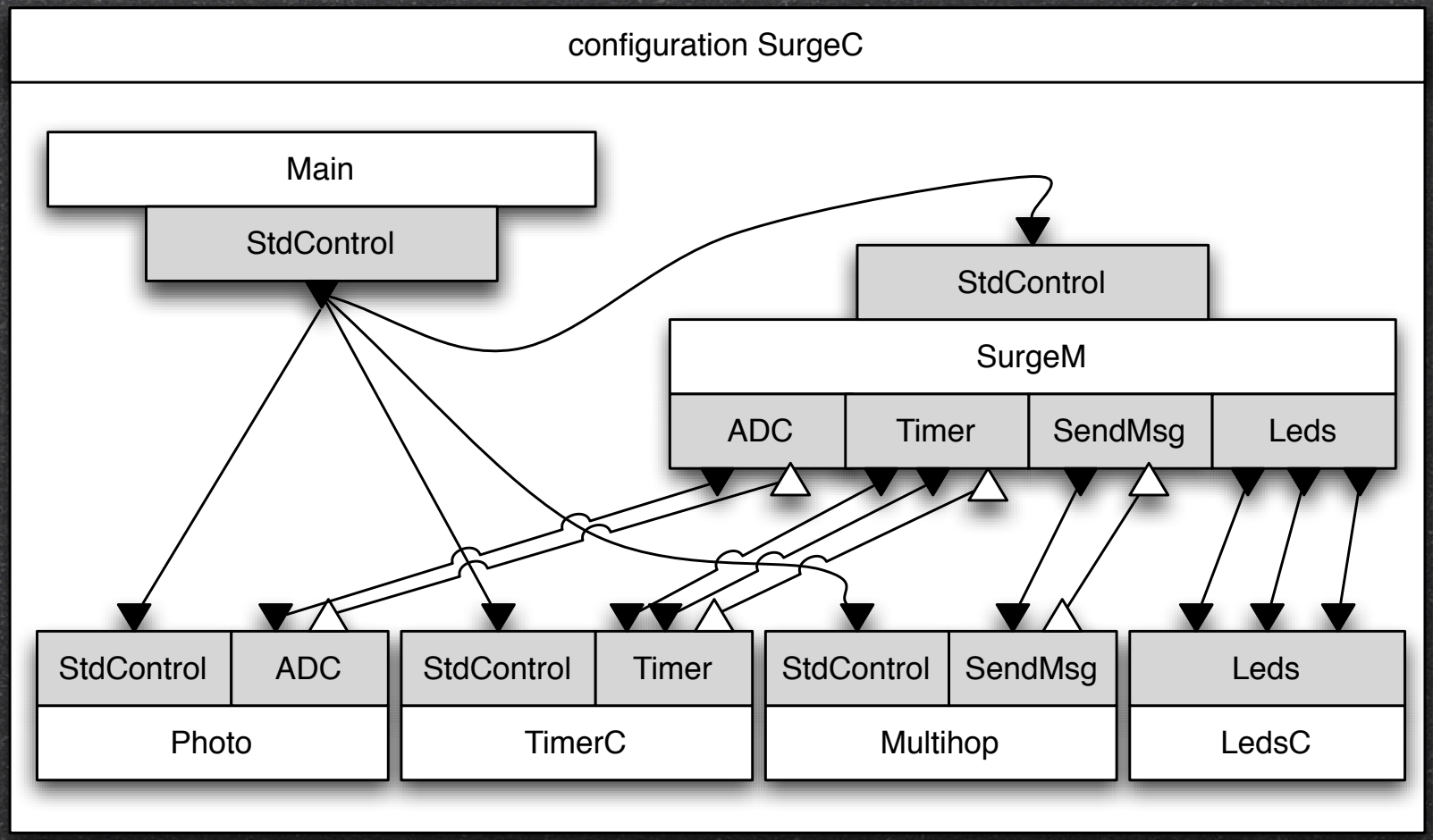

```
configuration TimerC {
 provides {
  interface StdControl;
  interface Timer;
 }
}

implementation {
 components TimerM, HWClock;

 StdControl = TimerM.StdControl;
 Timer = TimerM.Timer;

 TimerM.Clk -> HWClock.Clock;
}
```

More complete Example: The Surge Application



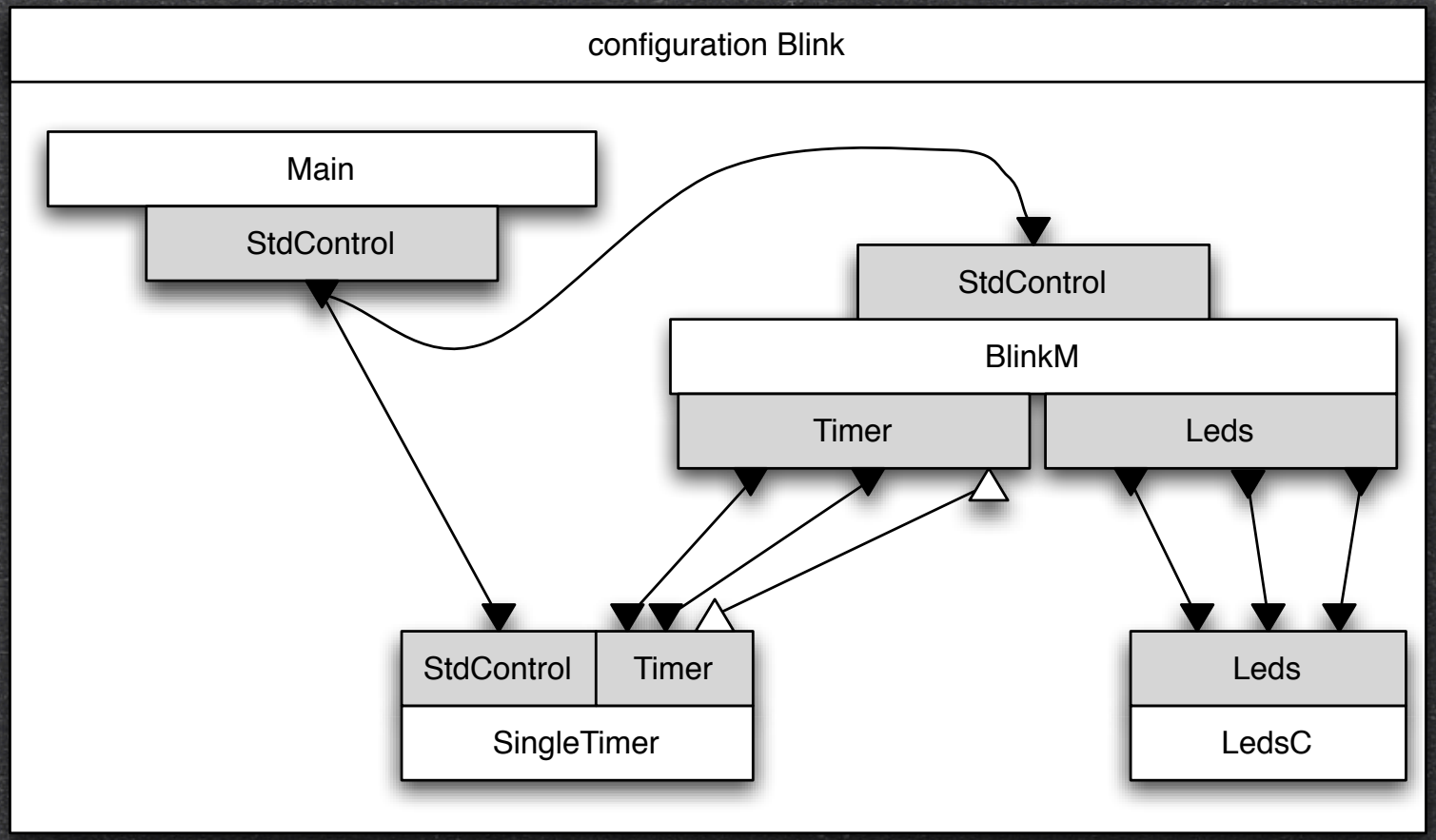
A Simple Example: the Blink Application

- TinyOS/nesC Tutorial.
- Blink application: The red LED is alternatively turned on and off every seconds.
- Events model: the only event we react to is the Clock (LEDs are simple enough to provide no events in response to commands)
- We use existing components:
  - ledsC (a configuration implementing the leds interface)
  - SingleTimer (a component to have a single timer firing at a fixed frequency)

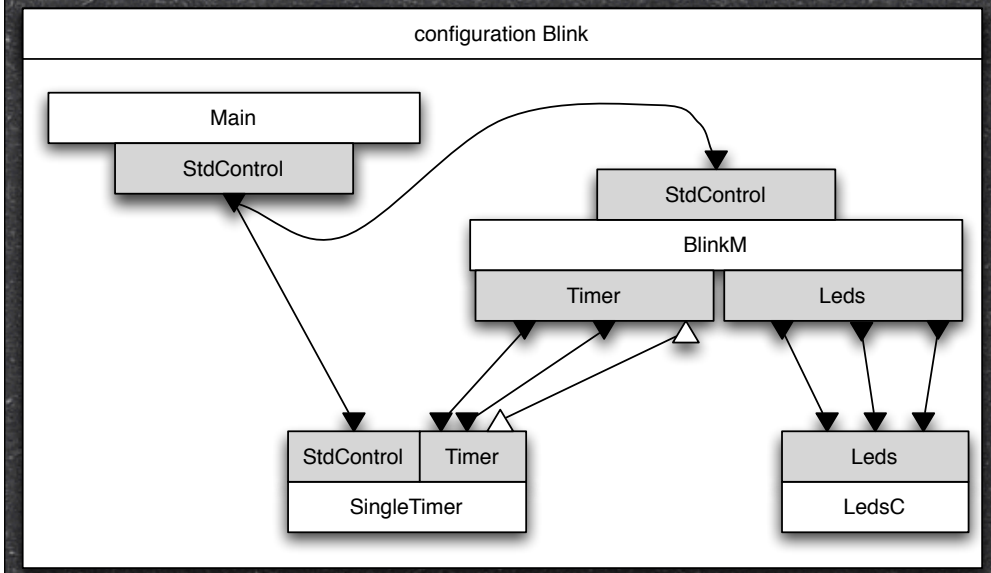
Higher level configuration

- Every application is described by a configuration
- Blink is composed by two components:
  - a module called “BlinkM.nc”
  - a configuration called “Blink.nc”
  - The configuration wires the different components together
  - while the module implements the actual behavior of the application
- Blink.nc is used to wire BlinkM.nc to the other modules.

The configuration BlinkC at a glance



Blink: the code




```
configuration Blink {
}
implementation {
 components Main, BlinkM,
    SingleTimer, LedsC;

 Main.StdControl ->
    BlinkM.StdControl;
 Main.StdControl ->
    SingleTimer.StdControl;
 BlinkM.Timer ->
    SingleTimer.Timer;
 BlinkM.Leds ->
    LedsC;
}
```

configuration Blink { }

- A configuration can provide and use interfaces.
- If so, they are declared inside the { }

- This provides the developper with the ability to compose configurations.
- Example: LedsC is a configuration.

```
implementation {
 components ...;
 ... -> ...
}
```

- components specifies the set of components that this configuration references.
- The rest wires the components together, connecting their respective interfaces.
- Main is a special component, the first to be executed in a TinyOS application. It uses only the StdControl interface.

StdControl interface

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

- init() is called when the component is initialized
- start() when it is started (executed for the first time)
- stop() when it is stopped (e.g. to power-off)
- init, start and stop may be called more than once
  - all inits before any start or stop
  - all stops follow a start

initialization

- All three of these commands have “deep” semantics.
- Calling init() on a component must make it call init() on all of its sub-components.
- SingleTimer and BlinkM init() function is called because they provide the StdControl interface and the Main.StdControl interface is linked to the corresponding interfaces of the components.
- However, LedsC does not provide the StdControl interface, thus the init() function of the BlinkM module MUST call explicitely the init() function of the Leds interface.
- The same rule apply for start() and stop().

Binding

- nesC uses arrows to determine relationships between interfaces.
- -> is like “binds to”.
- The left side of the arrow binds an interface to an implementation on the right side.
- nesC supports multiple implementations of the same interface. Timer is an example.
- Wiring can be implicit: BlinkM.Leds -> LedsC; is a shorthand for BlinkM.Leds -> LedsC.Leds;

The BlinkM.nc module

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

This module provides StdControl
Thus, it must implement all the StdControl commands and can signal all the StdControl Events (none in this case)

It also uses the interface Timer and Leds, thus it can call Timer and Leds commands (and sometimes must, like init), and MUST implement Timer and Leds Events (none for Leds, but some for Timer)

Leds interface

- The Leds interface is really simple, and provides only commands
- some like init()
- others like redOn(), redOff(), etc... to control the Leds.

Timer Interface

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

start(...) is not related to start() of StdControl. It is used to specify the type of timer and the interval

The fired event is called by the lower-level component (here SingleTimer) when the timer expires.

The unit of interval are millisecond; the valid types are TIMER_REPEAT and TIMER_ONE_SHOT

BlinkM MUST implement this event

BlinkM.nc, continued

```
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();
  }
```

BlinkM.nc, continued

```
  event result_t Timer.fired()
  {
    call Leds.redToggle();
    return SUCCESS;
  }
}
```

Compiling

- See later. Use Makefile provided by TyniOS.
- make will eventually call ncc, the nesC compiler
  - ncc -o main.exe -target=pc Blink.nc
- Makefile provides simpler interface, and self-documentation : make pc docs builds the
  - docs/nesdoc/pc/
- web pages to visualize the compiled dependency graph.

Exercise

- Leds provides the following commands:
  - (red|green|blue)(On|Off)()
  - Toggle(red|green|blue)()
- Modify the blinkM implementation to display the lower three bits of a counter that is incremented every second.

```
//Configuration is unchanged
//Module (BlinkM.nc)
module BlinkM {
 provides {
  interface StdControl;
 }
 uses {
  interface Timer;
  interface Leds;
 }
}
implementation {
 int counter;

 command result_t
   StdControl.init() {
  counter = 0;
  return call Leds.init();
 }
```

```
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() {
 counter++;
 if(counter & 1)
  call Leds.redOn();
 else
  call Leds.redOff();
 .... }
```

Interface Instances

```
module SenseM {
  provides { ... }
  uses {
    interface StdControl as ADCControl;
    interface ADC;
  }
}
```

- Gives the ability to use multiple times the interface in the same module by renaming it.
- ADCControl is used for wiring to a StdControl interface
- In fact, interface StdControl is just a shorthand for interface StdControl as StdControl
- Beware to wiring: SenseM.ADC -> Sensor is just a shorthand for SenseM.ADC -> Sensor.ADC; BUT SenseM.ADCControl -> Sensor is NOT A SHORTHAND FOR SenseM.ADCControl -> Sensor.ADCControl. IT IS A SHORTHAND FOR SenseM.ADCControl -> Sensor.StdControl !

Parameterized Interfaces

```
provides interface Timer[uint8_t id];
...
SenseM.Timer -> TimerC.Timer[3];
```

- A parameterized interface allows a component to provide multiple instances of an interface that are parameterized by a runtime or compile-time value.
- TimerC declare providing this interface, thus it provides 256 different instances of the Timer interface.
- By wiring the Timer interface in each component to a separate instance of the Timer interface provided by TimerC, each component can run its own “private” timer.

Parameterized Interfaces and unique

```
provides interface Timer[uint8_t id];
...
SenseM.Timer -> TimerC.Timer[unique(“Timer”)];
```

- To avoid identifier overlappings, nesC includes the unique and uniqueCount routines.
- unique(char *string) is resolved AT COMPILE TIME as a number different in each call, for the same string
- uniqueCount(char *string) is resolved AT COMPILE TIME as the number of counters for this string.
- Developpers have to use the same string to denote the same component. To avoid confusion, we use the name of the parameterized interface as an argument to the unique() function.

Tasks

- A Task is declared in the implementation part of a module using the syntax
  - `task void taskname() {...}`
- Tasks must return void and take no arguments.
- To dispatch (schedule) a task for (later) execution, use the syntax
  - `post taskname();`
- Posting a task is authorized within any context (command, event or even another task).

Tasks & Concurrency

```
async event result_t ADC.dataReady(uint16_t data) {
    putdata(data);
    post processData();
    return SUCCESS;
  }

task void processData() {
    int16_t i, sum=0;

    for (i=0; i < size; i++)
      sum += (rdata[i] >> 7);

    display(sum >> log2size);
  }
```

Concurrency & Atomicity

- Asynchronous Code (AC): code that is reachable from at least one interrupt handler
- Synchronous Code (SC): code that is only reachable from tasks.
  - Synchronous Code is atomic with respect to other Synchronous Code.
- Any update to shared state from AC is a potential Race
- Any update to shared state from SC that is also updated from AC is a potential Race.

Exemple: SurgeM (non-concurrent)

```
module SurgeM {
 provides interface
 StdControl;
 uses interface ADC;
 uses interface Timer;
 uses interface Send;
} implementation {
 uint16_t sensor;

 command result_t
 StdControl.init() {
  return call Timer.start
  (TIMER_REPEAT, 1000);
 }
```

```
event result_t
Timer.fired() {
 call ADC.getData();
 return SUCCESS;
}

event result_t
ADC.dataReady(uint16_t
data) {
 sensor = data;
 ... send message ...
 return SUCCESS;
}
...
}
```

How to avoid RC Example: SurgeM

```
module SurgeM {
  ...
} implementation {
  bool busy;
  norace uint16_t sensor;

  event result_t
      Timer.fired() {
    bool localBusy;
    atomic {
      localBusy = busy;
      busy = TRUE;
    }
    if( !localBusy )
      call ADC.getData();
    return SUCCESS;
  }

  ...
```

```
task void sendData() {
  adcPacket.data = sensor;
  call Send.send(&adcPacket,
     sizeof(adcPacket.data));
  return SUCCESS;
}

event result_t
   ADC.dataReady(uint16_t d)
{
  sensor = d;
  post sendData();
  return SUCCESS;
}

event result_t Send.sent(...)
{ ...
  atomic busy=FALSE;
  ... }
```

Concurrency & Atomicity

- Race Free Invariant: Any update to shared state is either not a potential race condition (SC only), or occurs within an atomic section
  - The compiler enforces this condition through the following rule:
  - If a variable x is accessed by AC, then any access of x outside of an atomic statement is a compile-time error.
- To remove the error, the programmer must either add an atomic section, or move the offending code in a task.
- Optionnaly, the programmer may hint to the compiler that some variable will not provoke a race condition (Dangerous).

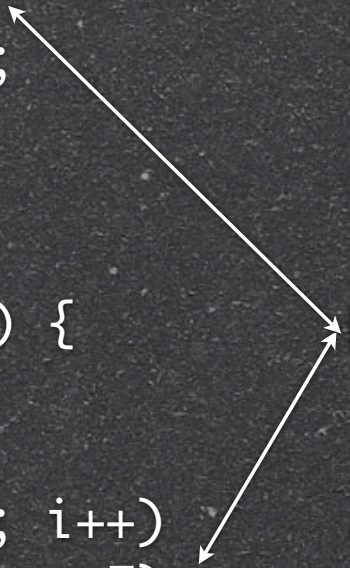
Exercise

```
async event result_t ADC.dataReady(uint16_t data) {
    putdata(data);
    post processData();
    return SUCCESS;
  }

task void processData() {
    int16_t i, sum=0;

    for (i=0; i < size; i++)
      sum += (rdata[i] >> 7);

    display(sum >> log2size);
  }
```



Remove the RC

Communication: Radio & UART

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

interface SendMsg {
  command result_t send(
     uint16_t address,
     uint8_t length,
     TOS_MsgPtr msg);
  event result_t sendDone(
     TOS_MsgPtr m,
     result_t success);
}
```

Communication: Radio & UART

```
configuration GenericComm
{
  provides {
    interface StdControl as Control;

    // The interface are as parameterised by the active message id
    interface SendMsg[uint8_t id];
    interface ReceiveMsg[uint8_t id];

    // How many packets were received in the past second
    command uint16_t activity();
  }
  uses {
    // signaled after every send completion for components which
    // wish to retry failed sends
    event result_t sendDone();
  }
}
```

Active Messages

- Each message type is identified by a unique id.
- This id defines which function should be called at message reception.
- In nesC, this is easily done through parameterized interfaces:
  - you just have to wire the component which provides the implementation of the function with the correct interface number.
  - Here, we seldom use the unique("") function, since sender AND receiver must agree on the unique id.
  - We use C enumerated values defined in .h files

Example (emission)

```
bool pending;
struct TOS_Msg data;

...

command result_t
  SomeComponent.SomeCommand
     (uint16_t value) {
 IntMsg *message =
    (IntMsg *)data.data;

 if( !pending ) {
  pending = TRUE;

  atomic {
   message->val = value;
   message->src =
   TOS_LOCAL_ADDRESS;
  }
```

```
   if(call
    Send.send(TOS_BCAST_ADDR,
       sizeof(IntMsg), &data))
      return SUCCESS;

   pending = FALSE;
  }
  return FAIL;
}
```

Example (reception)

```
RfmToIntM.ReceiveIntMsg -> GenericComm.ReceiveMsg[AM_INTMSG];

 /* ... */
   event TOS_MsgPtr ReceiveIntMsg.receive(TOS_MsgPtr m) {
     IntMsg *message = (IntMsg *)m->data;
     call IntOutput.output(message->val);

     return m;
   }
```

Exercise

- Write two TinyOS Applications:
  - One which sends a counter on the network
  - The other which receives this counter and displays the lowest three bits on its LEDs.

Simulation

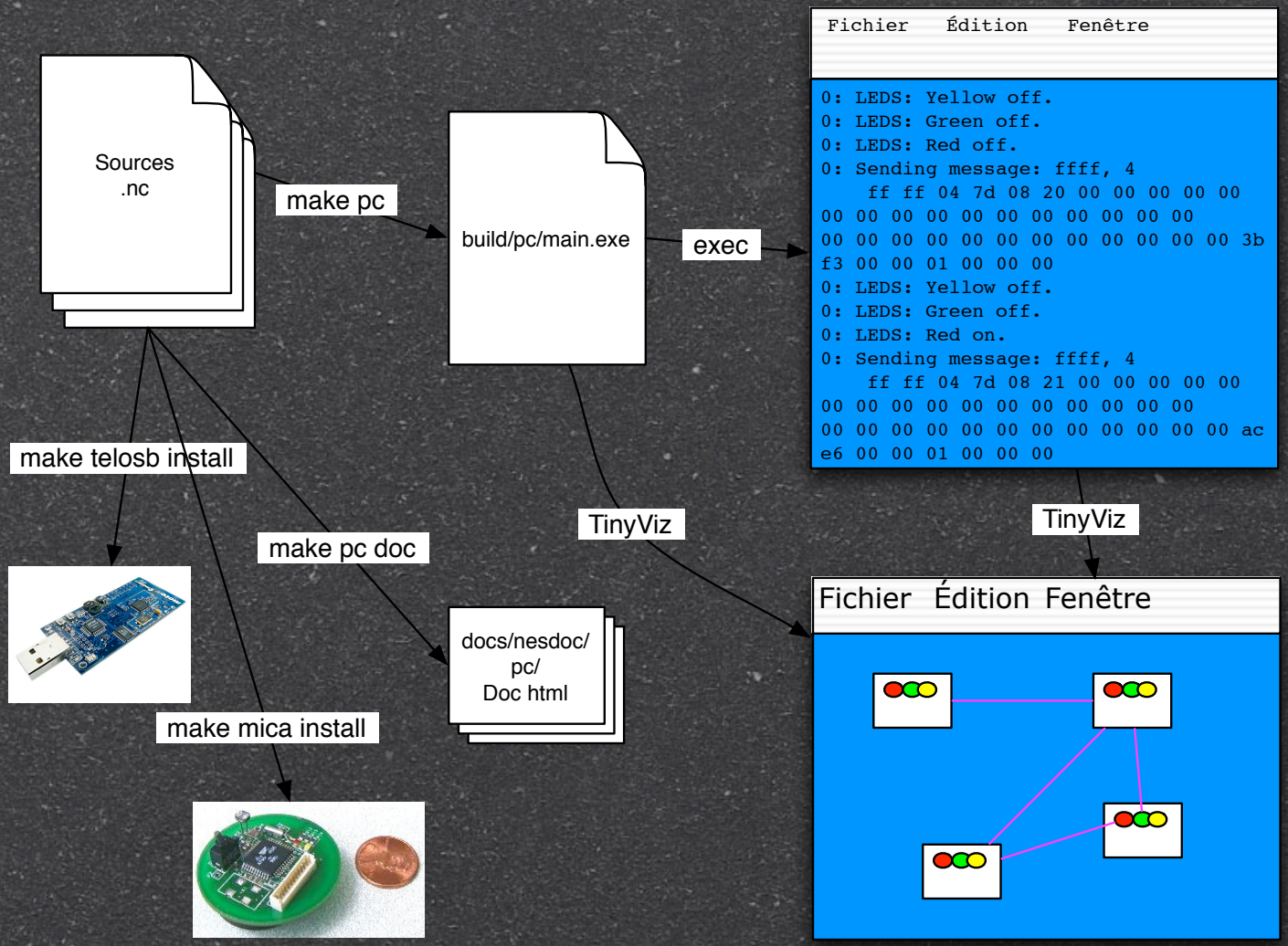
tossim

TOSSIM

- TinyOS Simulator.
- Compilation is similar to a mote architecture (make pc)
- TOSSIM can simulate 1000s nodes simultaneously
- Every mote runs the SAME TinyOS program

- TOSSIM provides debugging output
  - ASCII (default)
  - GUI (TinyViz), a Java-Based GUI.

TOSSIM and Compilation



Four key requirements

- Scalability.
  - The largest TinyOS sensor netwrok deployed was 850 nodes wide
  - The simulator should handle 10’000s
- Completeness.
  - Cover as many interactions as possible
- Fidelity.
  - behavior of the network at fine grain
  - subtle timing interactions
  - must reveal unanticipated interactions
- Bridging.
  - fill the gap between algorithms and implementation on real HW

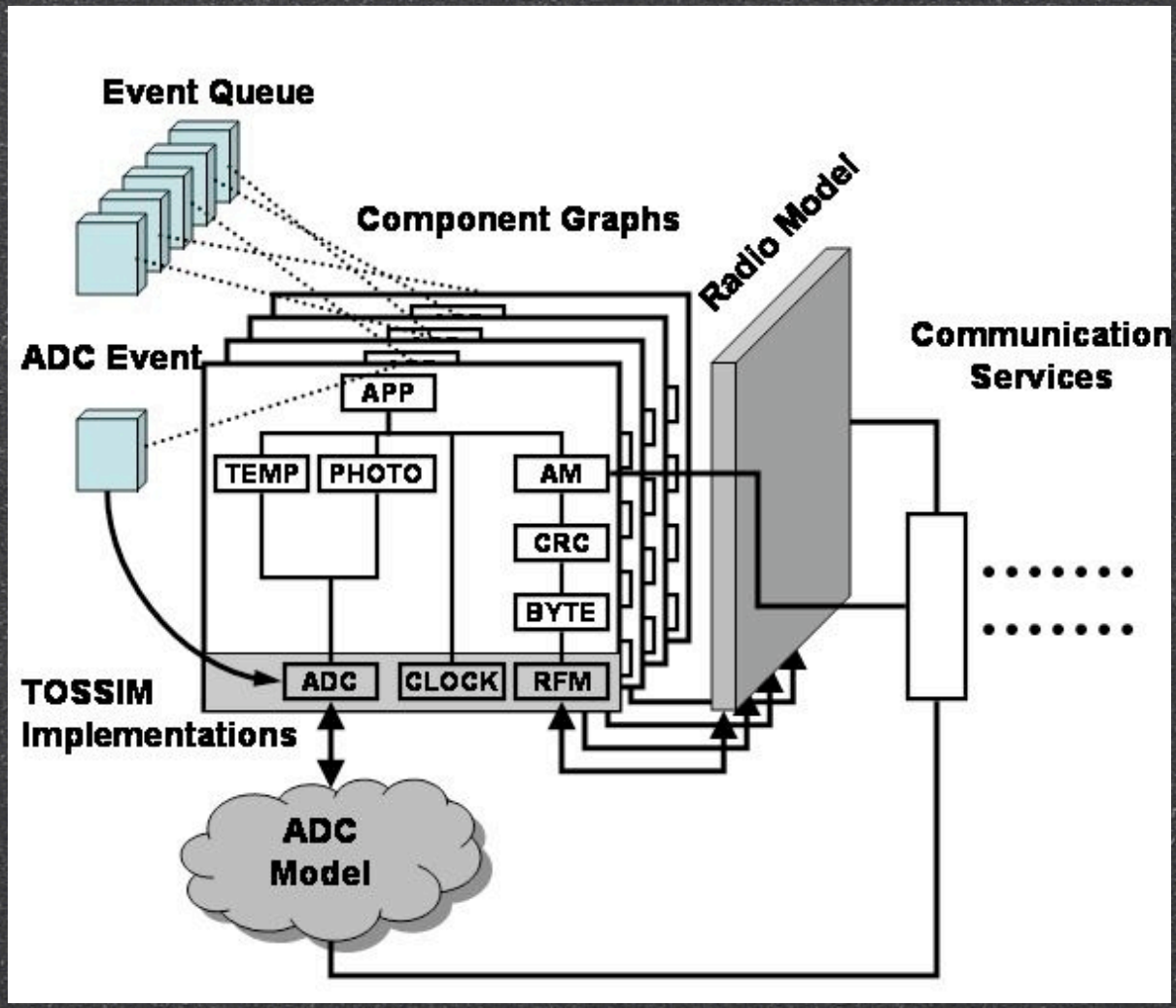
Discrete Event Simulation

- Discrete Event Simulation:
  - the simulator handles a list of events totally ordered (date)
  - the first event is poped and its actions on the current state produces a new state and potentially a set of new events that are injected in the sorted list
  - and so on
- TinyOS is event driven
- Event driven executions maps well on discrete events simulation
  - very simple simulation engine

Abstractions

- TOSSIM captures the behavior and interactions at network bit granularity.
- Tasks are run “instantly”: the virtual time does not progress during task completions
  - allowing real preemption and interleaving would reduce scalability a lot (interpretation of the code instead of execution)
  - TOSSIM will not help checking the data RCs, but the compiler should do that.

TOSSIM Architecture



TinyOS Tool Chain

- TOSSIM is “just another mote target”
  - the transistion between simulated and real networks is easy
  - the simulator runs native code, thus user can use debuggers in TOSSIM
- It also provide mechanisms for other programs to interact and monitor a running simulation

Example TOSSIM compilation

nesC TinyOS Code	Mote C Code	TOSSIM C Code

result_t StdControl.init() {
  state = 0;
  return SUCCESS;
}
	result_t Counter $StdControl$init (void) {
	  Counter$state=0;
	  return SUCCESS;
	}
		result_t Counter $StdControl$init (void) {
		  Counter$state [tos_state.current_ node] = 0;
		  return SUCCESS;
		}

Execution Model

- Core of TOSSIM: a simulator event queue
- Interrupts are modeled through simulator events (!= TinyOS events)
- A simulator event calls an interrupt handler in HW abstraction component

- TOSSIM keeps time at mote instruction clock cycle granularity (4MHz)
- At start, mote are given random times
- Every TOSSIM events happens at a given time and takes a delta time to run (non zero time to have total order of events, but sum of delta = 0)
- After running a simulation event, TOSSIM executes all the tasks on this mote

Hardware Emulation

- TinyOS abstracts each HW resource as a component.
  - ADC (Analog to Digital Converter)
  - Clock
  - Transmit strength variable potentiometer
  - EEPROM
  - boot sequence
  - several of the radio stack
  - low level components abstracting sensors (special case)
- E.G. getData() of the ADC will produce a dataReady() event later (depending on the duration of the getData operation)

Sample Execution

- SenseToLeds
  - at 1Hz, getData();
  - when dataReady(); do ledsToggle();
- timer events are 4 Million ticks appart
- ADC takes 50µs -> 200

Time (4MHz)	Action
3987340	Simulator event is dequeued Clock interrupt handler call Timer event signaled command getData(); ADC comp. puts a sim. event on the queue at 3987540 Clock puts a sim. event on the queue at 7987340
3987540	Sim. ADC event is dequeued ADC interrupt handler call ADC ready event signaled call ledsToggle();
7987340	Simulator event is dequeued Clock interrupt handler call ...

Network: Communication Service

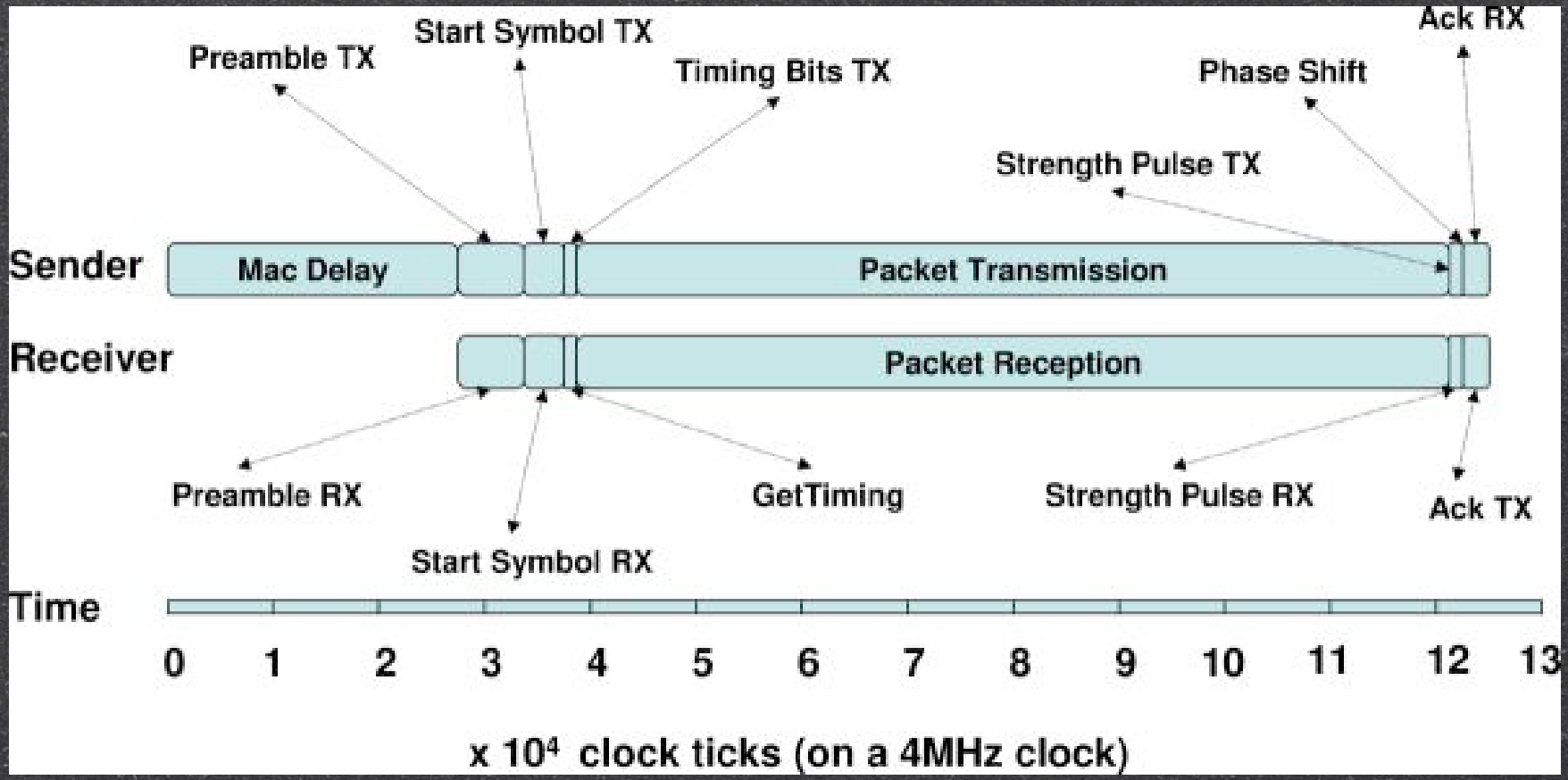
- TOSSIM provides mechanisms that allow PC applications to drive, monitor and actuate simulation through TCP/PI
- TOSSIM signals events with data to applications
  - e.g. debug messages
  - radio and UART packets sent
  - actuators (leds) and sensor readings
- Application can
  - change radio link probabilities
  - change sensor reading values
  - turn motes on and off
  - inject radio and UART packets

TinyViz is built on this interface

Data Link Layer

- Most Complex System of TOSSIM
- Provides the networking stack of TinyOS
- Must simulate the networking stack at high fidelity but keeping scalability
- Choice: bit-level simulation.
  - The user provides a single probability per link: the probability that a bit is flip.

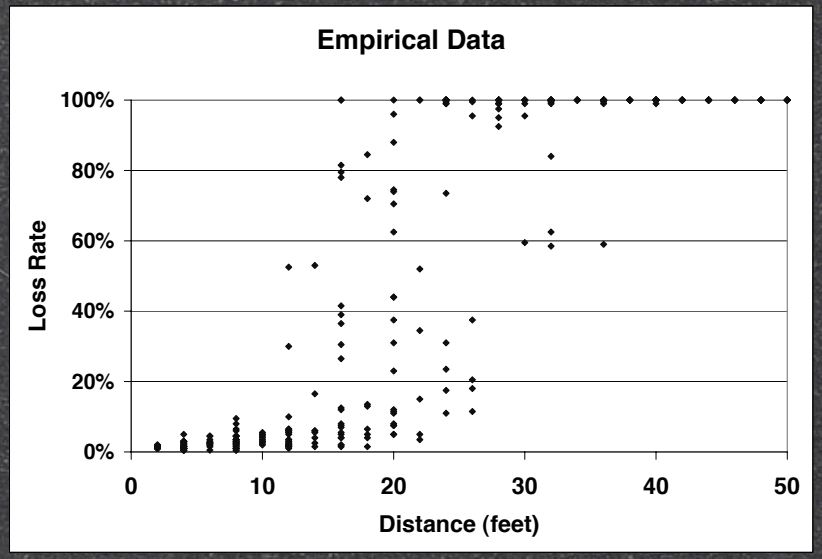
TinyOS Networking

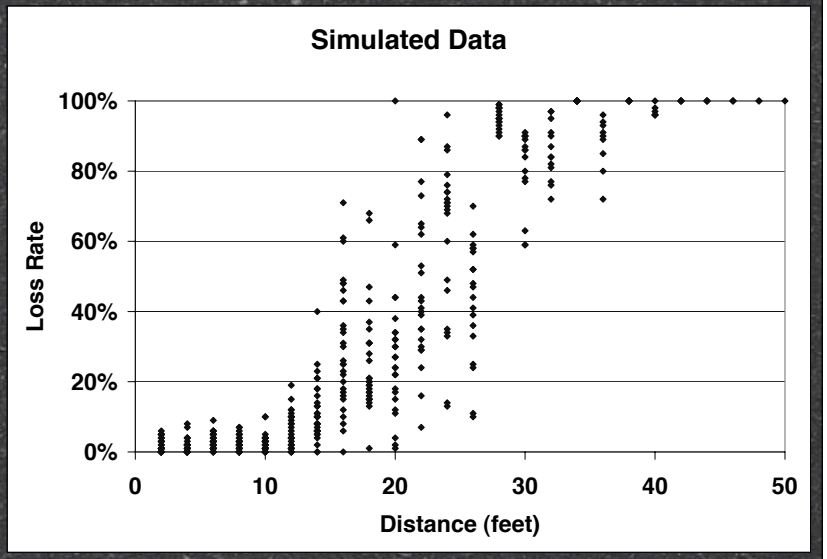


TOSSIM network simulation

- TinyOS uses three network sampling rates
  - 40KBps for data, 20 Kbps for receiving a start symbol, 10 Kbps for sending a start symbol
- in TOSSIM, adjustments to radio bit-rates are made by changing the period between radio clock events.
  - Handled by simulation events
- Exception: spin-loops to synchronize sender signal
  - first loop (0) is ignored
  - second loop (1): when a mote send it, it checks if any mote is in the listening state. If so, it enqueues a radio event for the receiver

Evaluation
1 - Fidelity/radio noise

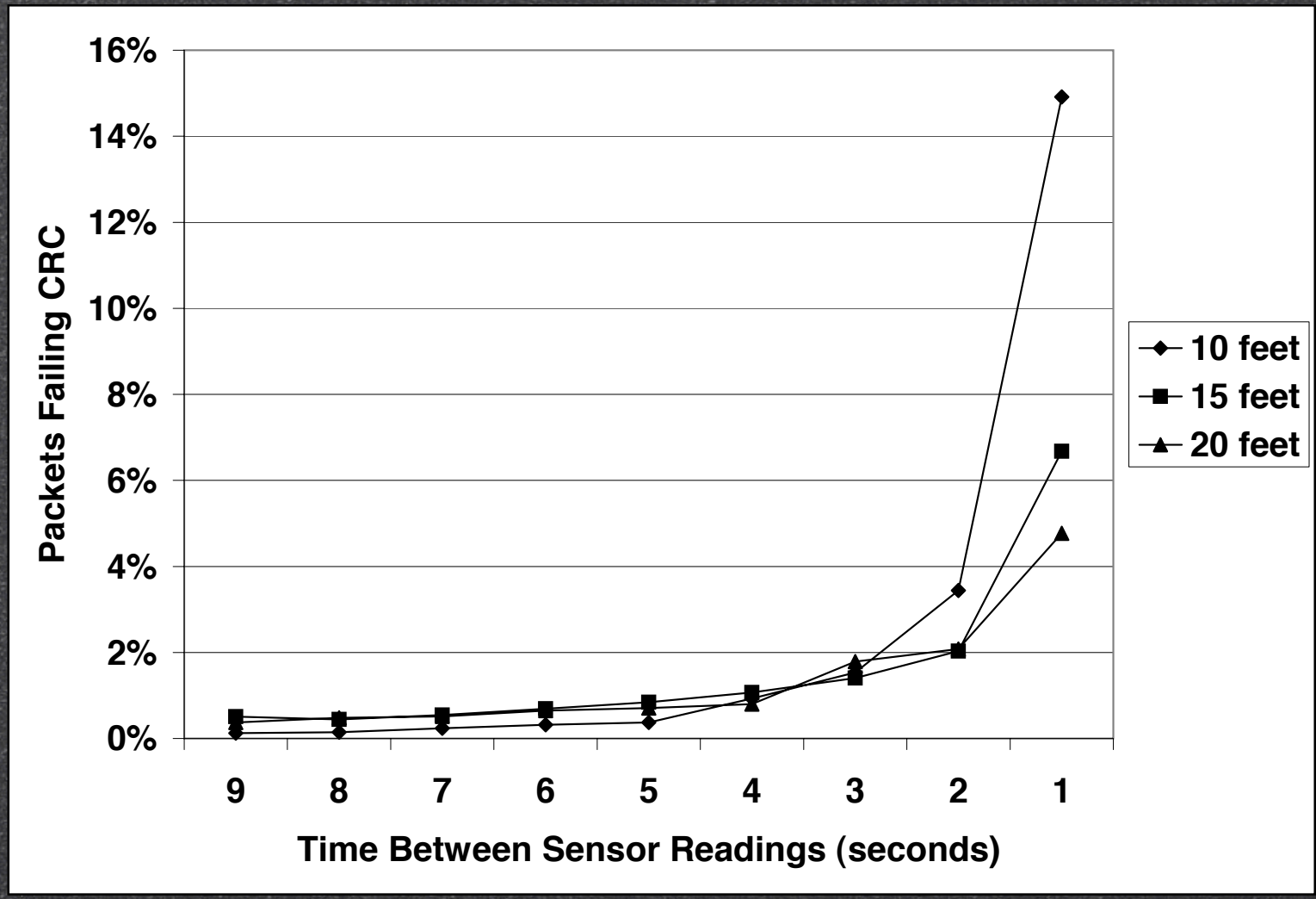




(a) Empirical

(b) Simulated

Fidelity: Packet-level interaction



Practical

tinyos

TinyOS WebSite

- www.tinyos.net
  - or “tinyos I’m feeling lucky in google”
- Tutorial:
  - www.tinyos.net/tinyos-1.x/doc/tutorial

Installation

- Simplest: use the Windows installation.
  - (Sadly, the simplest way of doing it)
  - (Sadly, because it will install Cygwin)
  - (So, a linux-native installation should be better)
- Install under Linux: follow the website.

What you need to simulate

- Java 1.5 JDK
- Cygwin (under windows)
- Native Compilers
  - AVR toolchain for mica family
  - MSP430 toolchain for telos family
  - One of them (AVR is simpler) for Simulation
- nesC compiler (which uses native compiler, so expect one)
- TinyOS source tree from CVS or stable tarball
- Graphviz if you want to make docs of your codes

Recommandations

- Use TinyOS 1.x, TinyOS 2.x is not compatible with 1.x
- Install the TinyOS 1.1.0 with wizard
  - Then optionally, upgrade Cygwin and install TinyOS 1.1.15 from CVS

Try

- After (windows) installation:
  - Launch a terminal (Cygwin icon)
  - TOS is in /opt/tinyos-1.x
  - The GUI of TOSSIM is located into
    - /opt/tinyos-1.x/tools/java/net/tinyos/sim/
    - it is the (java) program called tinyviz
- Go into the Blink application:
  - cd /opt/tinyos-1.x/apps/Blink
  - build the application: make pc
  - run it: ./build/pc/main.exe 1
  - Stop with Control-C (^C)

Try (2)

- By default, TOSSIM displays ALL the events. Most of them are with the radio, that is not important for this test.
- You can configure TOSSIM to select which events you want to display through the environment variable DBG. The expected format is a comma separated list of strings (e.g. “am,led” to display Active-Messages and Leds events). Find the complete list in the article “TOSSIM: A Simulator for TinyOS Networks” or ./build/pc/main.exe --help for other options.
  - Set DBG to led to display only Leds related events.
    - type export DBG=led
- relaunch ./build/pc/main.exe 1
- Stop with ^C again. The trace present leds blinking.

Try (3)

- Enable tinyviz: go into the /opt/tinyos-1.x/tools/java/net/tinyos/sim directory and copy the tinyviz script into /bin (or anywhere in your PATH)
- edit the /bin/tinyviz file and change the BASE variable to equal “/opt/tinyos-1.x/tools/java/net/tinyos/sim”
- return to the application directory (/opt/tinyos-1.x/apps) and try to launch tinyviz. If it doesn’t work, the copy or editing failed.

- Look at the TestTinyViz application into apps. It does random neighbour communication. build it with make pc
- set the DBG variable to usr1,am (to have the debug messages + the Active-Messages events) (export DBG=usr1,am)
- And launch the application inside tinyviz:
  - tinyviz -run build/pc/main.exe 30

Try (4)

- You need first to enable some Tinyviz Plugins (Radio Links, Sent Radio Packets and Debug Messages for this test)
- Then launch or stop/launch (Sim Time should progress)
- Have a look at lesson5 of inline tutorial to know more options of TinyViz

TinyOS Directory

- apps/
  - Applications you can find in the tutorial.
- doc/
  - docs, including the tutorial
- regression/
  - tests
- tools/
  - tools, including tinyviz
- tos/
  - “operating system”

the tos/ directory

- interfaces/
  - All the predefined interfaces. Take a look.
- lib/
  - Platform independant usefull modules, like Counters, LedsIntensity...
- platform/
  - low-level components (hw abstractions depending on the mote)
- sensorboards/
  - low-level components (hw abstractions depending on the sensor)
- system/
  - synthetic hardware components
- types/
  - some .h files

TinyOS 2.0

- Still Beta
- But will replace TinyOS 1.x soon.
- Code is not backward compatible with TinyOS 1.x

Platform/HW abstraction

- a platform is a collection of chips and some glue code that connect them together
  - e.g. mica2 = CC1000 radio chip + ATmega129 µcontroller + AVR compiler
- Hardware Abstraction Architecture
  - Hardware Presentation Layer
    - IO pins or registers as components
  - Hardware Abstraction Layer
    - built on top of HAA, presents the same functionalities with simpler interfaces
  - Hardware Independant Layer
    - Generalization (-> not all the functionalities)

Scheduler

- Tasks are slightly different:
  - Single TQ in 1.x could be full when posting completion of a split-phase operation
  - So, in 2.x, every task has its own reserved slot in TQ, and each task can only posted ONCE. A post fails iff the task has already been posted.
- Applications can replace the scheduler.
  - It is still important to maintain non-preemptiveness.

Booting/Initializing

- The 1.x interface StdControl has been split into two interfaces:
  - Init with only the init() command
  - StdControl with only the start() and stop() commands.
- The boot sequence does not signal start() automatically, it signals Boot.booted() event. The top-level application must wire this interface and call the corresponding start() commands

Virtualization

- TinyOS 2.0 is written in nesC 1.2 which introduce the concept of a ‘generic’ or instantiable component.
  - This provides reusable dataé structures, like queues and bits of vectors
- Many TinyOS 1.x parameterized components are now virtualized.
  - the virtualization does all of the wiring underneath (with a unique instantiator if necessary) automatically

Timers

- Timer interface has been improved.
  - Three types of times: millisecond, 32KHz or one or two high-precision timers that fire asynchronously
  - Components can query the timer for how much time remains before firing
  - Components can start timers in the future

Communication

- message buffer type is now message_t. It is opaque and components cannot reference its fields.
- Instead, all buffer access go through interfaces
  - e.g. instead of looking the msg->dest field, one call AMPacket.destination(msg)
- Send interfaces distinguish between the addressing modes
  - e.g. AM communication has the AMSend interface, to provide the AM destination address; broadcast has the Send (without address) interface
- No TOS_UART_ADDRESS. The component should wire to SerialActiveMessageC

Error Codes

- SUCCESS has been redefined from 1 (1.x) to 0 (2.0).
- This means that tests like
  ```
  if( call X.y() ) { ... }
  ```
  will now do the contrary of what is expected.
- Programmers must change to
  ```
  if( call X.y() == SUCCESS ) { ... }
  ```
- The result_t type is replaced by the error_t type.

Arbitration

- Some resource cannot be virtualized easily.
- For example, a shared bus on a µcontroller.
  - many different systems (sensors, storage, radio) might need to use the bus at the same time
- TinyOS 2.0 introduces the Resource interface which components use to request and acquire shared resources, and arbiters, which provide a policy of arbitrating access between multiple clients.
  - For some abstractions, the arbiter also provides a power management policy

Power Management

- Divided into two parts
  - power state of the µcontroller
    - chip-specific
  - power state of devices
    - arbiters
- Radio has low-power stacks for some chips

Network Protocols

- Components for
  - reliable dissemination
  - reliable collection

Bibliography

- Tinyos:
  - “System Architecture Directions for Networked Sensors”, Jason Hill, Rober t Szewczyk, Alec Woo, Seth Hollar, David Culler, Kristofer Pister, ASPLOS, 2000
- nesC:
  - "The nesC Language: A Holistic Approach to Networked Embedded Systems", David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer and David Culler. Proceedings of Programming Language Design and Implementation (PLDI), 2003
  - “nesC Language Reference Manual”, David Gay, Philip Levis, David Culler and Eric Brewer, 2003, http://www.tinyos.net/tinyos-1.x/doc/nesc/ref.pdf
- TOSSIM:
  - “TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications", Philip Levis and Nelson Lee and Matt Welsh and David Culler, Proceedings of the First ACM Conference on Embedded Networked Sensor Systems, 2003
  - TOSSIM: A Simulator for TinyOS Networks, Philip Levis and Nelso Lee, 2003 http://www.tinyos.net/tinyos-1.x/doc/nido.pdf