MODULE 2:

TinyOS and nesC

TinyOS and nesC

- **TinyOS:** Operating system for sensor networks
- **nesC:** Programming language for sensor networks

Why TinyOS?

- Traditional OSes are not suitable for networked sensors
- Characteristics of networked sensors
  - Small physical size & low power consumption
    - Software must make efficient use of processor & memory, enable low power communication
  - Concurrency intensive
    - Simultaneous sensor readings, incoming data from other nodes
    - Many low-level events, interleaved w/ high-level processing
  - Limited physical parallelism (few controllers, limited capability)
  - Diversity in design & usage
    - Software modularity – application specific

TinyOS Solution

- Support concurrency
  - event-driven architecture
- Software modularity
  - application = scheduler + graph of components
  - A component contains commands, event handlers, internal storage, tasks
- Efficiency: get done quickly and then sleep
- Static memory allocation

TinyOS Computational Concepts

1. Events

- Time critical
- Caused by interrupts (Timer, ADC, Sensors)
- Short duration

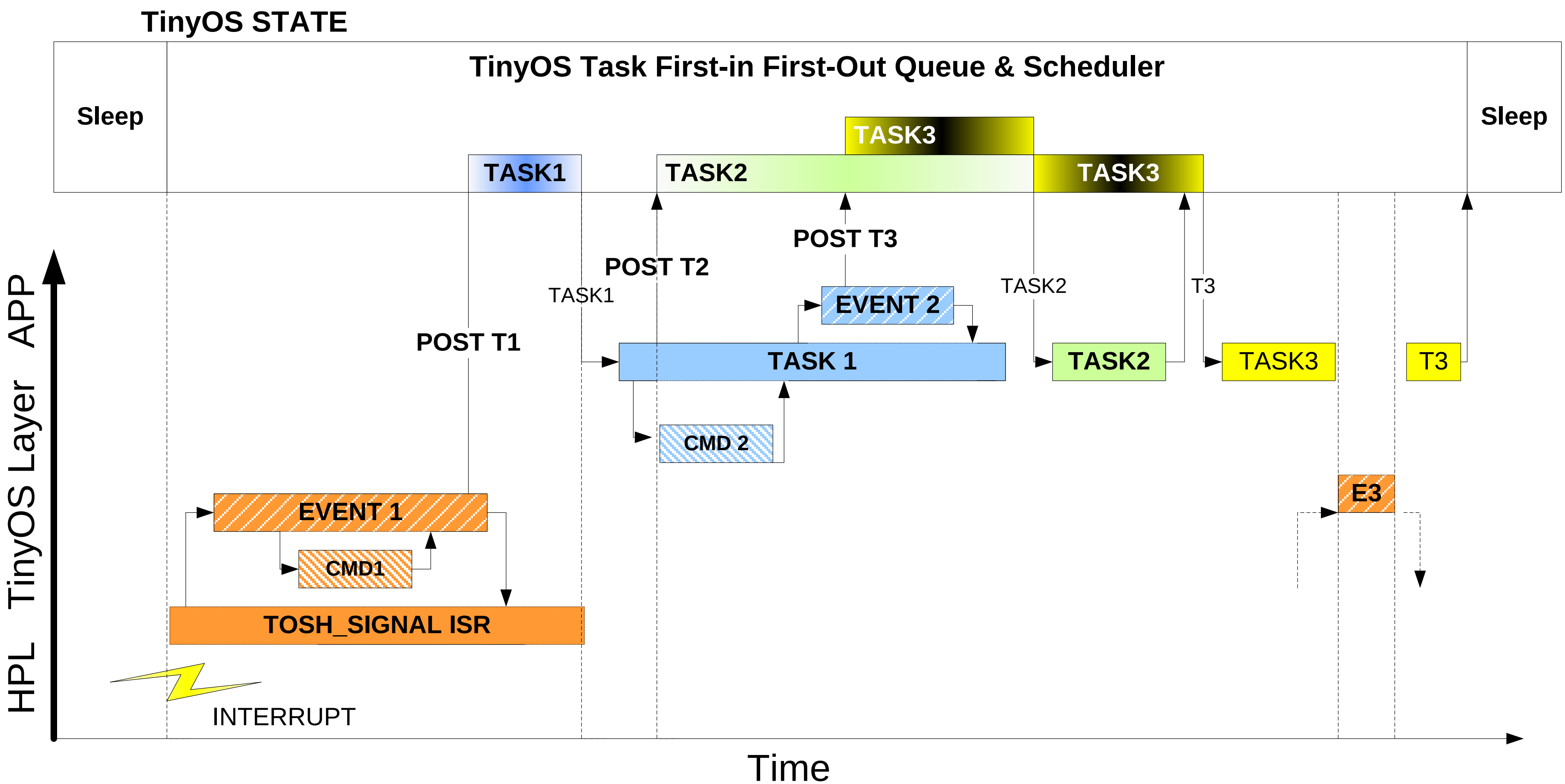
2. Commands

- Cause Actions to be initiated.
- Request to a component to perform service (e.g, start sensor reading)
- Non-blocking, need to return status
- Postpone time-consuming work by posting a task (split phase w/ callback event)
- Can call lower-level commands

3. Tasks

- Time flexible (delayed processing)
- Run sequentially by TOS Scheduler
- Run to completion with respect to other tasks
- Can be preempted by events

TinyOS Execution Model



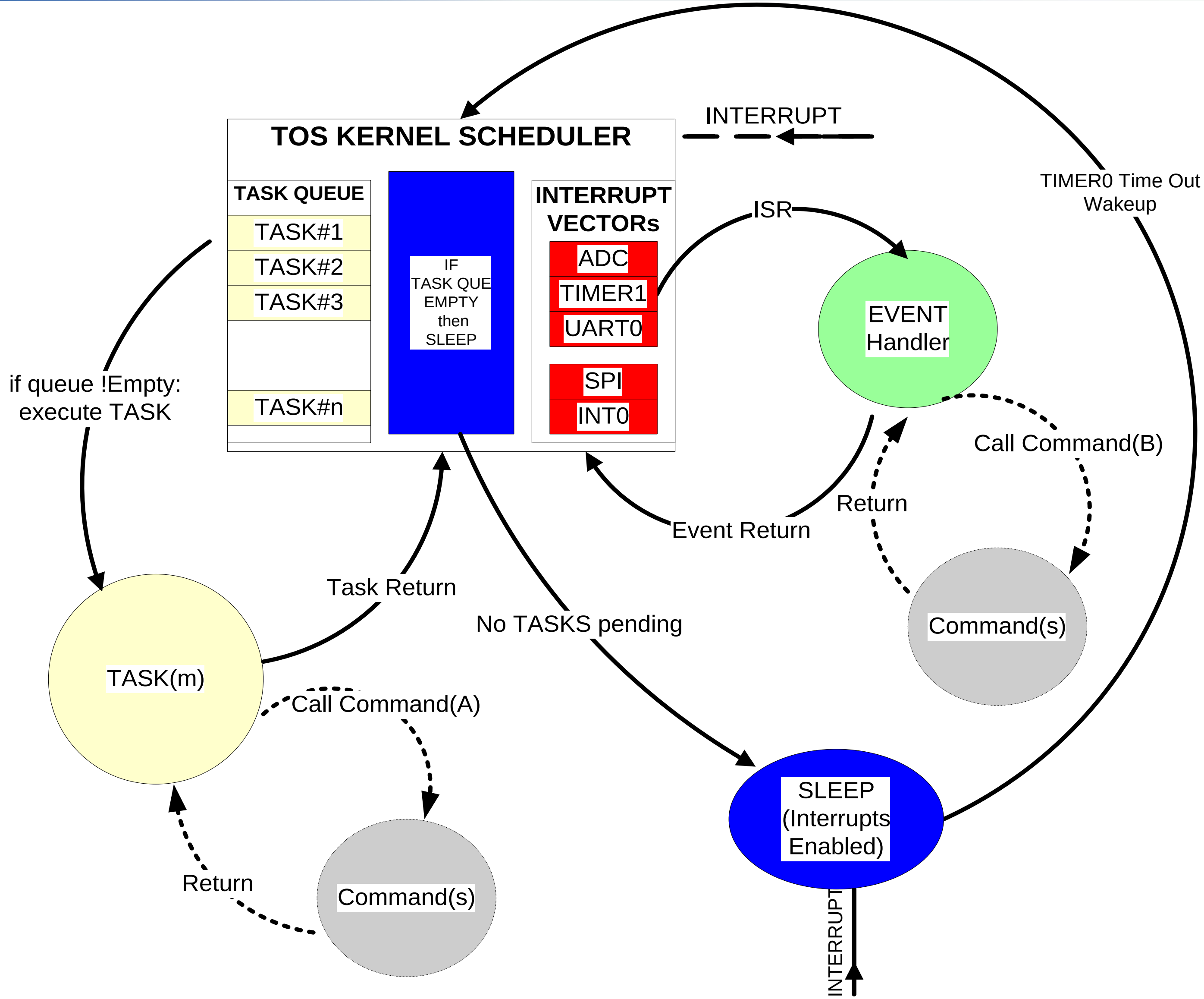
Concurrency

- Two threads of execution
  - Tasks
    - deferred execution
    - tasks cannot preempt other tasks
  - Hardware event handler: respond to interrupts
    - Interrupts can preempt tasks
- Scheduler
  - Two level scheduling
    - interrupts (vector) and tasks (queue)
  - Task queue is FIFO
  - Scheduler puts processor to sleep when no event/command is running and task queue is empty

Interface	Description
Clock	Hardware clock
EEPROMRead/Write	EEPROM read and write
HardwareId	Hardware ID access
I2C	Interface to I2C bus
Leds	Red/yellow/green LEDs
MAC	Radio MAC layer
Mic	Microphone interface
Pot	Hardware potentiometer for transmit power
Random	Random number generator
ReceiveMsg	Receive Active Message
SendMsg	Send Active Message
StdControl	Init, start, and stop components
Time	Get current time
TinySec	Lightweight encryption/decryption
WatchDog	Watchdog timer control

Fig. 1. Core interfaces provided by TinyOS

TinyOS Execution Model (revisited)



TinyOS Theory of Execution: Events & Tasks

- Consequences of an event
  - Runs to completion
  - Preempt Tasks
- What can an event do?
  - `signal` events
  - `call` commands
  - `post` tasks

- Consequences of a task
  - No preemption mechanism
  - Keep code as small execution pieces to not block other tasks too long
  - To run a long operations, create a separate task for each operation, rather than using on big task
- What can initiate (post) tasks?
  - Command, event, or another task

TinyOS Summary

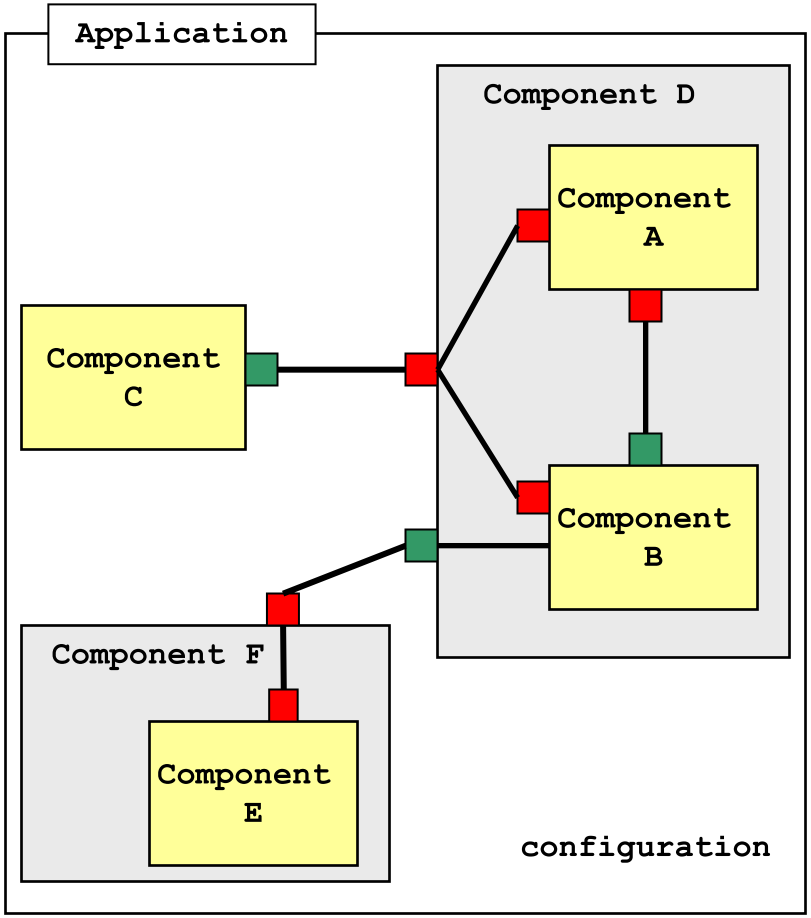
- Component-based architecture
  - Provides reusable components
  - Application: graph of components connected by “wiring”
- Three computational concepts
  - Event, command, task
- Tasks and event-based concurrency
  - Tasks: deferred computation, run to completion and do not preempt each other
  - Tasks should be short, and used when timing is not strict
  - Events: run to completion, may preempt tasks
  - Events signify completion of a (split-phase) operation or events from the environment (e.g., hardware, receiving messages)

nesC: A programming language for sensor networks

- Main features
  - Support and reflect TinyOS's design
    - Support components, event-based concurrency model
  - Extending C with support for components
    - Components *provide* and *use* interfaces
    - Application: wiring components using *configurations*
  - Whole-program analysis & optimization
    - Detect race conditions
  - Static language
    - No dynamic memory allocation, call-graph fully known at compilation
- No multiprogramming
  - Each mote runs a single application

nesC model

- Application: graph of components
  - Main component
    - init, start, stop
    - first component executed
  - Other components
- Components
  - modules
  - configurations
- Interfaces: point of access to components
  - uses
  - provides



nesC Component Types

Components

- **Modules**
  - Provides, uses, and implements interface(s)
  - Contains C-like code
- **Configurations**
  - A collection of other components put together by “wiring” syntax.
  - Does not use C-like code
  - Can also provide and use interfaces

Why Modules and Configurations?

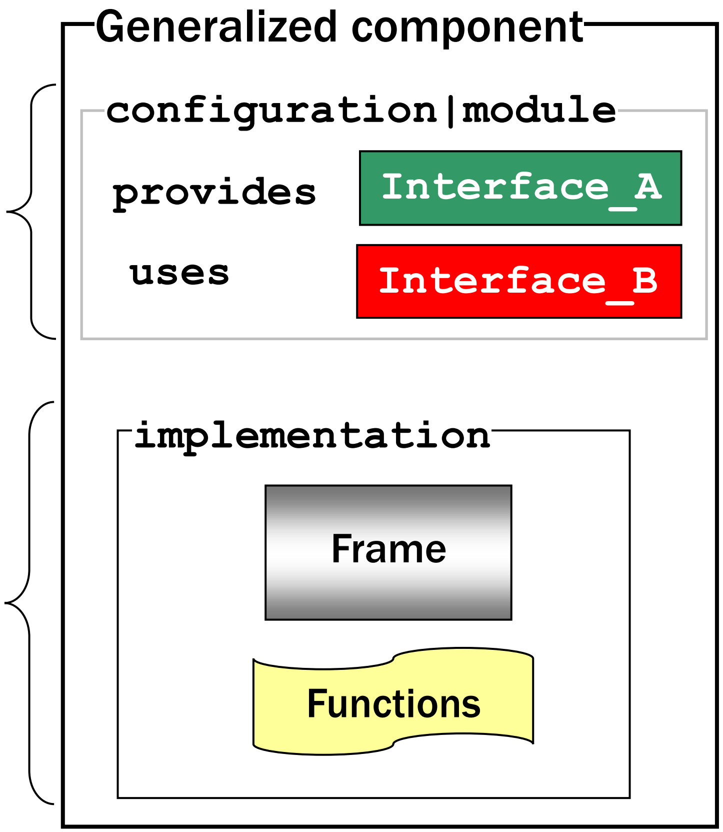
- Allow a developer to “snap together” applications using pre-build components without additional programming.

- For example, a developer could provide a configuration that simply wires together one or more pre-existing modules.

- The idea is for developers to provide a set of components, a “library,” that can be re-used in a wide range of applications.

nesC Component

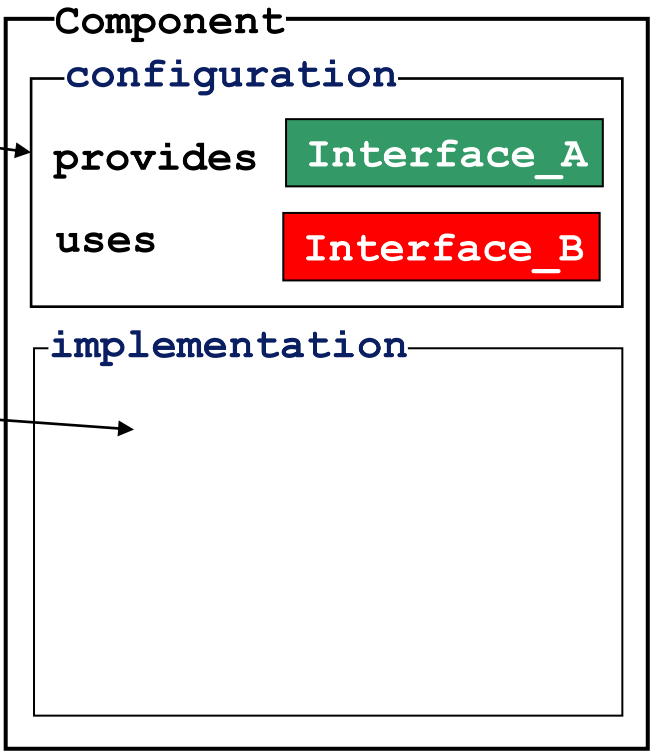
- **Specification**
  - Identified by the keyword `configuration` or `module`
  - List of interfaces that component
    - `uses`, `provides`
    - Alias interfaces `as` new name
- **Implementation**
  - Identified by the keyword `implementation`
  - Defines internal workings
  - May include other components and associated "wiring"



NOTE: This model applies to both modules and configurations

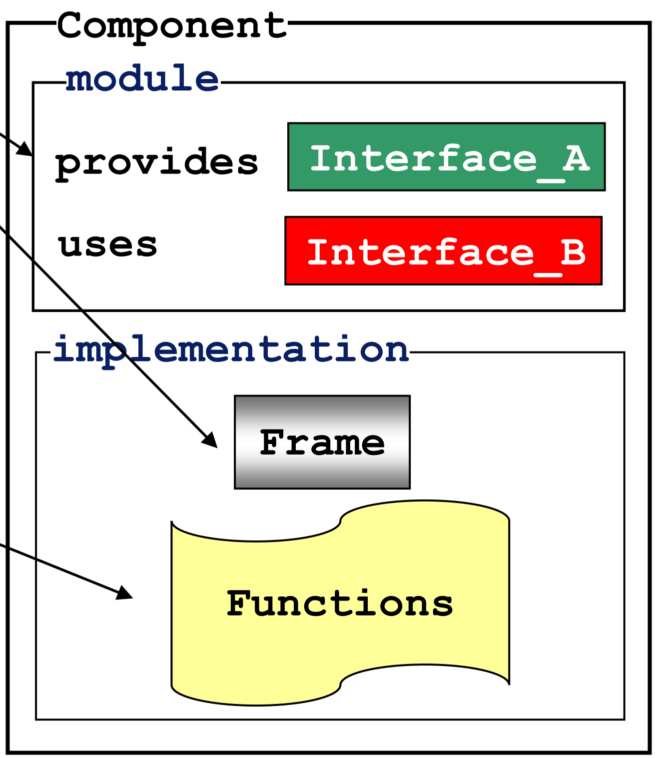
nesC Configuration – A Bare Minimum Configuration

```
configuration
  ConfigurationName {
    provides interface …;
    uses interface …
}
implementation {
    // wiring


    ……


}
```



nesC Module – A Bare Minimum Module

```
module ModuleName {
   provides interface StdControl;
}
implementation {
   // ========== FRAME =========

   // ========== FUNCTIONS =====
   command result_t StdControl.init()
   {
      return SUCCESS;
   }
   command result_t StdControl.start()
   {
      return SUCCESS;
   }
   command result_t StdControl.stop()
   {
      return SUCCESS;
   }
}
```



Example: Blink Configuration

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

  Main.StdControl ->
SingleTimer.StdControl;
  Main.StdControl -> BlinkM.StdControl;

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

Example: Blink Module

```
module BlinkM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
    interface Leds;
  }
}
implementation {
  command result_t StdControl.init() {
    call Leds.init();
    return SUCCESS;
  }
```

Example: Blink Module (cont'd)

```
command result_t StdControl.start() {
    // Start a repeating timer that fires every
1000ms
    return call Timer.start(TIMER_REPEAT,
1000);
  }

  command result_t StdControl.stop() {
    return call Timer.stop();
  }

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

nesC Interface

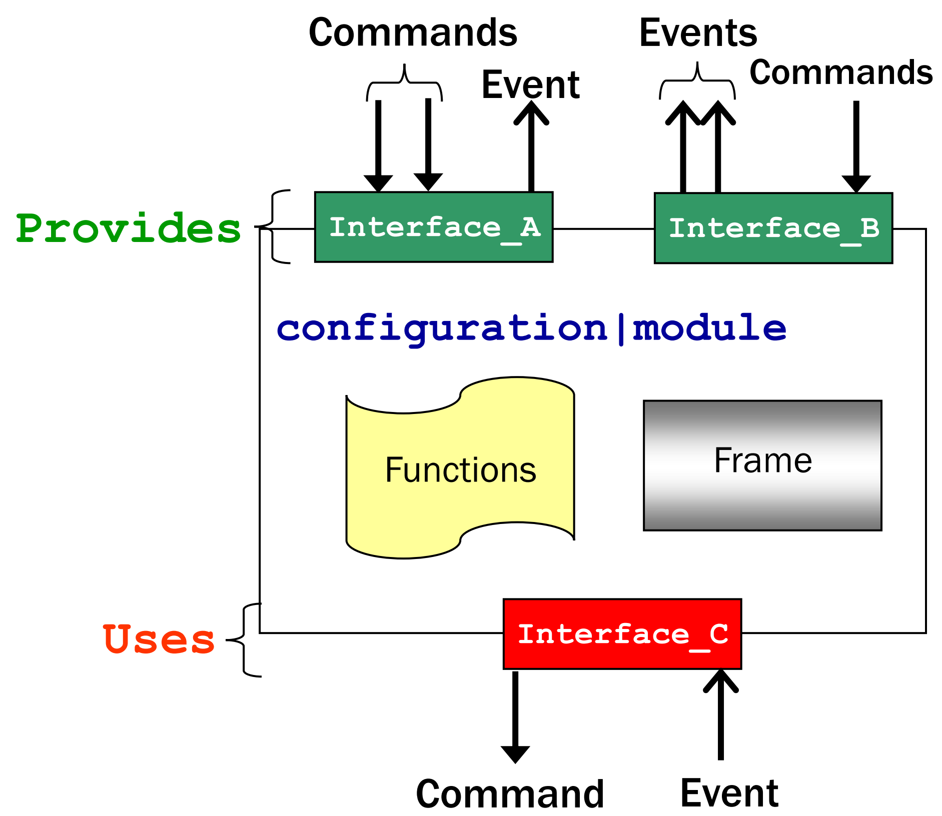
- define “public” methods that a component can use
- contain one or more **commands** and/or **events**
- group functionality, e.g.,
  - Standard control interface
  - Split-phase operation

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

```
interface Send {
 command void send(TOS_Msg *m);
 event void sendDone();
}
```

Interface : provides & uses

- Provides: Exposes functionality to others
- Uses: Requires another component
- Interface *provider* must implement commands
- Interface *user* must implement events



nesC Interface Examples

```
module MyAppM {
  provides interface StdControl;
  uses interface ADC;
  uses interface Timer;
  uses interface Send;
}
implementation {
  …
}
```

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

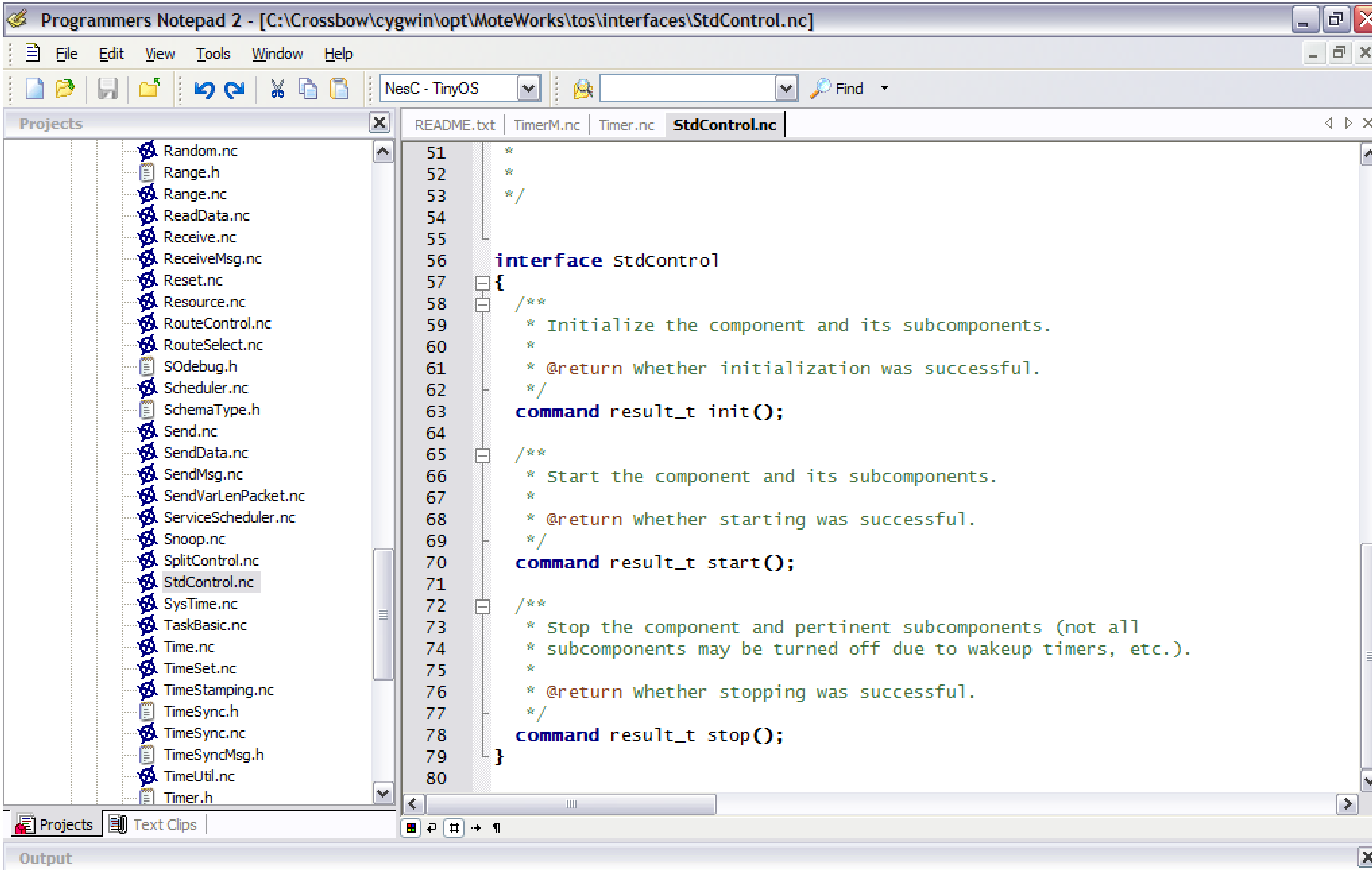
```
interface ADC {
 command void getData();
 event void dataReady(int data);
}
```

```
interface Timer {
 command void start(int interval);
 command void stop();
 event void fired();
}
```

```
interface Send {
 command void send(TOS_Msg *m);
 event void sendDone();
}
```

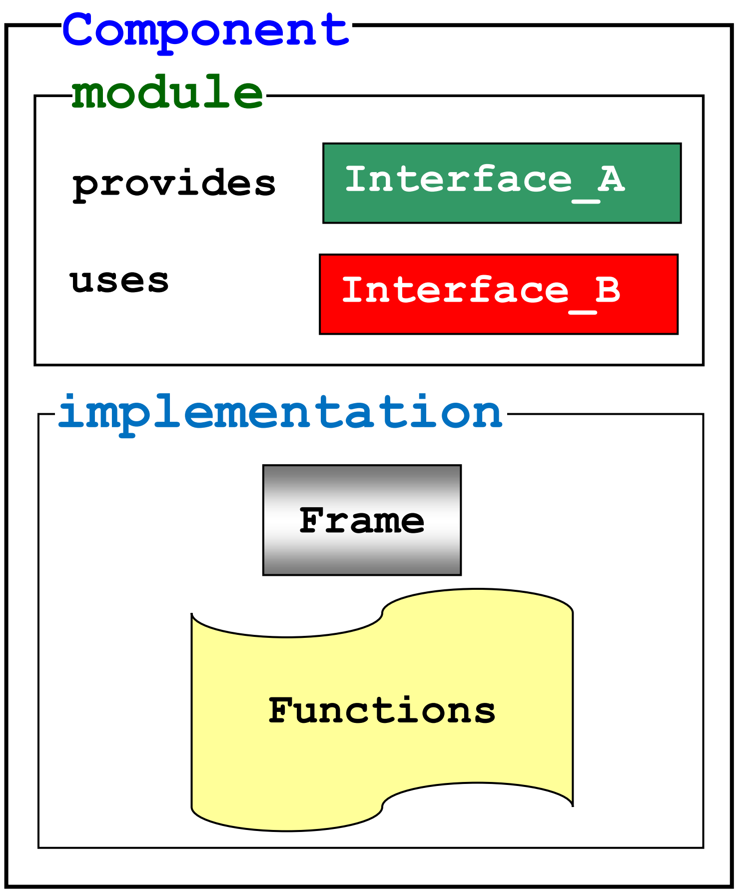
Questions: what need to be implemented by MyApp?

Interfaces: in */MoteWorks/tos/interfaces/*



nesC Module Implementation

- Frame
  - Global variables and data
  - One per component
  - Statically allocated
  - Fixed size
- Functions
  - Implementation of:
    - Commands, events, tasks
  - Commands and Events are simply C function calls



Frame

```
module fooM {
  provides interface I1;
  uses interface I2;
}

implementation {
  uint8_t count=0;

  command void I1.increment() {
    count++;
  }

  event uint8_t I2.get() {
    return count;
  }
}
```

```
Call I1.increment(); //first call
Call I1.increment(); //second call
signal I2.get();     //return 2
```

Modules maintain local and persistent state

- Module states or variables are statically allocated
- Each Module has a separate variable name space
- Variables declared in Command and Event are local and allocated on the stack

What can a module do?

Calling commands

& signaling events

```
module MyAppM {
  provides interface StdControl;
  uses interface Clock;
  …
}
implementation {
  command result_t StdControl.init(){
     call Clock.setRate(TOS_I1PS, TOS_S1PS);
  }…
}
```

Posting tasks

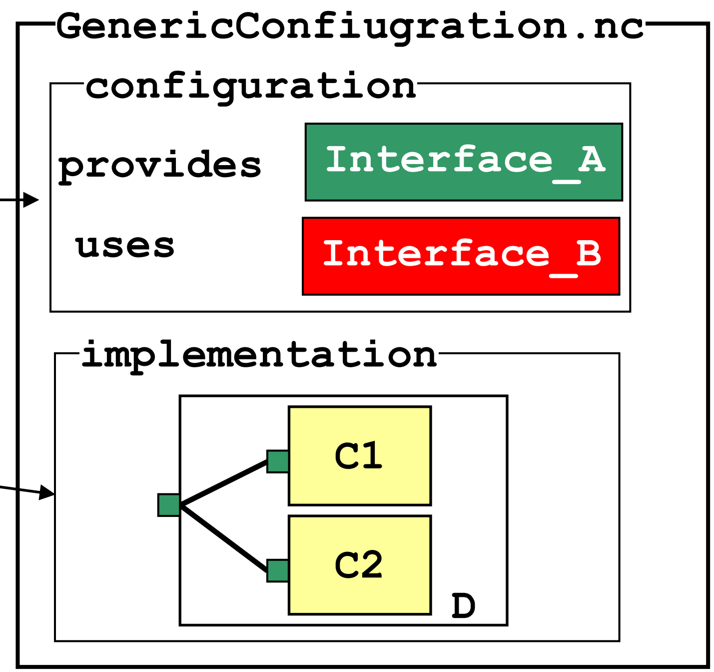
```
module MyAppM {
…
}
implementation {
  task void processing(){
    if(state) call Leds.redOn();
    else call Leds.redOff();
  }
  event result_t Timer.fired()
  {
    state = !state;
    post processing();
    return SUCCESS;
  }…
}
```

nesC Configuration

- wiring together other components
    - No code, just wiring

GenericConfiguration.nc

```
configuration D {
   provides interface_A;
   uses interface_B;
}
implementation {
   components C1, C2;
   X = C1.X;
   X = C2.X;
}
```



nesC Wiring Syntax

- Binds (connects) the User to an Provider's implementation

```
User.interface -> Provider.interface
```

  - Example

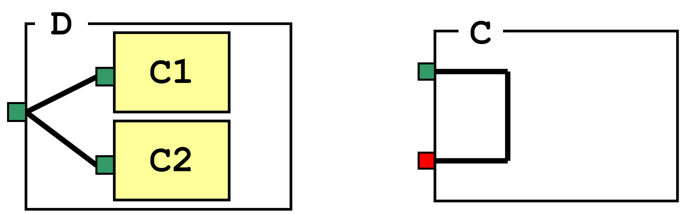
```
MyApp_Timer.Timer -> TimerC.Timer[unique("Timer")];
```

- Connected elements must be compatible
  - Interface to interface, command to command, event to event
  - Example: you can only wire interface `Send` to `Send`, but cannot connect `Send` to `Receive`

Three nesC Wiring Statements

1. Alias or pass through linkage

   endpoint$_1$ = endpoint$_2$

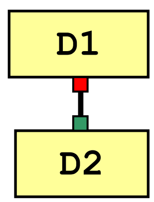


C.nc

```
configuration C {
 provides interface X as
   Xprovider;
 uses interface X as Xuser;
}
implementation {
   Xuser = Xprovider;
}
```

2. Direct linkage, style 1

   endpoint$_1$ -> endpoint$_2$



3. Direct linkage, style 2

   endpoint$_1$ <- endpoint$_2$

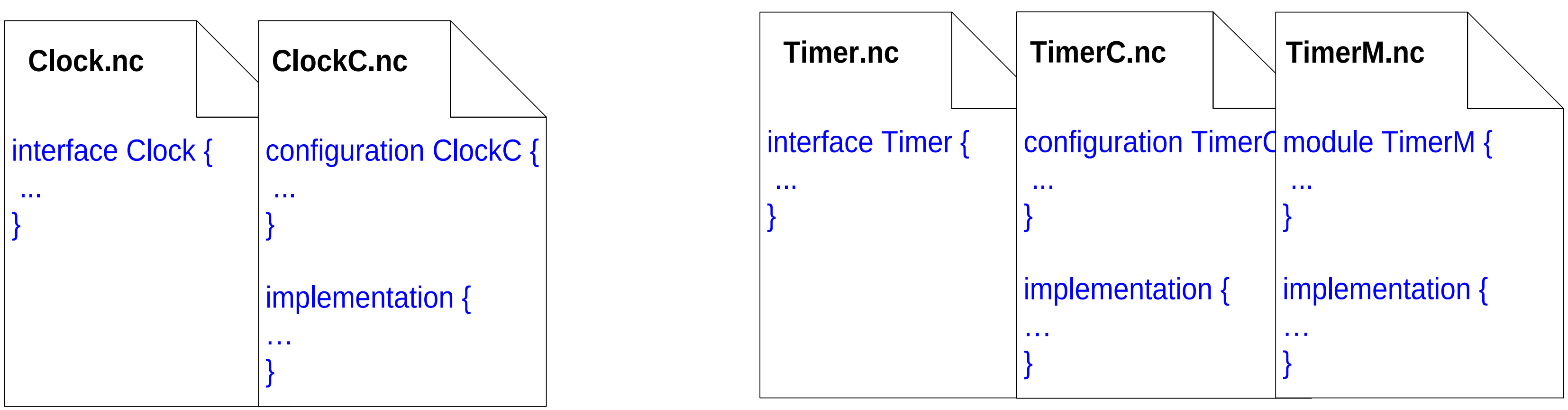
   which is *equivalent* to: endpoint$_2$ -> endpoint$_1$

How mote application starts?

```
int main() {
  call hardwareInit();              //init hardware pins

  TOSH_sched_init();                //init scheduler

  call StdControl.init();           //init user app
  call StdControl.start();          //start user app

  while(1) {
    TOSH_run_task();                //infinite spin loop
                                      for task
  }
}
```

nesC Filename Convention

- nesC file suffix: `.nc`
- `C` is for **configuration** (`Clock`, `ClockC`)
  - "`C`" distinguishes between an interface and the component that provides it
- `M` is for **module** (`Timer`, `TimerC`, `TimerM`)
  - "`M`" when a single component has both configuration and module



Dealing with Concurrency in nesC

- TinyOS two execution threads: Tasks and interrupts
  - Tasks cannot preempt other tasks
  - Interrupts can preempt tasks
- Race condition
  - Concurrent update to shared state
- Any update to shared state that is reachable from *asynchronous* code is a potential race condition
  - *Asynchronous* code
    - Code that is reachable from at least one interrupt handler
  - *Synchronous* code
    - Commands, events, or tasks only reachable from tasks

`atomic` Keyword

- `atomic`: denote a block of code that runs uninterrupted (interrupts disabled)
  - Prevents race conditions

- When should it be used?
  - Required to update global variables that are referenced in `async` event handlers
  - Must use atomic block in all other functions and tasks where variable is referenced

- nesC compiler generates warning messages for global variables that need atomic blocks, e.g.,

```
SensorAppM.nc:44: warning: non-atomic
accesses to shared variable ‘voltage’
```

`atomic` Keyword (cont'd)

- Disables all interrupts, therefore use with caution and intention
- Affects code block within {…} that follows it
- Useful for locking a resource

Example

```
atomic {
    if (!busy)
        busy = TRUE;
}
```

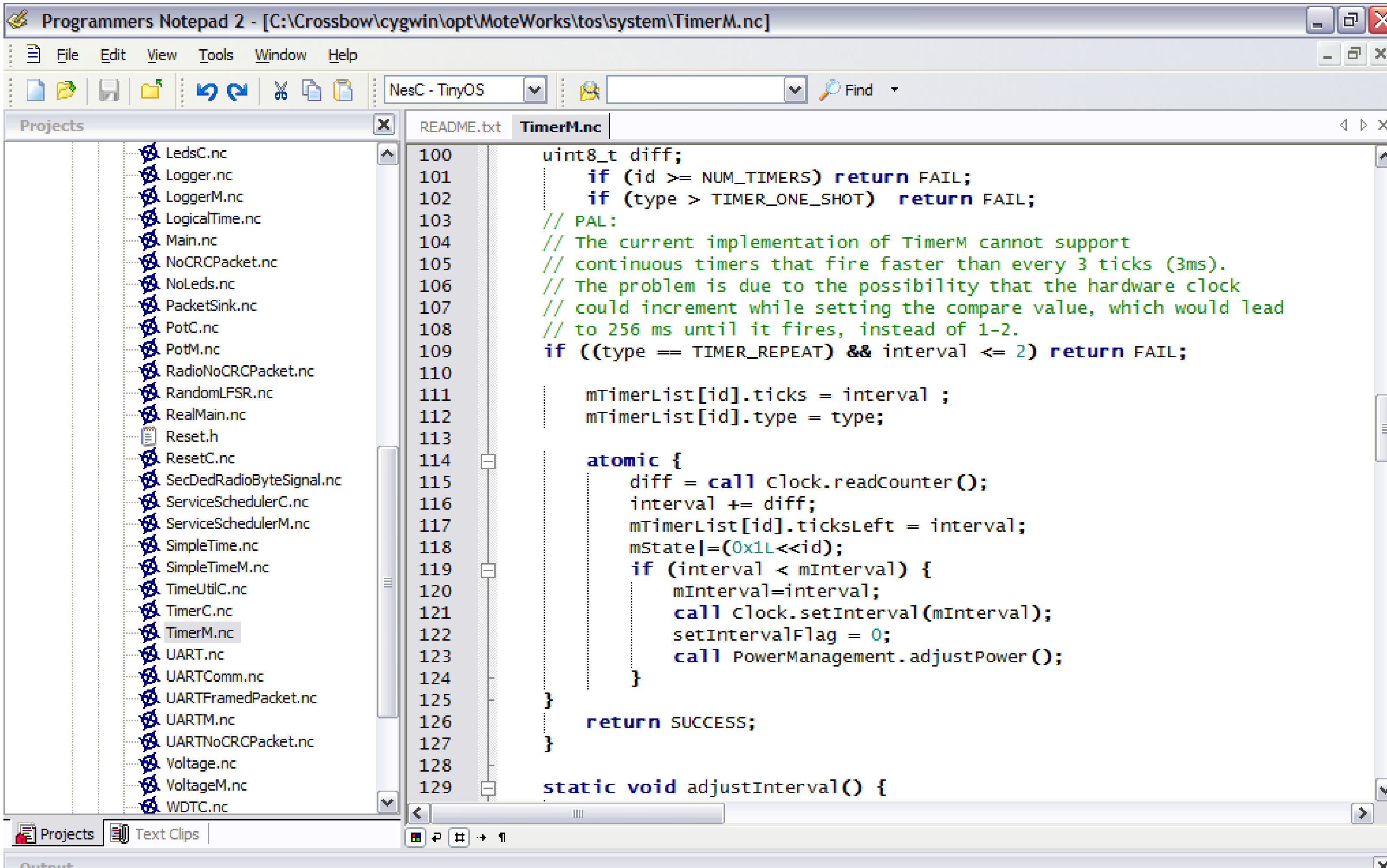
Compiles to:

```
   cli();               // disable interrupts
   lda r1, busy         // load busy to register
   jnz r1, inUse        // check busy
   str busy, 1          // set busy to true
inUse:
   sbi();               // enable interrupts
```

No interrupts will disrupt code flow

`atomic` Syntax Example

From MoteWorks/tos/system/TimerM.nc



`async` Syntax

- `async` attribute used to indicate that command or event is part of an asynchronous flow
- `async` processes are "decoupled" by posting a task (to return quickly)

tinyos-1.x/tos/system/TimerM.nc

```
async event result_t Clock.fire() {
      post HandleFire();
        return SUCCESS;
    }
```

`Post` task "decouples" the `async` event

nesC Features for Concurrency

- `post`
  - Puts a function on a task queue
  - Must be `void foo(void)`

```
task void do-work() { //declares a task that does work}
post do-work();
```

- `atomic`
  - Turn off interrupts
- `async`
  - Use `async` to tell compiler that this code can be called from an interrupt context – used to detect potential race conditions
- `norace`
  - Use `norace` to tell compiler it was wrong about a race condition existing (the compiler usually suggests several false positives)

Concurrency Example

```
module SurgeM {…}
implementation {
 bool busy;
 norace unit16_t sensorReading;

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

Concurrency Example (cont'd)

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

  event result_t ADC.dataReady(unit16_t data) {
     sensorReading = data;
     post sendData();
    return SUCCESS;
  }

...
}
```

nesC Keywords

Keyword	Description
`component`	Building blocks of a nesC application. Can be a module or a configuration
`module`	A basic component implemented in nesC
`configuration`	A component made from wiring other components
`interface`	A collection of event and command definitions
`implementation`	Contains code & variables for module or configuration
`provides`	Defines interfaces provided by a component
`uses`	Defines interfaces used by a module or configuration
`as`	Alias an interface to another name
`command`	Direct function call exposed by an interface
`event`	Callback message exposed by an interface

Appendix: TinyOS Programming Tips

- By Phil Levis guide to TinyOS 2.0 Programming Guide
  - Note: Not all TinyOS 2.0 concepts apply to MoteWorks
- Last update: June 28, 2006

TinyOS Programming Hints – Condensed (1 of 4)

Programming Hint 1: It's dangerous to signal events from commands, as you might cause a very long call loop, corrupt memory and crash your program.

Programming Hint 2: Keep tasks short.

Programming Hint 3: Keep code synchronous when you can. Code should be `async` only if its timing is very important or if it might be used by something whose timing is important.

Programming Hint 4: Keep `atomic` sections short, and have as few of them as possible. Be careful about calling out to other components from within an atomic section.

Programming Hint 5: Only one component should be able to modify a pointer's data at any time. In the best case, only one component should be storing the pointer at any time.

TinyOS Programming Hints – Condensed (2 of 3)

Programming Hint 6: Allocate all state in components. If your application requirements necessitate a dynamic memory pool, encapsulate it in a component and try to limit the set of users.

Programming Hint 7: Conserve memory by using `enums` rather than const variables for integer constants, and don't declare variables with an `enum` type.

*__Programming Hint 8:__ In the top-level configuration of a software abstraction, auto-wire `init` to `MainC`. This removes the burden of wiring `init` from the programmer, which removes unnecessary work from the boot sequence and removes the possibility of bugs from forgetting to wire.

*__Programming Hint 9:__ If a component is a usable abstraction by itself, its name should end with C. If it is intended to be an internal and private part of a larger abstraction, its name should end with P. Never wire to P components from outside your package (directory).

*TinyOS 2.0 specific

TinyOS Programming Hints – Condensed (3 of 4)

Programming Hint 10: Use the `as` keyword liberally.

Programming Hint 11: Never ignore combine warnings.

Programming Hint 12: If a function has an argument which is one of a small number of constants, consider defining it as a few separate functions to prevent bugs. If the functions of an interface all have an argument that's almost always a constant within a large range, consider using a *parameterized interface* to save code space. If the functions of an interface all have an argument that's a constant within a large range but only certain valid values, implement it as a parameterized interface but expose it as individual interfaces, to both minimize code size and prevent bugs.

Programming Hint 13: If a component depends on `unique`, then `#define` a string to use in a header file, to prevent bugs from string typos.

*TinyOS 2.0 specific

TinyOS Programming Hints – Condensed (4 of 4)

- Programming Hint 14: Never, ever use the “packed” attribute.

- Programming Hint 15: Always use platform independent types when defining message formats.

- Programming Hint 16: If you have to perform significant computation on a platform independent type or access it many (hundreds or more) times, then temporarily copying it to a native type can be a good idea.