TinyOS

TinyOS

- ***"System architecture directions for network sensors"***, Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, Kristofer Pister . ASPLOS 2000, Cambridge, November 2000
- System software for networked sensors
- Tiny Microthreading Operating System: TinyOS
  - Component-based
  - Event-driven
- TinyOS is written in nesC programming language

nesC

- nesC programming language
  - An extension to C
  - Designed for sensor network nodes
- Basic concepts behind nesC
  - Separation of construction and composition
    - Many components, "*wired*"(link) those you want
  - Component provide a set of interfaces
  - Interfaces are bidirectional
    - Command (down call), event (up call)

  - nesC compiler signals the potential data races

Support Multiple Platforms

- Hardware platforms
  - eyesIFXv2, ETH Zurich
    - TI MSP430F1611, Infineon TDA5250
  - Intelmote2, Intel
    - PXA271 XScale Processor, TI (Chipcon) CC2420
  - Mica2, UCB
    - Atmel128, TI (Chipcon) CC1000
  - Mica2dot, UCB
    - Atmel128, TI (Chipcon) CC1000
  - Micaz, UCB
    - Atmel128, TI (Chipcon) CC2420
  - Telosb, UCB
    - MSP430F1611, TI (Chipcon) CC2420
  - Tinynode, EPFL Switzerland
    - MSP430F1611, Semtech radio transceiver XE1205
- Three different microcontrollers, four different radio transceivers and many other peripheral ICs

TinyOS and nesC

Slides from David Gay

- TinyOS is an operating system designed to target limited-resource sensor network nodes
  - TinyOS 0.4, 0.6 (2000-2001)
  - TinyOS 1.0 (2002): first nesC version
  - TinyOS 1.1 (2003): reliability improvements, many new services
  - TinyOS 2.0 (2006): complete rewrite, improved design, portability, reliability and documentation
- TinyOS and its application are implemented in nesC, a C dialect:
  - nesC 1.0 (2002): Component-based programming
  - nesC 1.1 (2003): Concurrency support
  - nesC 1.2 (2005): Generic components, “external” types

Version of TinyOS

- Latest release
  - TinyOS 2.0.2
- History
  - Start with TinyOS 1.x
    - Latest 'CVS snapshot release': 1.1.15
  - Due to some problems, development of TinyOS 1.x suspended
    - "many basic design decisions flawed or too tied to mica-family platforms"
  - TinyOS 2.0 working group formed September 2004
- TinyOS 2.x is not backward compatible
  - Code written on TinyOS 1.x cannot compile on TinyOS 2.x
  - Require minor modification
- TinyOS 1.x is popular
  - Many research group still using it
  - Many protocols available on TinyOS 1.x, but not on TinyOS 2.x
- But, I will talk about TinyOS 2.x in the class
  - **MUCH** better documentations
  - The basic idea is similar, you can still programming TinyOS 1.x

Why Abandon TinyOS 1.x

- The first platform for sensor network is Mica
  - Atmel processor, CC1000 radio
- TinyOS 1.x was designed based on this platform
- Sensor network became popular, more and more platforms available
- Different platforms has different design and architecture
  - Most important, different microcontrollers
  - Wide range of varieties
- It is very difficult to support all the platforms, especially when you didn't consider this issue at the beginning
  - They kept fighting with compatibility issue
- many basic design decisions in TinyOS 1.x make the system unreliable

Slides from David Gay

Other OSes for Mote-class Devices

- SOS https://projects.nesl.ucla.edu/public/sos-2x/
  - C-based, with loadable modules and dynamic memory allocation
  - also event-driven
- Contiki http://www.sics.se/contiki
  - C-based, with lightweight TCP/IP implementations
  - optional preemptive threading
- Mantis http://mantis.cs.colorado.edu
  - C-based, with conventional thread-based programming model
  - semaphores+IPC for inter-thread communication

Why TinyOS is Popular

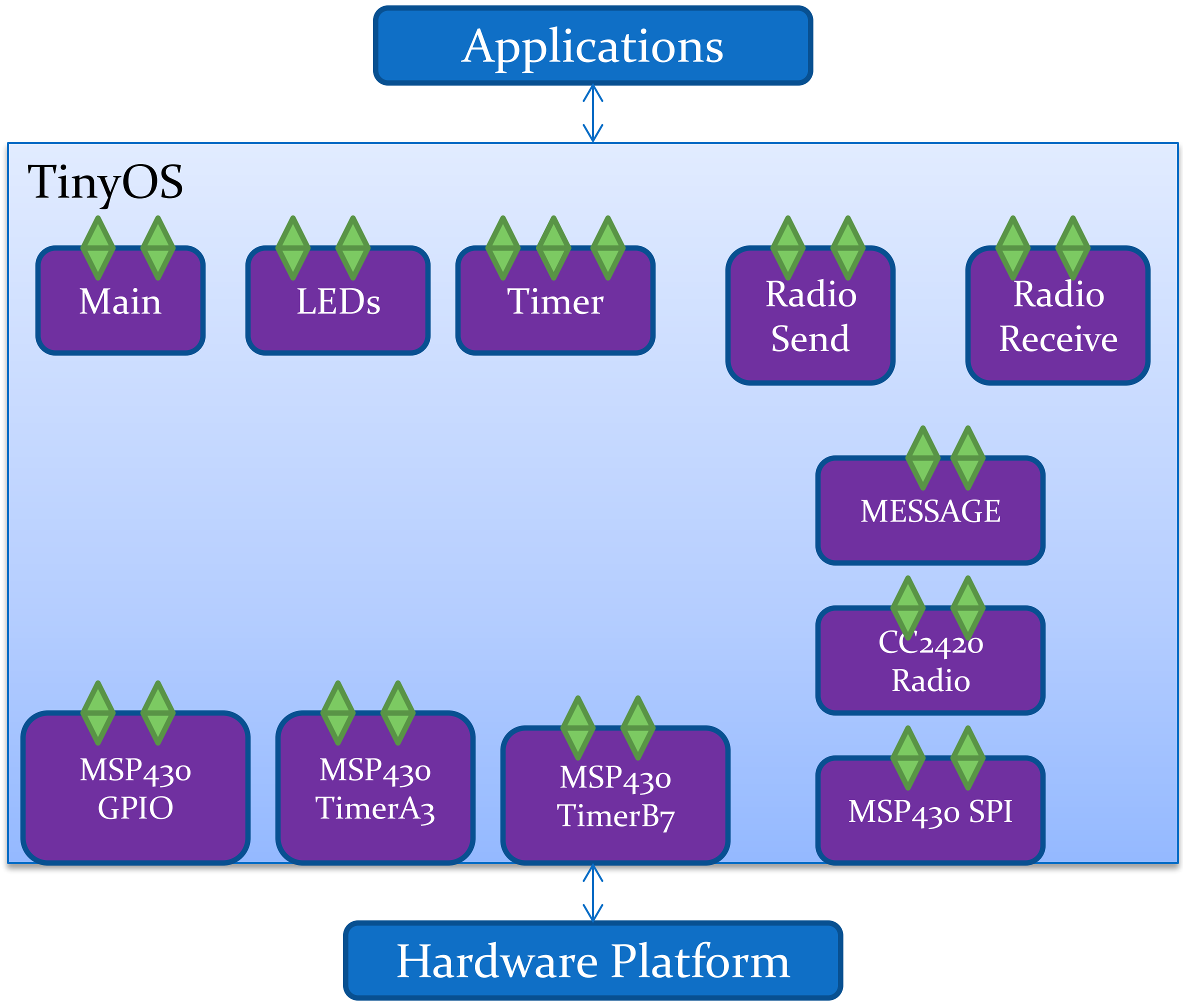
- They are the first sensor network operating system
- **Platforms are commercially available**
- "**Efficient Memory Safety for TinyOS**", Nathan Cooprider, Will Archer, Eric Eide, David Gay and John Regehr *Sensys'07: ACM International Conference on Embedded Networked Sensor Systems, Sydney, Australia, November 2007*
  - nesC is quite similar to C
  - **TinyOS provides a large library of ready-made components**, thus saving much programmer work for common tasks
  - The nesC compiler has a built-in race condition detector that helps developers avoid concurrency bugs
  - TinyOS is designed around a static resource allocation model
- You can program a sensor node without (or with minimum) hardware and microcontroller programming knowledge
  - But, debugging will be a big problem if you don't know what's going on in the lower layer

TinyOS Concept

Components Based

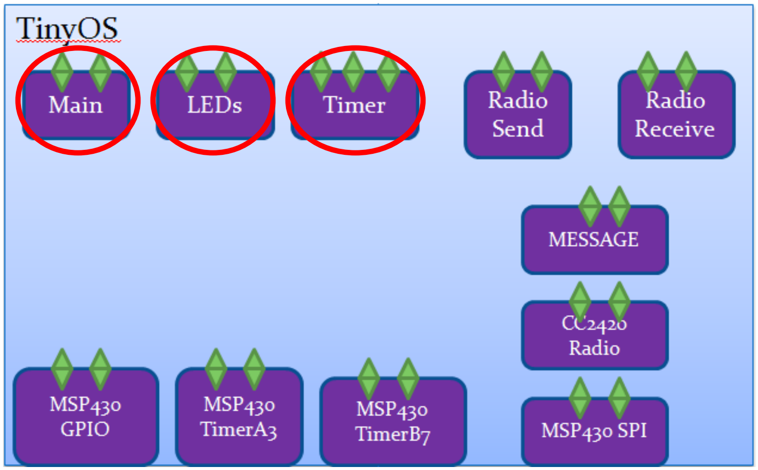
It looks like a library, those components are objects in the library and the interfaces are APIs. But it actually has more functions than just a library

Interfaces	
Components	



An Example: Blink

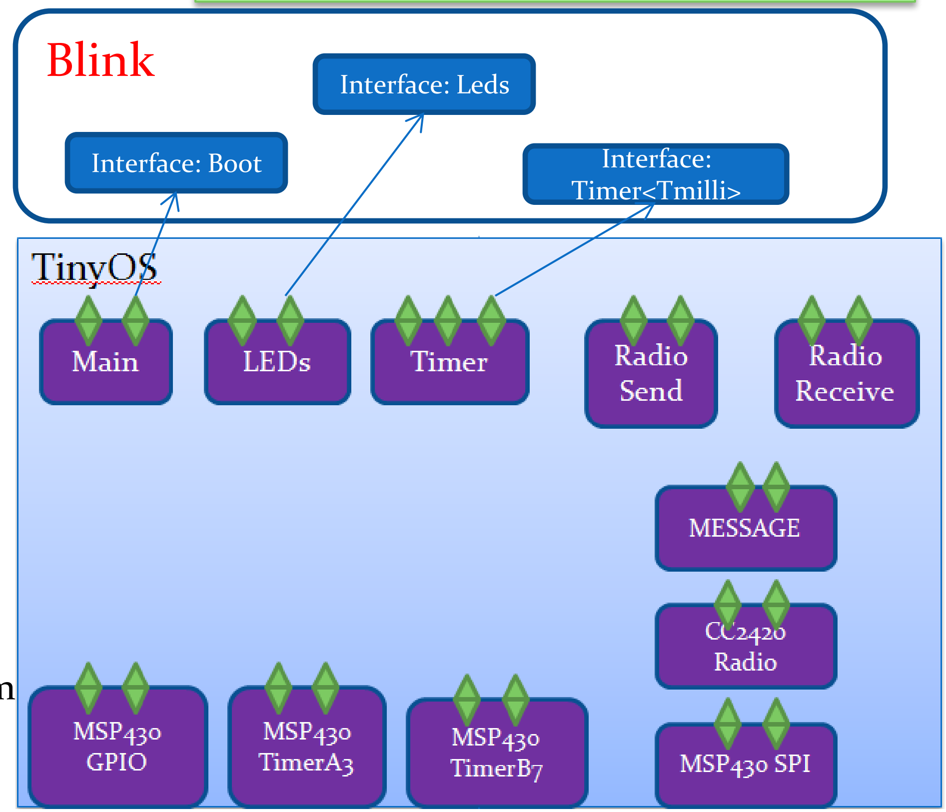
- How to build an application from TinyOS
  - “wired” (link) the components you need
  - Implement the action you intended to do
- Application: Blink
  - Toggle Red LED @ 0.25 Hz
  - Toggle Green LED @ 0.5 Hz
  - Toggle Yellow LED @ 1 Hz
- What components you need?
  - LEDs
  - Timer
  - Main → every program needs a main



Interfaces

Main.Boot: for initialization and boot up
LEDs.Leds : control LEDs (on, off, toggle)
Timer.Timer<Tmilli>: timer in millisecond resolution. you can specific a period (eg. 250), it will signal you when the timer expire.

Components provide interfaces. Application program use these interfaces to control the lower layer components and hardware.



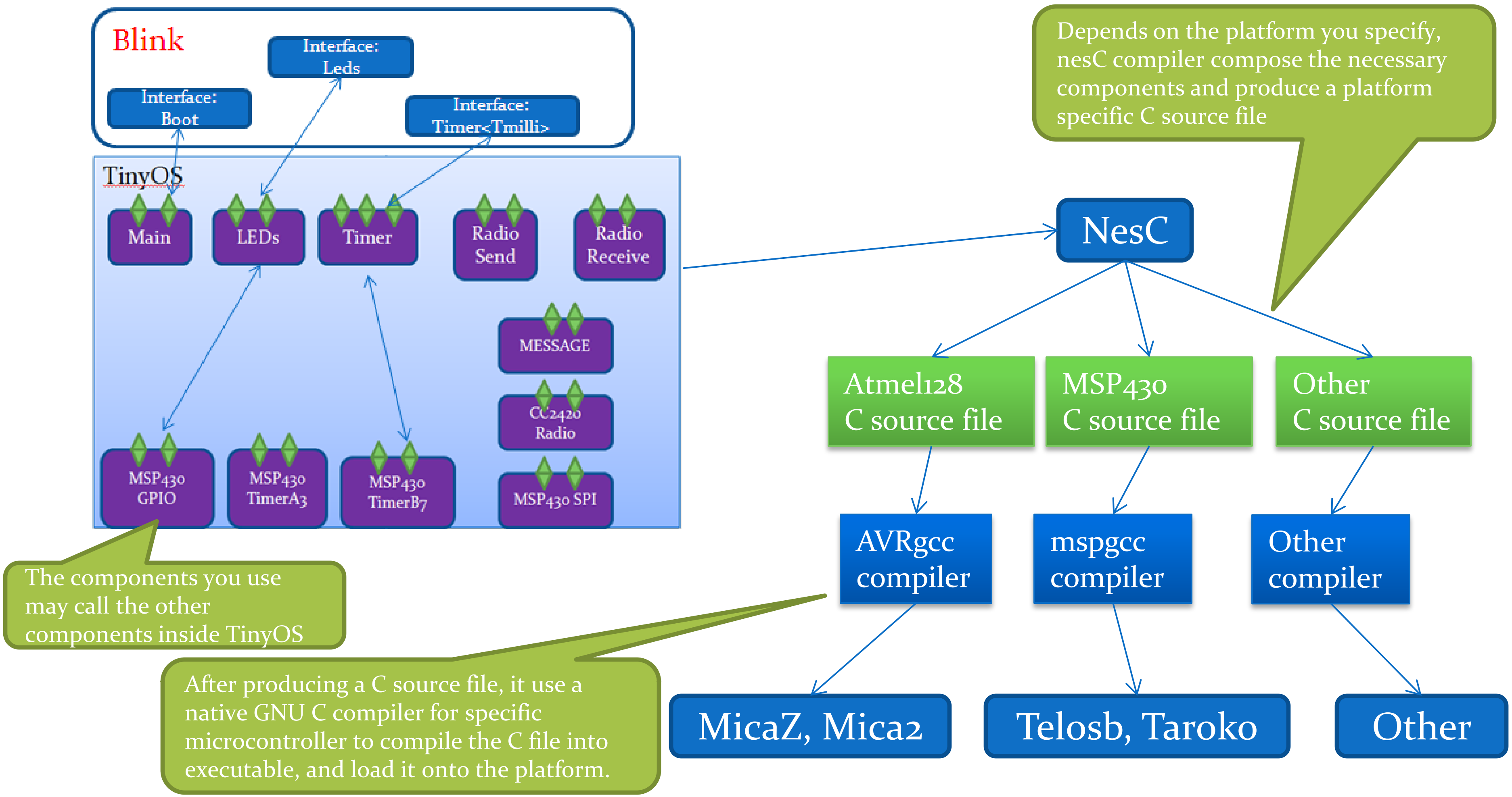
In Blink application, you will have something like this:

```
{
  uses interface Timer<TMilli> as Timer0;
  uses interface Timer<TMilli> as Timer1;
  uses interface Timer<TMilli> as Timer2;
  uses interface Leds;
  uses interface Boot;
}
```

and you implement what you want to do in your program

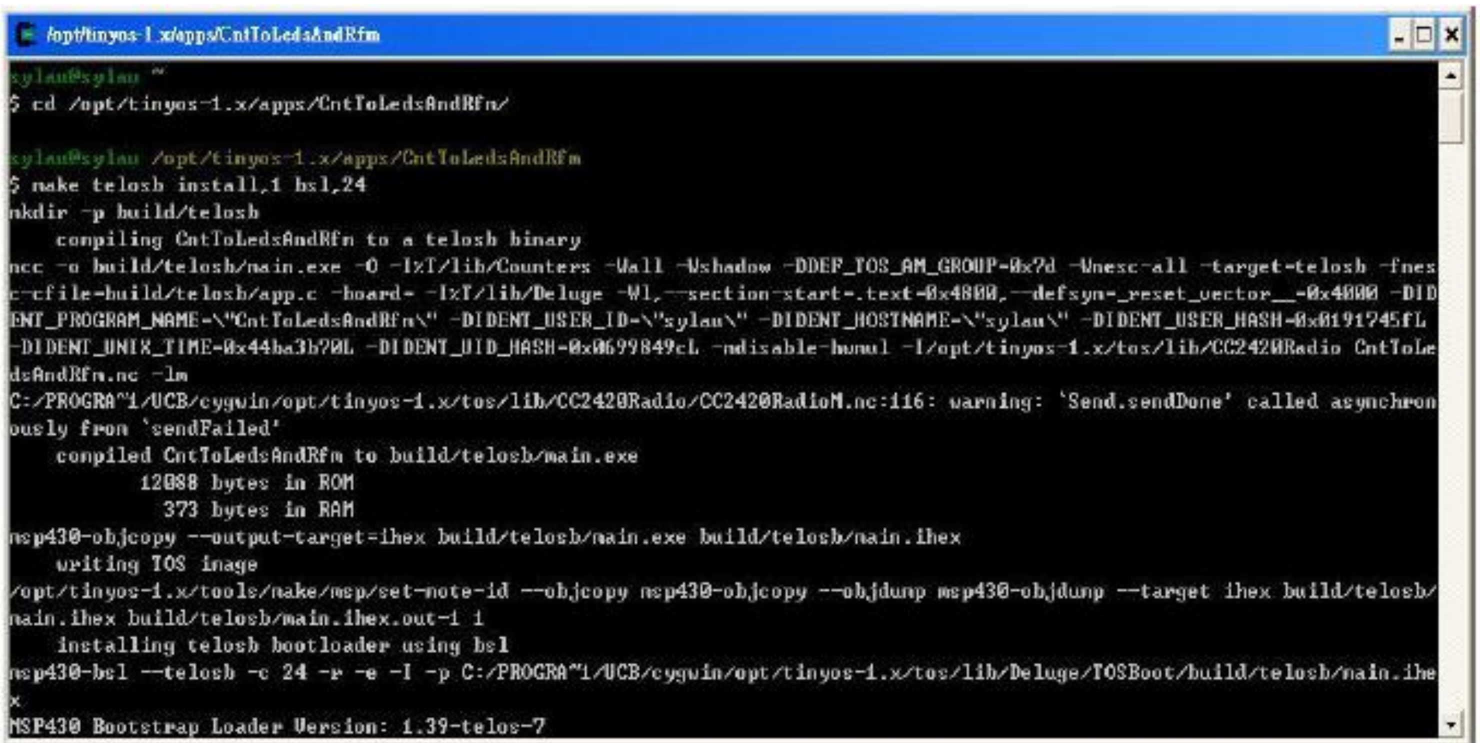
```
{
   when timer fired, toggle LED;
}
```

Composition And Compile



Development Environment

- Command line interface
  - On windows: Cygwin + TinyOS



Installation

- Easiest way
  - One-step Install with a Live CD
  - Use VMware → Linux envoriment
- Easier way
  - Cygwin + TinyOS
  - Install TinyOS 1.1.11 (Windows Installshield)
    - Windows Installshield Wizard for TinyOS CVS Snapshot 1.1.11
  - If you still want TinyOS 1.x
    - Install TinyOS 1.1.15
    - TinyOS CVS Snapshot Installation Instructions
  - Install native tools and TinyOS 2.x
    - http://www.tinyos.net/tinyos-2.x/doc/html/upgrade-tinyos.html
    - Follow the upgrade instructions above

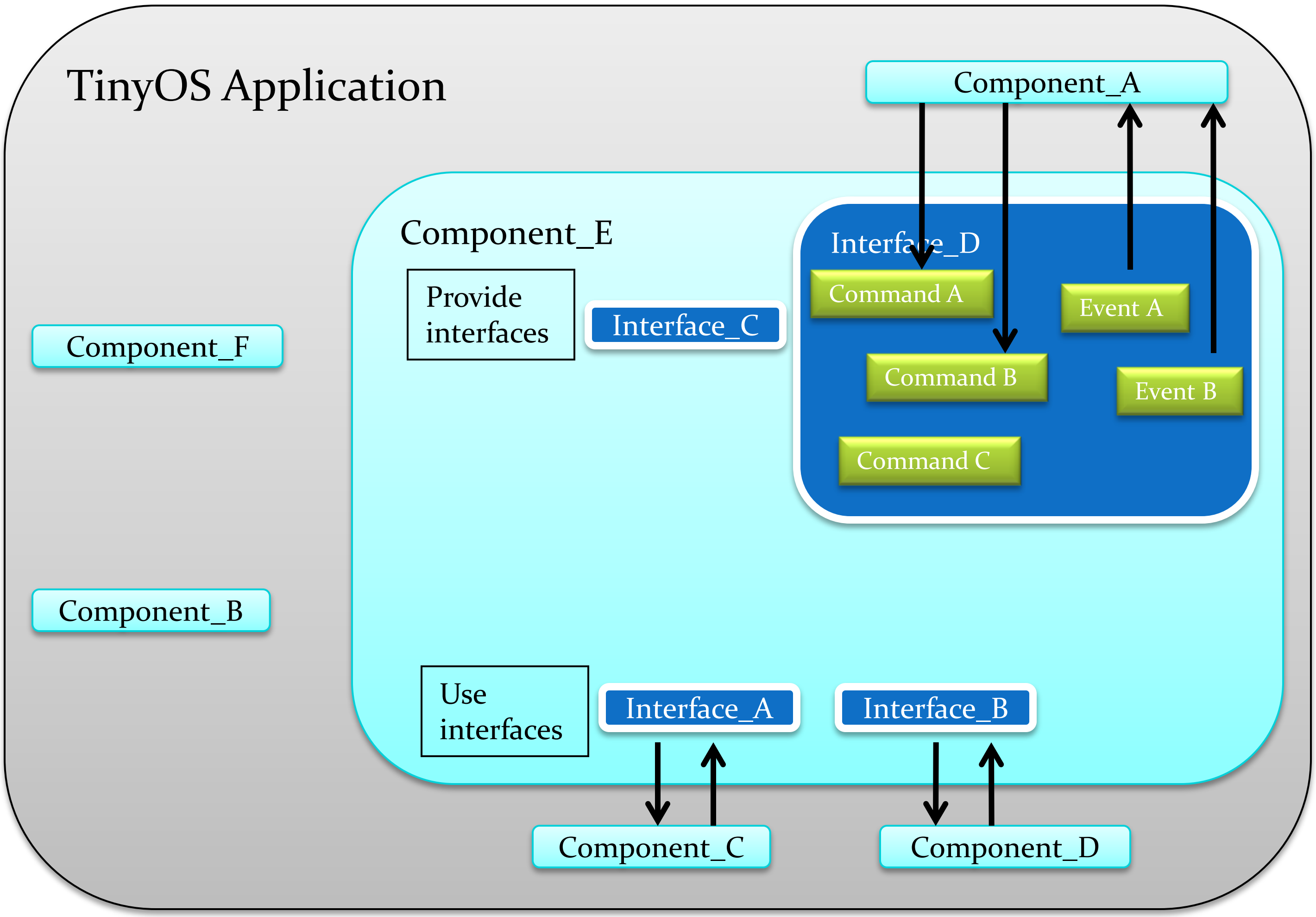
Optional

Upload Program

- make <*platform*> install,<*node id*> bsl,<*COMport – 1*>

```
/opt/tinyos-2.x/apps/Blink

sylau@yoshikun-msbers /opt/tinyos-2.x/apps/Blink
$ make telosb install bsl,7
mkdir -p build/telosb
    compiling BlinkAppC to a telosb binary
ncc -o build/telosb/main.exe -Os -O -mdisable-hwmul -Wall -Wshadow -DDEF_TOS_AM_
GROUP=0x7d -Wnesc-all -target=telosb -fnesc-cfile=build/telosb/app.c -board= -DI
DENT_PROGRAM_NAME=\"BlinkAppC\" -DIDENT_USER_ID=\"sylau\" -DIDENT_HOSTNAME=\"yos
hikun-msbers\" -DIDENT_USER_HASH=0xdade3930L -DIDENT_UNIX_TIME=0x47512bc8L -DIDE
NT_UID_HASH=0xb653e46bL  BlinkAppC.nc -lm
    compiled BlinkAppC to build/telosb/main.exe
            2654 bytes in ROM
              55 bytes in RAM
msp430-objcopy --output-target=ihex build/telosb/main.exe build/telosb/main.ihex
    writing TOS image
cp build/telosb/main.ihex build/telosb/main.ihex.out
    installing telosb binary using bsl
tos-bsl --telosb -c 7 -r -e -I -p build/telosb/main.ihex.out
MSP430 Bootstrap Loader Version: 1.39-telos-8
Mass Erase...
Transmit default password ...
Invoking BSL...
Transmit default password ...
Current bootstrap loader version: 1.61 (Device ID: f16c)
Changing baudrate to 38400 ...
Program ...
2686 bytes programmed.
Reset device ...
rm -f build/telosb/main.exe.out build/telosb/main.ihex.out

sylau@yoshikun-msbers /opt/tinyos-2.x/apps/Blink
$
```

TinyOS Application



1. Application consists one or more components.
2. Components provide and/or use interfaces.
3. Interfaces specify *commands* (down call) and *events* (up call)

Components

- Two types of components: Modules and Configurations
  - Configuration: link components together
  - Module: actual implementation
- Every component has an implementation block
  - In configuration: it define how components link together
  - In module: it allocate state and implement executable logic

Configurations

Configurations are used to assemble other components together, connecting interfaces used by components to interfaces provided by others

This line export the interface provided by module *modA* through *interfA*

Specify the components you will *wire*

```
configuration config {
    provide interface interfA
}
Implementation {
    component modA, configB;

    interfA = modA.interf_a;

    modA.interf_b -> configB.interf_b
}
```

```
module modA {
    provide interface interf_a
    use interface interf_b
}
Implementation
{
    (Your actual code is in here.)
}
```

Modules provide the implementations of one or more interfaces

The -> operator maps between the interfaces of components that a configuration names,
The = operator maps between a configuration's own interfaces and components that it names,

```
configuration(or module) configB {
    provide interface interf_b
    use interface interf_c
}
```

Modules

```
configuration config {
    provide interface interfA
 }
Implementation {
    component modA, configB;
    interfA = modA.interf_a;
    modA.interf_b -> configB.interf_b
}
```

```
module modA {
    provide interface interf_a
    provide interface interf_c
    use interface interf_b
}
Implementation
{
   uint8_t i=0;
   command void interf_a.start() {
       if( interf_b.isSet() ) {
          i++;
          signal interf_a.fired();
       }
   }
   command void interf_a.stop() {
       ...........
   }
   command void interf_c.get() {
       ...........
   }

   event void interf_b.readDone() {
       ............
   }
}
```

A module **MUST** implement
- every command of interfaces it provides, and
- every event of interfaces it uses

It **should(must??)** also signal
- every event of interfaces it provides

It must implements the commands it provides

It can use the command it interf_b provided by configB because there are wired together

It must implements the events it uses

It **should(must??)** signal the events it provides

```
configuration(or module) configB {
    provide interface interf_b
}
```

```
interface interf_b {
  command void isSet();
  event void readDone();
}
```

```
interface interf_c {
  command void get();
}
```

```
interface interf_a {
 command void start();
 command void stop();
 event void fired();
}
```

Another file specify the available commands and events in the interface

Convention

- All nesC files must have a *.nc* extension
  - The nesC compiler requires that the filename match the interface or component name
- File name convention

File Name	File Type
Foo.nc	Interface
Foo.h	Header File
FooC.nc	Public Component
FooP.nc	Private Component

- TinyOS use following type declare
  - You can still use native C type declaration (int, unsigned int, …)
  - But "int" on one platform is 16-bit long, it could be 32-bit long on another platform

	8 bits	16 bits	32 bits	64 bits
signed	int8_t	int16_t	int32_t	int64_t
unsigned	uint8_t	uint16_t	uint32_t	uint64_t

An Example: Blink

Blink

- Application: Blink
  - Toggle Red LED @ 0.25 Hz
  - Toggle Green LED @ 0.5 Hz
  - Toggle Yellow LED @ 1 Hz
- What do you need?
  - Boot up -> initialization
  - Generate three time intervals
  - A method to control LEDs

In The Module

- apps/Blink/BlinkC.nc

```
module BlinkC {
  uses interface Timer<TMilli> as Timer0;
  uses interface Timer<TMilli> as Timer1;
  uses interface Timer<TMilli> as Timer2;
  uses interface Leds;
  uses interface Boot;
}
implementation
{
  // implementation code omitted
}
```

module keyword indicate this is a module file

Interface's name

It's parameter

Alias name

In the module, you use the interfaces you need to build the application

- How to find the available interfaces to use
  - Interface file name: *Foo.nc*
  - /opt/tinyos-2.x/tos/interfaces (demo)
  - Look at the sample applications
    - Most common way

What Components to Wire?

- You know the interfaces you want to use
  - But which components provide these interfaces?
- How to find the component?
  - Again, Look at the sample applications
  - Read TinyOS 2.x documentation
  - Search in the /opt/tinyos-2.x/tos directory (demo)
    - grep –r "provides interface *(interface name)*" *
      - /opt/tinyos-2.x/tos/system/LedsC.nc
      - /opt/tinyos-2.x/tos/system/TimerMilliC.nc
      - /opt/tinyos-2.x/tos/system/MainC.nc

Blink: Configuration

- Every nesC application start by a top level configuration
  - *wire* the interfaces of the components you want to use
- You already know what components to reference
- In configuration of Blink
  - apps/Blink/BlinkAppC.nc

Configuration keyword indicate this is a configuration file

```
configuration BlinkAppC {
}
implementation {
  components MainC, BlinkC, LedsC;
  components new TimerMilliC() as Timer0;
  components new TimerMilliC() as Timer1;
  components new TimerMilliC() as Timer2;

  BlinkC -> MainC.Boot;
  BlinkC.Timer0 -> Timer0;
  BlinkC.Timer1 -> Timer1;
  BlinkC.Timer2 -> Timer2;
  BlinkC.Leds -> LedsC;
}
```

In the configuration, you specific the components you want to reference. This configuration references 6 components

How to Wire

- A full wiring is A.a->B.b, which means "interface a of component A wires to interface b of component B.“
- Naming the interface is important when a component uses or provides multiple instances of the same interface. For example, BlinkC uses three instances of Timer: Timer0, Timer1 and Timer2
- When a component only has one instance of an interface, you can elide the interface name

```
configuration BlinkAppC {
}
implementation {
  components MainC, BlinkC, LedsC;
  components new TimerMilliC() as Timer0;
  components new TimerMilliC() as Timer1;
  components new TimerMilliC() as Timer2;

  BlinkC -> MainC.Boot;
  BlinkC.Timer0 -> Timer0;
  BlinkC.Timer1 -> Timer1;
  BlinkC.Timer2 -> Timer2;
  BlinkC.Leds -> LedsC;
}
```

```
configuration LedsC {
  provides interface Leds;
}
implementation {
```

```
generic configuration TimerMilliC() {
  provides interface Timer<TMilli>;
}
implementation {
```

```
configuration MainC {
  provides interface Boot;
  uses interface Init as SoftwareInit;
}
```

BlinkC component has one instance of *Boot* and *Leds* interface, but it has three instances of *Timer* interface. So, it can elide the interface name *Boot* and *Leds*, but cannot elide *Timer*.

BlinkC.Boot -> MainC.Boot;
BlinkC.Timer0 -> Timer0.Timer;
BlinkC.Timer1 -> Timer1.Timer;
BlinkC.Timer2 -> Timer2.Timer;
BlinkC.Leds -> LedsC.Leds;

Events And Commands

- What events and commands inside a interface?
  - Search the interface file
    - Command: # locate *interface_name*.nc
      - /opt/tinyos-2.x/tos/lib/timer/Timer.nc
      - /opt/tinyos-2.x/tos/interfaces/Leds.nc
      - /opt/tinyos-2.x/tos/interfaces/Boot.nc
    - Take a look at these files (demo)
- Command
  - Available functions you can use
- Event
  - You must implement a handler for every event in the interface you use

Implementation

This module didn't provide interface, it use five interfaces

A module **MUST** implement
- every command of interfaces it provides, and
- every event of interfaces it uses

Timer0.startPeriodic(250)
= BlinkC.Timer<TMilli>.startPeriodic(250)
= Timer0.Timer<TMilli>.startPeriodic(250)
= TimerMilliC.Timer<TMilli>.startPeriodic(250)

```
module BlinkC {
  uses interface Timer<TMilli> as Timer0;
  uses interface Timer<TMilli> as Timer1;
  uses interface Timer<TMilli> as Timer2;
  uses interface Leds;
  uses interface Boot;
}
implementation
{
  event void Boot.booted()
  {
    call Timer0.startPeriodic( 250 );
    call Timer1.startPeriodic( 500 );
    call Timer2.startPeriodic( 1000 );
  }

  event void Timer0.fired()
  {
    call Leds.led0Toggle();
  }

  event void Timer1.fired()
  {
    call Leds.led1Toggle();
  }

  event void Timer2.fired()
  {
    call Leds.led2Toggle();
  }
}
```

In module, Timer0 is an interface. In configuration, Timer0 is a component

What it says here is pretty straight forward. After the system booted, start the timer periodically. When the timer fired, toggle LED.

```
implementation {
  components MainC, BlinkC, LedsC;
  components new TimerMilliC() as Timer0;
  components new TimerMilliC() as Timer1;
  components new TimerMilliC() as Timer2;

  BlinkC -> MainC.Boot;
  BlinkC.Timer0 -> Timer0;
  BlinkC.Timer1 -> Timer1;
  BlinkC.Timer2 -> Timer2;
  BlinkC.Leds -> LedsC;
}
```

Dig Into The Lowest Layer

- We use the Leds interface to find out how it is actually implemented in the lowest layer
  - Trace the file down to the lowest layer
  - configuration links the components
  - module details the implemention
  - Interface
    - **MUST** have some module to implement the interface

Start With BlinkC.nc

```
module BlinkC {
  uses interface Timer<TMilli> as Timer0;
  uses interface Timer<TMilli> as Timer1;
  uses interface Timer<TMilli> as Timer2;
  uses interface Leds;
  uses interface Boot;
}
implementation
{
  event void Boot.booted()
  {
    call Timer0.startPeriodic( 250 );
    call Timer1.startPeriodic( 500 );
    call Timer2.startPeriodic( 1000 );
  }

  event void Timer0.fired()
  {
    call Leds.led0Toggle();
  }

  event void Timer1.fired()
  {
    call Leds.led1Toggle();
  }

  event void Timer2.fired()
  {
    call Leds.led2Toggle();
  }
}
```

BlinkC.Leds wire to *LedsC.Leds*, so we check *LecsC.nc*

```
implementation {
  components MainC, BlinkC, LedsC;
  components new TimerMilliC() as Timer0;
  components new TimerMilliC() as Timer1;
  components new TimerMilliC() as Timer2;

  BlinkC -> MainC.Boot;
  BlinkC.Timer0 -> Timer0;
  BlinkC.Timer1 -> Timer1;
  BlinkC.Timer2 -> Timer2;
  BlinkC.Leds -> LedsC;
}
```

LedsC.nc

```
module LedsP {
   provides {
      interface Init;
      interface Leds;
   }
   uses {
      interface GeneralIO as Led0;
      interface GeneralIO as Led1;
      interface GeneralIO as Led2;
   }
}
implementation {
   command error_t Init.init() {
      atomic {
         dbg("Init", "LEDS: initialized.\n");
         call Led0.makeOutput();
         call Led1.makeOutput();
         call Led2.makeOutput();
         call Led0.set();
         call Led1.set();
         call Led2.set();
      }
      return SUCCESS;
   }

   /* Note: the call is inside the dbg, as it'
       location, so can't be deadcode eliminate
#define DBGLED(n) \
   dbg("LedsC", "LEDS: Led" #n " %s.\n", call

   async command void Leds.led0On() {
      call Led0.clr();
      DBGLED(0);
   }

   async command void Leds.led0Off() {
      call Led0.set();
      DBGLED(0);
   }

   async command void Leds.led0Toggle() {
      call Led0.toggle();
      DBGLED(0);
   }
```

```
configuration LedsC {
  provides interface Leds;
}
implementation {
  components LedsP, PlatformLedsC;

  Leds = LedsP;

  LedsP.Init <- PlatformLedsC.Init;
  LedsP.Led0 -> PlatformLedsC.Led0;
  LedsP.Led1 -> PlatformLedsC.Led1;
  LedsP.Led2 -> PlatformLedsC.Led2;
}
```

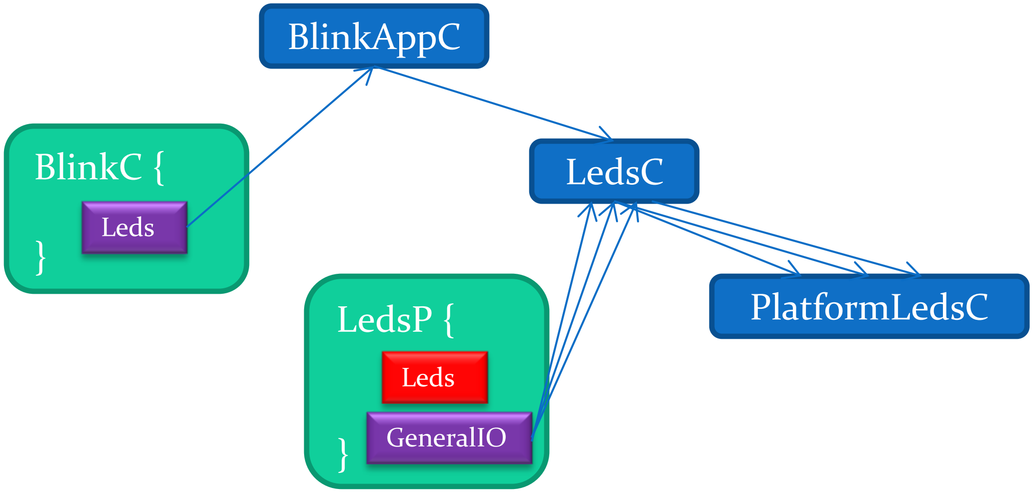
In *LedsC*, it export the interface from *LedsP*. And it wire the interface (*GeneralIO*) used by *LedsP* to *PlatformLedsC*

Interface *Leds* is implemented by *LedsP*. It use three instances of *GeneralIO* to implement these commands.

Every command in the *Leds* interface must be implemented by *LedsP* (demo)

Component Graph

Name	color
Configuration	
Module	
Used interface	
Implemented interface	



Now we know interface Leds is implemented by module LedsP, and we have a new interface GeneralIO, which the LedsP use.

PlatformLedsC.nc

```
configuration PlatformLedsC {
  provides interface GeneralIO as Led0;
  provides interface GeneralIO as Led1;
  provides interface GeneralIO as Led2;
  uses interface Init;
}
implementation
{
  components
      HplMsp430GeneralIOC as GeneralIOC
    , new Msp430GpioC() as Led0Impl
    , new Msp430GpioC() as Led1Impl
    , new Msp430GpioC() as Led2Impl
    ;
  components PlatformP;

  Init = PlatformP.LedsInit;

  Led0 = Led0Impl;
  Led0Impl -> GeneralIOC.Port54;

  Led1 = Led1Impl;
  Led1Impl -> GeneralIOC.Port55;

  Led2 = Led2Impl;
  Led2Impl -> GeneralIOC.Port56;

}
```

Msp430GpioC is a module. It implement the commands in interface GeneralIO. It use interfaces HplMsp430GeneralIO to implement these commands. (demo)

HplMsp430GeneralIOC provide a bunch of interfaces, three of them (Port54, Port55, Port56) is used by Msp430GpioC (demo)

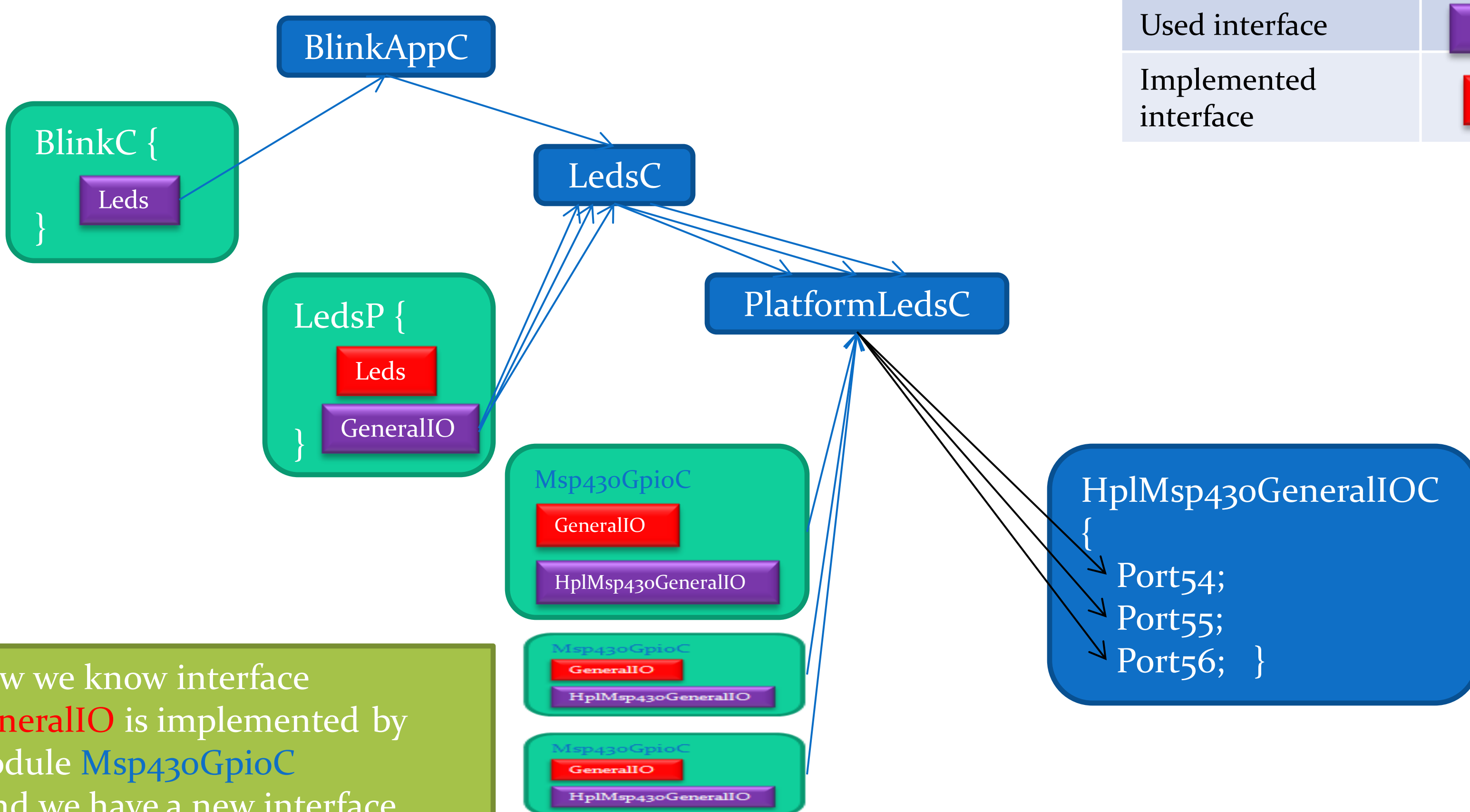
Msp430GpioC.nc

```
generic module Msp430GpioC() {
  provides interface GeneralIO;
  uses interface HplMsp430GeneralIO as HplGeneralIO;
}
implementation {

  async command void GeneralIO.set() { call HplGeneralIO.set(); }
  async command void GeneralIO.clr() { call HplGeneralIO.clr(); }
  async command void GeneralIO.toggle() { call HplGeneralIO.toggle(); }
  async command bool GeneralIO.get() { return call HplGeneralIO.get(); }
  async command void GeneralIO.makeInput() { call HplGeneralIO.makeInput(); }
  async command bool GeneralIO.isInput() { return call HplGeneralIO.isInput(); }
  async command void GeneralIO.makeOutput() { call HplGeneralIO.makeOutput(); }
  async command bool GeneralIO.isOutput() { return call HplGeneralIO.isOutput(); }

}
```

It use interface HplMsp430GeneralIO to implement commands in interface GeneralIO (demo)

Component Graph

Name	color
Configuration	
Module	
Used interface	
Implemented interface	



Now we know interface GeneralIO is implemented by module Msp430GpioC
, and we have a new interface HplMsp430GeneralIO
, which the Msp430GpioC use.

HplMsp430GeneralIOC.nc

```
configuration HplMsp430GeneralIOC
{
#ifdef __msp430_have_port5
   provides interface HplMsp430GeneralIO as Port50;
   provides interface HplMsp430GeneralIO as Port51;
   provides interface HplMsp430GeneralIO as Port52;
   provides interface HplMsp430GeneralIO as Port53;
   provides interface HplMsp430GeneralIO as Port54;
   provides interface HplMsp430GeneralIO as Port55;
   provides interface HplMsp430GeneralIO as Port56;
   provides interface HplMsp430GeneralIO as Port57;
#endif
implementation
{
  components
#ifdef __msp430_have_port5
    new HplMsp430GeneralIOP(P5IN_, P5OUT_, P5DIR_, P5SEL_, 0) as P50,
    new HplMsp430GeneralIOP(P5IN_, P5OUT_, P5DIR_, P5SEL_, 1) as P51,
    new HplMsp430GeneralIOP(P5IN_, P5OUT_, P5DIR_, P5SEL_, 2) as P52,
    new HplMsp430GeneralIOP(P5IN_, P5OUT_, P5DIR_, P5SEL_, 3) as P53,
    new HplMsp430GeneralIOP(P5IN_, P5OUT_, P5DIR_, P5SEL_, 4) as P54,
    new HplMsp430GeneralIOP(P5IN_, P5OUT_, P5DIR_, P5SEL_, 5) as P55,
    new HplMsp430GeneralIOP(P5IN_, P5OUT_, P5DIR_, P5SEL_, 6) as P56,
    new HplMsp430GeneralIOP(P5IN_, P5OUT_, P5DIR_, P5SEL_, 7) as P57,
#endif
```

In HplMsp430GeneralIOC, it export the interface from HplMsp430GeneralIOP.

```
#ifdef __msp430_have_port5
  Port50 = P50;
  Port51 = P51;
  Port52 = P52;
  Port53 = P53;
  Port54 = P54;
  Port55 = P55;
  Port56 = P56;
  Port57 = P57;
#endif
```

Which means that Port54 = HplMsp430GeneralIOP(P5IN_, P5OUT_, P5DIR_, P5SEL_, 4).

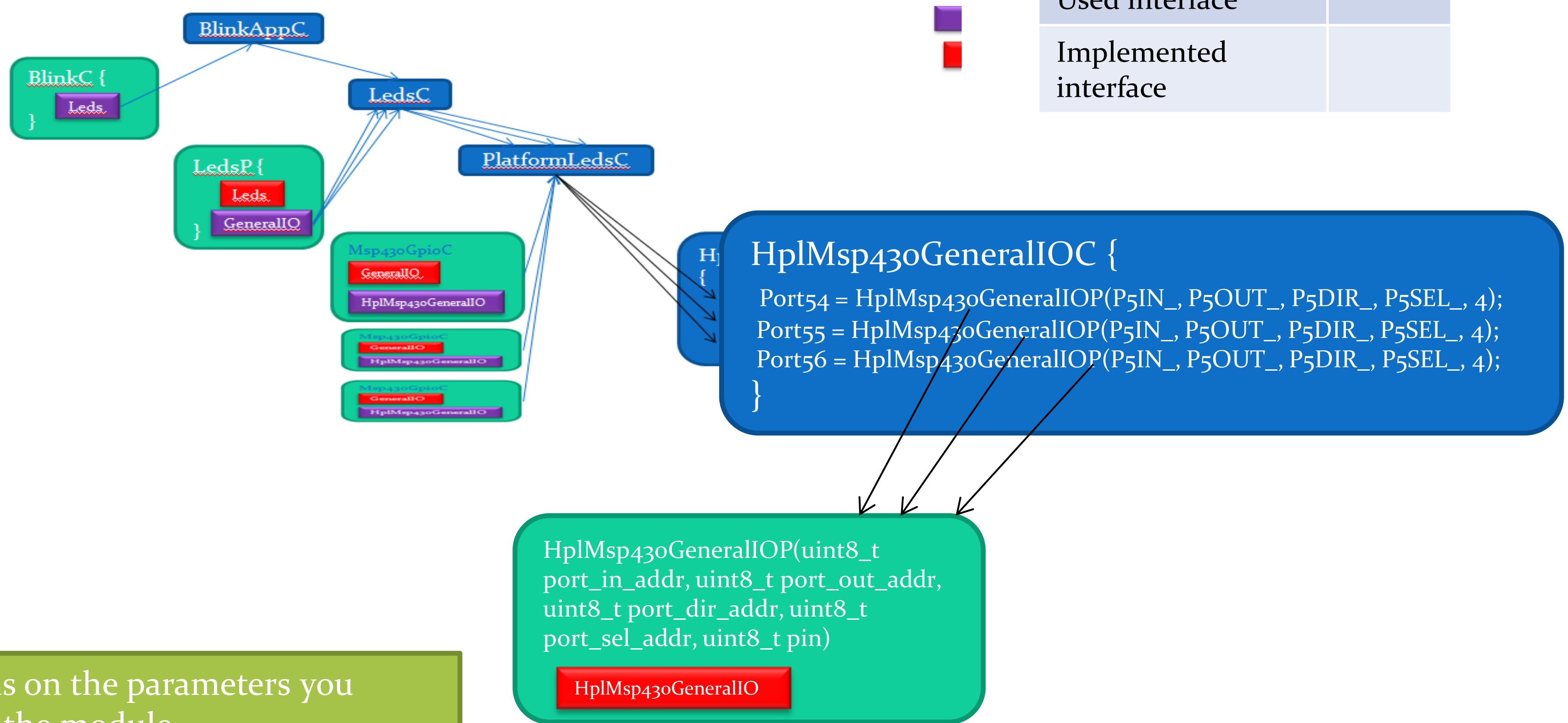
HplMsp430GeneralIOP.nc

```
generic module HplMsp430GeneralIOP(
                                uint8_t port_in_addr,
                                uint8_t port_out_addr,
                                uint8_t port_dir_addr,
                                uint8_t port_sel_addr,
                                uint8_t pin
                                )
{
  provides interface HplMsp430GeneralIO as IO;
}
implementation
{
  #define PORTxIN (*(volatile TYPE_PORT_IN*)port_in_addr)
  #define PORTx (*(volatile TYPE_PORT_OUT*)port_out_addr)
  #define PORTxDIR (*(volatile TYPE_PORT_DIR*)port_dir_addr)
  #define PORTxSEL (*(volatile TYPE_PORT_SEL*)port_sel_addr)

  async command void IO.set() { atomic PORTx |= (0x01 << pin); }
  async command void IO.clr() { atomic PORTx &= ~(0x01 << pin); }
  async command void IO.toggle() { atomic PORTx ^= (0x01 << pin); }
  async command uint8_t IO.getRaw() { return PORTxIN & (0x01 << pin); }
  async command bool IO.get() { return (call IO.getRaw() != 0); }
  async command void IO.makeInput() { atomic PORTxDIR &= ~(0x01 << pin); }
  async command bool IO.isInput() { return (PORTxDIR & (0x01 << pin)) == 0; }
  async command void IO.makeOutput() { atomic PORTxDIR |= (0x01 << pin); }
  async command bool IO.isOutput() { return (PORTxDIR & (0x01 << pin)) != 0; }
  async command void IO.selectModuleFunc() { atomic PORTxSEL |= (0x01 << pin); }
  async command bool IO.isModuleFunc() { return (PORTxSEL & (0x01<<pin)) != 0; }
  async command void IO.selectIOFunc() { atomic PORTxSEL &= ~(0x01 << pin); }
  async command bool IO.isIOFunc() { return (PORTxSEL & (0x01<<pin)) == 0; }
}
```

Port54.toggle()
= HplMsp430GeneralIOP(P5IN_, P5OUT_, P5DIR_, P5SEL_, 4).toggle()
= "P5OUT_ ^= (0x01 << 4);"

Component Graph

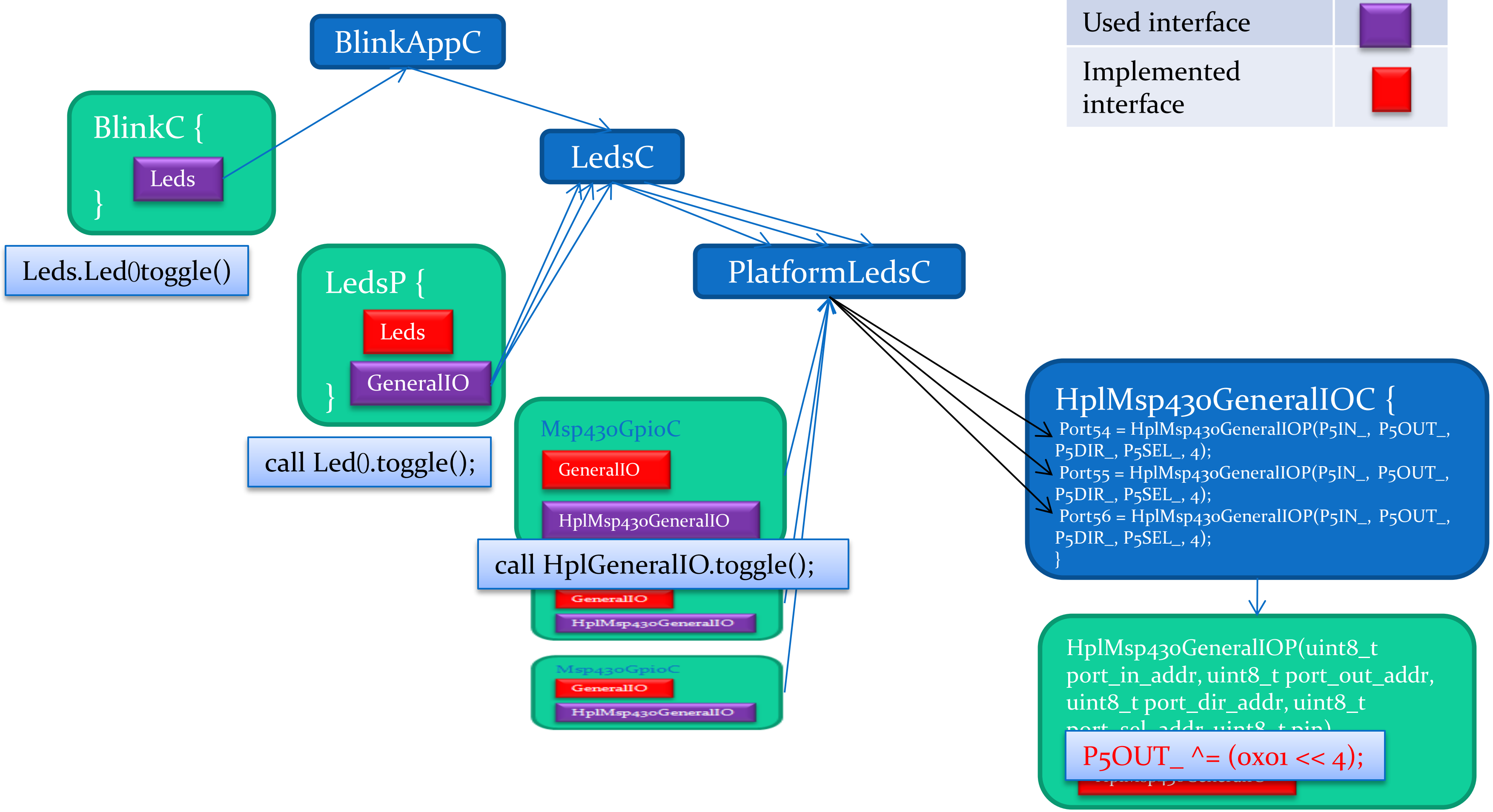
Name	color
Configuration	
Module	
Used interface	
Implemented interface	



Depends on the parameters you specify, the module HplMsp430GeneralIOP implements the interface HplMsp430GeneralIO

Component Graph

Name	color
Configuration	(blue)
Module	(green)
Used interface	(purple)
Implemented interface	(red)



Hardware Abstraction

- Toggle LED is such a simple operation, why so many call?
  - Hardware abstraction
- Hardware abstraction
  - Hide the hardware detail
    - So you can program motes without hardware knowledge
  - Improve reusability and portability
- But what about performance and optimization?

Hardware Abstraction Architecture

- Borrowed slides from TinyOS website
  - http://www.tinyos.net/ttx-02-2005/tinyos2/ttx2005-haa.ppt
  - By Vlado Handziski
  - ***Flexible Hardware Abstraction for Wireless Sensor Networks***, V. Handziski, J.Polastre, J.H.Hauer, C.Sharp, A.Wolisz and D.Culler, in *Proceedings of the 2nd European Workshop on Wireless Sensor Networks (EWSN 2005)*, Istanbul, Turkey, 2005
  - I added some comments

Boot Up

Blink In C

- If you wrote a Blink application in C

```
main() {
    setting GPIO registers (for LEDs)
    setting Timer registers

    start Timer

    for(;;) {
    }
}

Timer ISR {
     toggle LEDs
}
```

- What about the main() in TinyOS

Boot Sequence

BlinkC.nc

```
implementation
{
  event void Boot.booted()
  {
    call Timer0.startPeriodic( 250 );
    call Timer1.startPeriodic( 500 );
    call Timer2.startPeriodic( 1000 );
  }
```

- In the Blink application, there is a interface Boot
  - This interface has a event booted
  - If you trace down the components, you will find that this interface is actually implemented by a module RealMainP
  - This is where the *main()* stay
    - So every application requires a interface Boot,
    - And wire it to the MainC.Boot

RealMainP.nc

- In the RealMainP.nc

```
module RealMainP {
  provides interface Booted;
  uses {
    interface Scheduler;
    interface Init as PlatformInit;
    interface Init as SoftwareInit;
  }
}
```

The TinyOS boot sequence has four steps:
1. Task scheduler initialization
2. Component initialization
3. Signal that the boot process has completed
4. Run the task scheduler

```
implementation {
  int main() __attribute__ ((C, spontaneous)) {
    atomic {
      call Scheduler.init();                      // Step 1
      call PlatformInit.init();                   // Step 2
      while (call Scheduler.runNextTask());
      call SoftwareInit.init();                   // Step 2
      while (call Scheduler.runNextTask());
    }
    __nesc_enable_interrupt();
    signal Boot.booted();                         // Step 3
    call Scheduler.taskLoop();                    // Step 4
    return -1;
  }
  default command error_t PlatformInit.init() { return SUCCESS; }
  default command error_t SoftwareInit.init() { return SUCCESS; }
  default event void Boot.booted() { }
}
```

This boot sequence is different from TinyOS 1.x. If you are using TinyOS 1.x, check “TEP 106: Schedulers and Tasks” and “TEP 107: Boot Sequence” for more detail.

Atomic

This section of codes runs to the end. It can't be preempted. Basically it is implemented by disable global interrupt.

```
implementation {
  int main() __attribute__ ((C, spontaneous)) {
    atomic {
      call Scheduler.init();
      call PlatformInit.init();
      while (call Scheduler.runNextTask());
      call SoftwareInit.init();
      while (call Scheduler.runNextTask());
    }
    __nesc_enable_interrupt();
    signal Boot.booted();
    call Scheduler.taskLoop();
    return -1;
  }
  default command error_t PlatformInit.init() { return SUCCESS; }
  default command error_t SoftwareInit.init() { return SUCCESS; }
  default event void Boot.booted() { }
}
```

- Use a atomic section to protect you code
  - It disable global interrupt, make it short

MainC.nc

```
configuration MainC {
  provides interface Boot;
  uses interface Init as SoftwareInit;
}
implementation {
  components PlatformC, RealMainP, TinySchedulerC;

  RealMainP.Scheduler -> TinySchedulerC;
  RealMainP.PlatformInit -> PlatformC;

  // Export the SoftwareInit and Booted for applications
  SoftwareInit = RealMainP.SoftwareInit;
  Boot = RealMainP;
}
```

Export these two interfaces to applications

Automatically wiring these two to the system's scheduler and platform initialization sequence. Hide them from applications

```
implementation {
  int main() __attribute__ ((C, spontaneous)) {
    atomic {
      call Scheduler.init();
      call PlatformInit.init();
      while (call Scheduler.runNextTask());
      call SoftwareInit.init();
      while (call Scheduler.runNextTask());
    }
    __nesc_enable_interrupt();
    signal Boot.booted();
    call Scheduler.taskLoop();
    return -1;
  }
  default command error_t PlatformInit.init() { return SUCCESS; }
  default command error_t SoftwareInit.init() { return SUCCESS; }
  default event void Boot.booted() { }
}
```

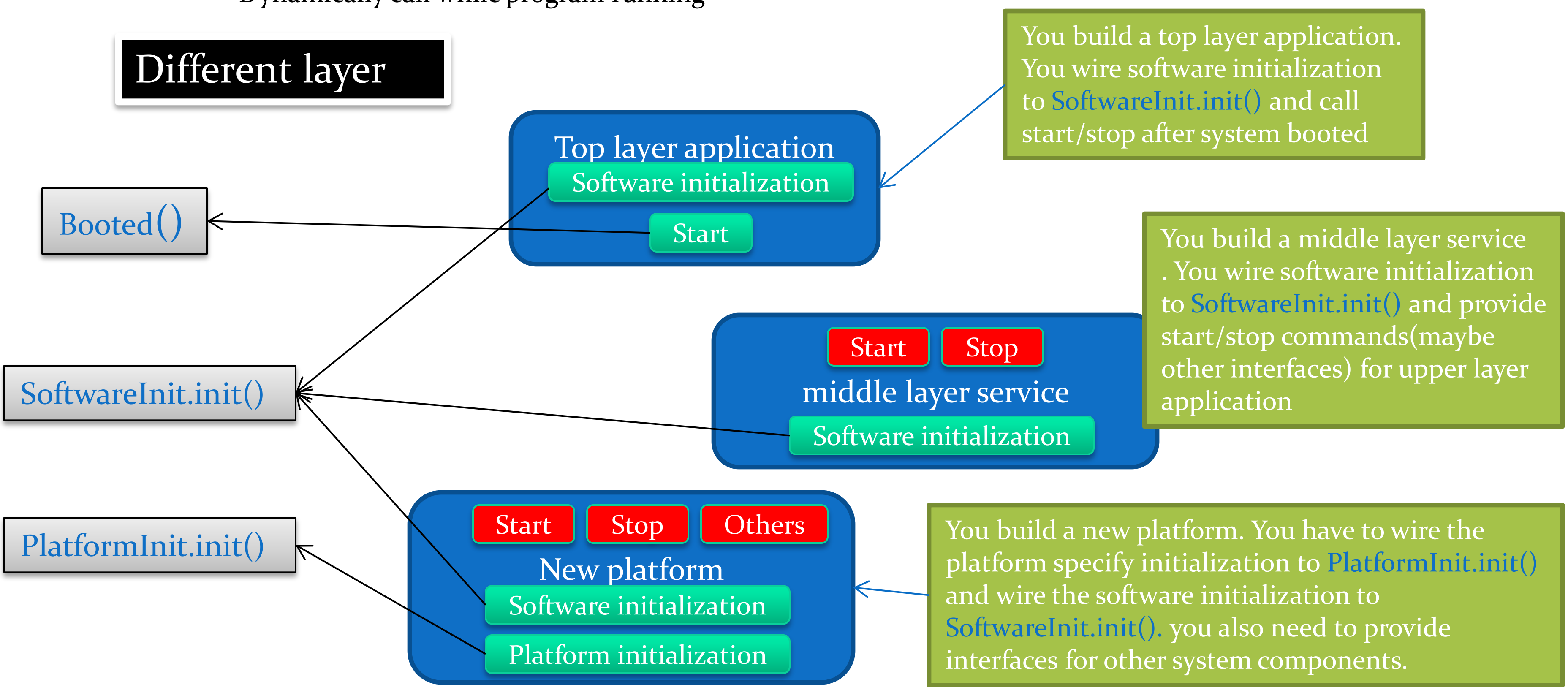
When RealMainP calls Scheduler.init(), it automatically calls the TinySchedulerC.init().

Initialization

- **Task scheduler Initialization**
  - Initialize the task scheduler
- **Component initialization.**
  - PlatformInit
    - wired to the platform-specific initialization component
    - No other component should be wired to PlatformInit
  - SoftwareInit
    - Any component that requires initialization can implement the Init interface and wire itself to MainC's SoftwareInit interface
- **Signal that the boot process has completed**
  - Components are now free to call start() and other commands on any components they are using

Separate Initialization And Start/Stop

- For example, radio service
  - Initialization: specify node address, PAN id and etc.
    - Only run once
  - Start/stop: start or stop the radio transceiver
    - Dynamically call while program running



Wire SoftwareInit

When RealMainP calls softwareInit, it will wires to FooP.Init.init(), which is implemented by FooP module

```
Configuration FooC {
}
Implementation {
  components MainC, FooP;

  MainC.SoftwareInit -> FooP;
}
```

```
module FooP {
  provides interface Init;
}
Implementation {
  command error_t Init.init() {
      initialization something
      ...............
  }
}
```

```
interface Init {
  command error_t init();
}
```

- Task And Scheduler

Software Architectures

- Round Robin with Interrupts

```
for(;;)  // forever loop
{
    1. wait for interrupt(sleep)
    if( Event 1 occurred) {
        do something
    }
    if( Event 2 occurred) {
        do something
    }
    if( Event 3 occurred) {
        do something
    }
}
```

```
(ISR) Interrupt Service Routines  1 ()
{
    1. do critical things
    2. set event 1 occurred flag
}
```

```
(ISR) Interrupt Service Routines  2 ()
{
    1. do critical things
    2. set event 2 occurred flag
}
```

```
(ISR) Interrupt Service Routines  3 ()
{
    1. do critical things
    2. set event 3 occurred flag
}
```

- Problem: no proirity

Software Architectures

- **Function-Queue-Scheduling**

```
for(;;)  // forever loop
{
    1. wait for interrupt(sleep)
    While (function queue is not  empty)
     {
         call first function on queue
     }
}
```

```
(ISR) Interrupt Service Routines  1 ()
{
    1. do critical things
    2. put function_1 on queue
}
```

```
(ISR) Interrupt Service Routines  2 ()
{
    1. do critical things
    2. put function_2 on queue
}
```

```
(ISR) Interrupt Service Routines  3 ()
{
    1. do critical things
    2. put function_3 on queue
}
```

- Worst wait for highest priority
    - bounded by the longest function

On TinyOS

- Software Architecture of TinyOS
  - Function-Queue-Scheduling
- Essentially, when running on a platform
  - TinyOS is not a Operating System
    - It depends on your definition of "OS"
  - It performs many check at compile time through nesC
    - Check memory usage
    - Prevent dynamic memory allocation
    - Warn potential race condition
    - Determine lowest acceptable power state (for low power)

Tasks And Scheduler

A task can be post to the task queue by a ISR or other task

- Tasks And Scheduler in TinyOS

Task_5	Task_2	Task_1	Task_3	Task_7

```
for(;;)  // forever loop
{
    1. wait for interrupt(sleep)
    While (task queue is not  empty)
    {
        call a task in queue based on FIFO schedule
    }
}
```

```
Task_5 () {
    1. do something
    2. post task_7
}
```

```
(ISR) Interrupt Service Routines 1 ()
{
    1. do critical things
    2. post task_1
}
```

```
(ISR) Interrupt Service Routines 2 ()
{
    1. do critical things
    2. post task_2
}
```

```
(ISR) Interrupt Service Routines 3 ()
{
    1. do critical things
    2. post task_3
}
```

- Worst wait
  - Total execution time of tasks ahead

Tasks

- How to use
  - declare: `task void taskname() { ... }`
  - post: `post taskname();`

```
task void computeTask() {
  uint32_t i;
  for (i = 0; i < 400001; i++) {}
}

event void Timer0.fired() {
  call Leds.led0Toggle();
  post computeTask();
}
```

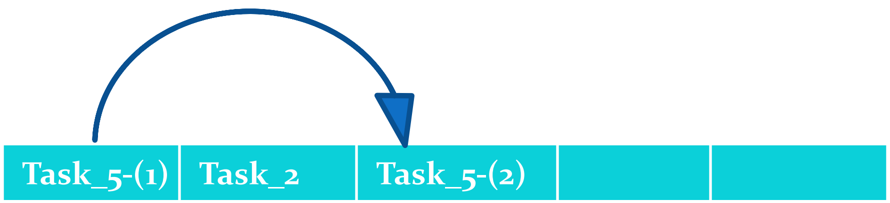
- Tasks in TinyOS 2.x
  - A basic post will only fail if and only if the task has already been posted and has not started execution
    - You cannot have two same idle task in the queue
  - At most 255 tasks in queue

Rules of Thumb

- Keep task short
- Divided long task into short sub-tasks

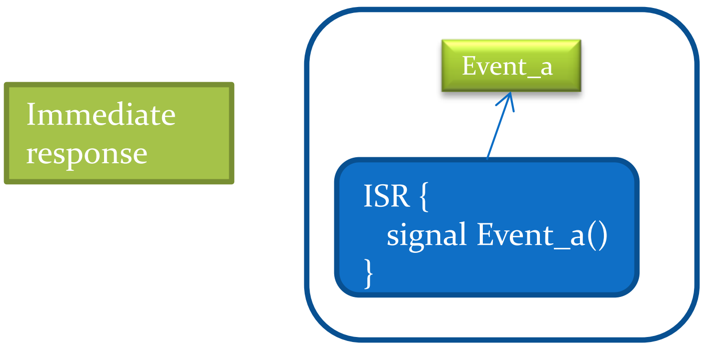
If Task_5 runs 5 seconds.
Task_2 toggle a LED, occurred every second.
In this situation, LED will only toggle every 5 seconds.

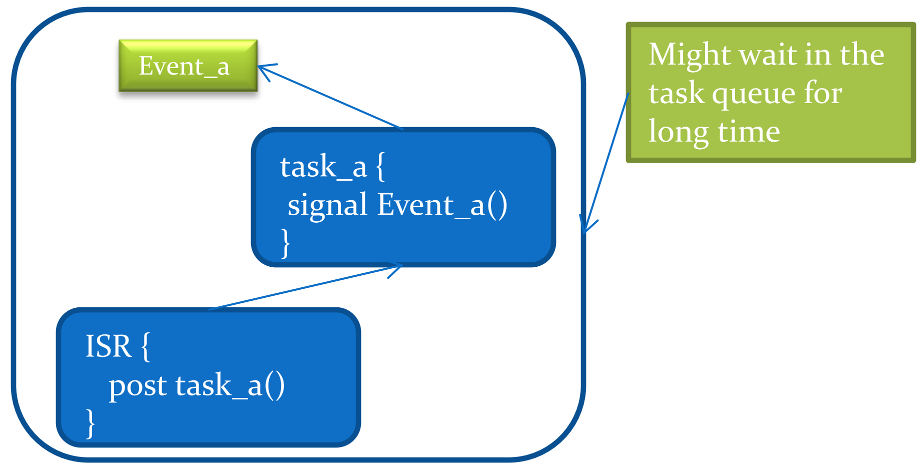
Divided Task_5 into 10 sub-tasks, each runs 0.5 second. A sub-task post another consecutive sub-task after it finish.
Now, LED can toggle every 1 seconds.



Interrupts In TinyOS

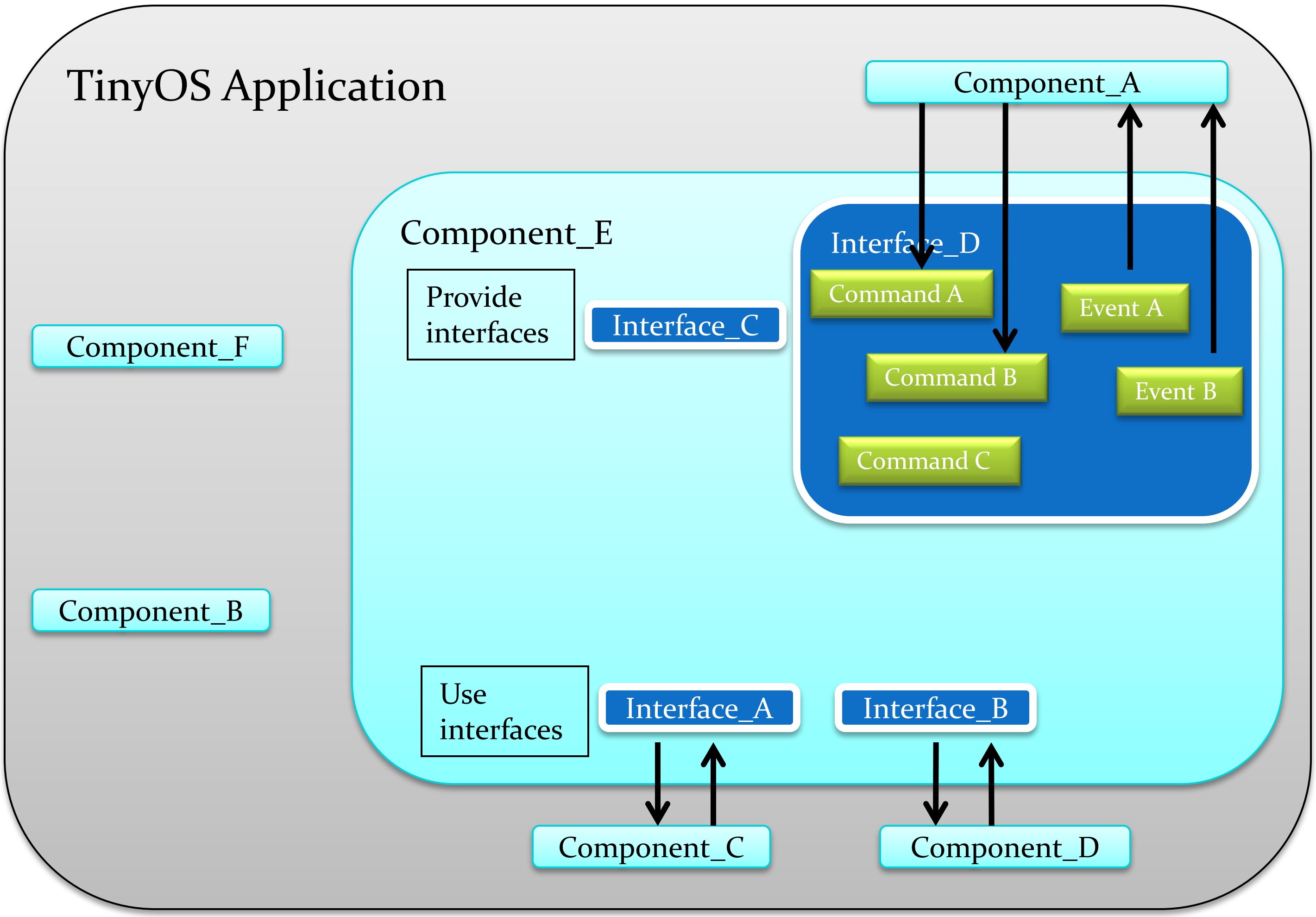
- Is an event call from a ISR (Interrupt Service Routine)?
  - I don't know!!
  - Didn't specify in their documentation (or I miss it)
  - But it is important
    - If your application requires a real-time response to external event, it must call from ISR





- What I found is
  - commands and events that are called from interrupt handlers **must** be marked *async (demo)*

Summary



1. Application consists one or more components.
2. Components provide and/or use interfaces.
3. Interfaces specify *commands* (down call) and *events* (up call)

Summary

- Application consists one or more components.
  - Configuration:
    - wire interfaces of different components together
  - Module
    - Implementation of interfaces
- Different components communicate through interfaces
  - Command: down-call
  - Event: up-call
- Writing a top layer TinyOS application
  - Choose the interface you want to use
  - Provide interfaces if necessary
  - Wire the interfaces to other components provide/use these interfaces
  - Implement events and commands

Further Reading

- Tutorials
  - http://www.tinyos.net/tinyos-2.x/doc/html/tutorial/index.html
  - A good starting point
- TinyOS Programming Manual
  - http://www.tinyos.net/tinyos-2.x/doc/pdf/tinyos-programming.pdf
  - nesC programming language
- TinyOS Enhancement Proposals (TEPs)
  - describe the structure, design goals, and implementation of parts of the system as well as nesC and Java source code documentation
  - http://www.tinyos.net/tinyos-2.x/doc/

About TinyOS

- My opinions
  - Writing a high level program is relative easy
  - But debugging could be a big problem
    - You don't know what's going on inside
  - Documentation is important
    - One of the big problem in TinyOS 1.x
    - They put a lots of effort in documenting TinyOS 2.x
      - Still some parts missing, some inconsistency
      - But it is much better than TinyOS 1.x
  - Trade off between (efficiency, optimization) and (portability, reusability)
  - Is portability important?