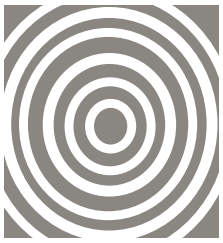
センサアプリケーションMyApp_Sensorの作成

目標

- センサボードから光データを読むMoteファームウェアを作成
- シリアルポートにセンサデータを含むメッセージを送信



ハードウェアとPCの設定

1. 2つのMICA Mote:
   MICA2 (MPR4x0) あるいは MICAz (MPR2400J)
2. 1つのセンサボード:
   MDA100, MTS300 あるいは MTS310
3. 1つのゲートウェイ:
   MIB510, MIB520 あるいは MIB600 と各関連するハードウェア（ケーブル、電源）
4. *MoteWorks*をインストールしてあるWindows PC

アプリケーションを作成するのに必要なこと:

- 入力あるいはコピー&ペーストした必要なコードと補助ファイル
- アプリケーションのビルド（コンパイル）とインストール
- コードと補助ファイルをよく読むこと

MyApp_Sensorについて

MyApp_Sensorとは

- 簡単なセンシングアプリケーションでセンサボードの光センサをサンプリングし、データをパケットにして基地局に送信します。

学ぶこと

- この演習はTinyOSとnesCのプログラミングに慣れるためにあります。

MyApp_Timer（tutorials/lesson_1）アプリケーションとの違い

- 次のセンサボードを使用して照度を取得します:
  MTS300/310 あるいは MDA100
- Moteのシリアルポート（UART）と無線を使用して基地局にセンサデータを送信します。
- センサをサンプルしたときに緑のLEDが点滅します。
- センサデータのメッセージが基地局に正常に送信されたときに黄のLEDが点滅します。
- 必要に応じてコンパイルとデバッグがあります。

MyApp_Sensor 実習 – アプリケーションフォルダの作成

アプリケーションフォルダ（ディレクトリ）はトップレベルのアプリケーションコードと関連ファイルが保存されています。

1. ***/MoteWorks/apps/tutorials/*** ディレクトリに移動します。
2. MyApp_Sensorという名前でサブフォルダ（ディレクトリ）を作成します。
3. ***/MoteWorks/apps/tutorials/lesson_2***フォルダの全てのファイルをMyApp_Sensorフォルダにコピーします。
   cp –r を使ってもよい

MyApp_Sensor 実習 – アプリケーションフォルダの作成

新しいこと

- sensorboardsApp.hファイル

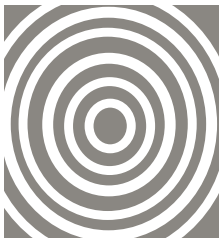
何のために使用されるか？

- パケット構造の定義
  - XSensorヘッダを定義します。
  - センサデータペイロードを定義します。
  - シリアルデータストリームのバイトが何を意味するのか理解出来ます。
- クリティカルフィールドのデフォルト値を定義
  - SENSOR_BOARD_ID
  - 送信したアプリケーションが何であるかXServeが特定するためにパケットに“Tags”をつけます。
  - センサデータパケットはXServeが適切なデータベースのテーブルかファイルに格納します。

実習: MyApp_Sensor

ステップ

- Makefile
- Makefile.component
- アプリケーションのトップレベルコンフィグレーション
- トップレベルモジュール
- アプリケーションのコンパイルとMoteにインストール
- nesCの自動ドキュメンテーション



MyApp_Sensor – Makefile と Makefile.component

Makefileにパラメータの保存

ファイル名	Makefile
ファイルの種類	All files (“.”)
ファイル形式	ファイル形式は変更しない

Makefile.componentにパラメータの保存

ファイル名	Makefile.component
ファイルの種類	All files (“.”)
ファイル形式	ファイル形式は変更しない

MyApp_Sensor – Makefile.component

センシングアプリケーションを作成するときはMakefile.componentファイルで使用するセンサボードを指定してください。

一般的な記述

```
SENSORBOARD=<sensorboard>
```

MyApp_Sensor用, `<sensorboard>` = `mts310cb`

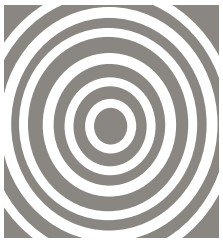
この行は何ですか？

- MTS310センサボードのセンサにアクセスするのに必要なTinyOSコンポーネントをリンクするようにnesCコンパイラに伝えます。
- MTS310センサボードのコンポーネントは***/MoteWorks/tos/sensorboards/mts310(cb)***フォルダにあります。
- 注意: ***/MoteWorks/tos/sensorboards*** のサブフォルダに他のセンサボードのコンポーネントが同様にあります。

実習: MyApp_Sensor

ステップ

- Makefile
- Makefile.component
- アプリケーションのトップレベルコンフィグレーション
- トップレベルモジュール
- アプリケーションのコンパイルとMoteにインストール
- nesCの自動ドキュメンテーション



MyApp_Sensorのコンフィグレーション – 光センサのサンプリング

```
includes sensorboardApp;

/**
 * This module shows how to use the Timer, LED, ADC and Messaging
 * components.
 * Sensor messages are sent to the serial
**/
configuration MyApp {
}
implementation {
  components Main, MyAppM, TimerC, LedsC, Photo,
             GenericComm as Comm;

  Main.StdControl -> TimerC.StdControl;
  Main.StdControl -> MyAppM.StdControl;
  Main.StdControl -> Comm.Control;

  MyAppM.Timer -> TimerC.Timer[unique("Timer")];
  MyAppM.Leds -> LedsC.Leds;
  MyAppM.PhotoControl -> Photo.PhotoStdControl;
  MyAppM.Light -> Photo.ExternalPhotoADC;

  MyAppM.SendMsg -> Comm.SendMsg[AM_XSXMSG];
}
```

NEW! Photoコンポーネントは光センサを作動させるのに使用します。

NEW! GenericComm コンポーネントはシリアルポートと無線にメッセージを送信するのに使用します。

MyApp_Sensorのコンフィグレーション – 光センサのサンプリング

```
includes sensorboardApp;

/**
 * This module shows how to use the Timer, LED, ADC and Messaging
 * components.
 * Sensor messages are sent to the serial port
**/
configuration MyApp {
}
implementation {
  components Main, MyAppM, TimerC, LedsC, Photo,
             GenericComm as Comm;

  Main.StdControl -> TimerC.StdControl;
  Main.StdControl -> MyAppM.StdControl;
  Main.StdControl -> Comm.Control;

  MyAppM.Timer -> TimerC.Timer[unique("Timer")];
  MyAppM.Leds -> LedsC.Leds;
  MyAppM.PhotoControl -> Photo.PhotoStdControl;
  MyAppM.Light -> Photo.ExternalPhotoADC;

  MyAppM.SendMsg -> Comm.SendMsg[AM_XSXMSG];
}
```

Photoコンポーネントは光センサをオン／オフするためのStdControlインタフェースとADCポートからセンサ値をサンプリングするためのADCインタフェースを実装します。

MyApp_Sensorのコンフィグレーション – 光センサのサンプリング

```
includes sensorboardApp;

/**
 * This module shows how to use the Timer, LED, ADC and Messaging
 * components.
 * Sensor messages are sent to the serial port
**/
configuration MyApp {
}
implementation {
  components Main, MyAppM, TimerC, LedsC, Photo,
             GenericComm as Comm;

  Main.StdControl -> TimerC.StdControl;
  Main.StdControl -> MyAppM.StdControl;
  Main.StdControl -> Comm.Control;

  MyAppM.Timer -> TimerC.Timer[unique("Timer")];
  MyAppM.Leds -> LedsC.Leds;
  MyAppM.PhotoControl -> Photo.PhotoStdControl;
  MyAppM.Light -> Photo.ExternalPhotoADC;

  MyAppM.SendMsg -> Comm.SendMsg[AM_XSXMSG];
}
```

- Photo.PhotoStdControl (光センサのStdControlインタフェース)へ MyAppM.PhotoControl (StdControlインタフェース)
- Photo.ExternalPhotoADC (光センサのためのADCインタフェース)へ MyAppM.Light (ADCインタフェース)

MyApp_Sensorのコンフィグレーション – 光センサのサンプリング

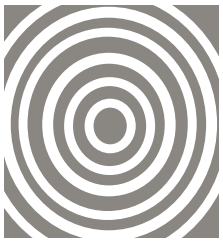
```
includes sensorboardApp;

/**
 * This module shows how to use the Timer, LED, ADC and Messaging
 * components.
 * Sensor messages are sent to the serial port
**/
configuration MyApp {
}
implementation {
  components Main, MyAppM, TimerC, LedsC, Photo,
  Main.StdControl -> MyAppM.StdControl;
  Main.StdControl -> Comm.Control;

  MyAppM.Timer -> TimerC.Timer[unique("Timer")];
  MyAppM.Leds -> LedsC.Leds;
  MyAppM.PhotoControl -> Photo.PhotoStdControl;
  MyAppM.Light -> Photo.ExternalPhotoADC;

  MyAppM.SendMsg -> Comm.SendMsg[AM_XSXMSG];
}
```

- これはアプリケーションのsendインタフェースにGenericCommのXSensorチャネルを繋ぎます。

実習: MyApp_Sensor

ステップ

- Makefile
- Makefile.component
- アプリケーションのトップレベルコンフィグレーション
- トップレベルモジュール
- アプリケーションのコンパイルとMoteにインストール
- nesCの自動ドキュメンテーション



MyApp_Sensor – コンフィグレーションとモジュール

MyApp.ncに保存

ファイル名	MyApp.nc
ファイルの種類	All files (“.”)
ファイル形式	ファイル形式は変更しない

MyAppM.ncに保存

ファイル名	MyAppM.nc
ファイルの種類	All files (“.”)
ファイル形式	ファイル形式は変更しない

トップレベルモジュールの作成

アプリケーションモジュールはMyAppM.ncファイルの中にあります。

MyApp_TimerのMyAppM.ncと違うところ

- この新しいモジュールはタイマが起動したときに光センサをサンプリングする機能を追加しているところがMyApp_Timerと違います。
- サンプリングが完了したときにMoteのシリアルポート(UART)にセンサメッセージを送信します。
- (他のセッションでは無線通信になります。)

nesCのキーワード – 実装

基本的な実装のキーワード	
call	コマンドの実行
signal	イベントの実行
post	実行キューにタスクを入れる
task	バックグランドで実行させる機能
includes	ヘッダファイルの挿入
競合状態の自動防止のサポート	
async	非同期で実行されるコマンドやイベントに使用
atomic	式が分割されずに(割り込まれないで)実行されることを保証
norace	nesCが検知した競合状態のワーニングを排除

MyAppM.nc – 仕様

- MTS300/310のハードウェア仕様を定義します。
- アプリケーションディレクトリにあります。

```
includes sensorboardApp;
/**
 * This module shows how to use the Timer, LED, ADC and Messaging
 * components
 * Sensor messages are sent to the serial port
 **/
module MyAppM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
    interface Leds;
    interface StdControl as PhotoControl;
    interface ADC as Light;
    interface SendMsg;
  }
}
```


MyAppM.nc – 実装

```
implementation {
  bool sending_packet = FALSE;
  TOS_Msg msg_buffer;
  XDataMsg *pack;

  /**
   * Initialize the component.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.init() {
    call Leds.init();
    call PhotoControl.init();
    // Initialize the message packet with default values
    atomic {
      pack = (XDataMsg *)&(msg_buffer.data);
      pack->xSensorHeader.board_id = SENSOR_BOARD_ID;
      pack->xSensorHeader.node_id = TOS_LOCAL_ADDRESS;
      pack->xSensorHeader.rsvd = 0;
    }

    return SUCCESS;
  }
```


nesC インタフェース -- ADC

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

ADCインタフェースは2つのコマンドで指定します:

- getData
- getContinuousData

1つのイベント

- dataReady

サーミスタセンサをサンプル

- **getDataコマンドの呼び出し**
  - これはADCインタフェースを通して光センサをサンプルする処理を開始します。
- この処理はdataReadyイベントで光センサ値を受信することで完了します。

センサのサンプルとコールバック

```
event result_t Timer.fired()
  {
    call Leds.redToggle();
    call PhotoControl.start();
    call Light.getData();
...
  async event result_t Light.dataReady(uint16_t data) {
    atomic pack->xData.datap1.light = data;
    post SendData();
    call Leds.yellowToggle();
...
```

1. Timer.fired()は最初にStdControlインタフェースを通るstart()コマンドを呼び出して光センサをオンにします。これが発生したときに赤のLEDが点滅します。
2. 次に光の値をサンプリングする処理を開始するためにADCインタフェースを通るgetData()コマンドを呼び出します。
3. そしてサンプリングが完了したときにdataReady()イベントがコールバックされます。

センサのサンプルとコールバック

```
event result_t Timer.fired()
  {
    call Leds.redToggle();
    call PhotoControl.start();
    call Light.getData();
...
  async event result_t Light.dataReady(uint16_t data) {
    atomic pack->xData.datap1.light = data;
    post SendData();
    call Leds.yellowToggle();
...
```

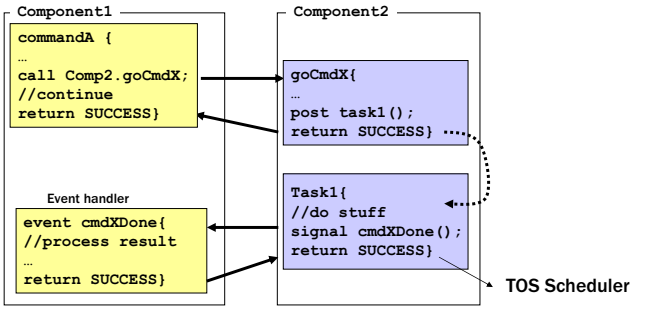
4. dataReady()イベントにより、16ビット(重要なのは10ビット)の光センサ値をメッセージパケットに格納します。
5. 最後に、センサデータを含むメッセージを送信するためにtaskをpostします。そして黄のLEDを反転します。

nesCのキーワード – 実装

基本的な実装のキーワード	
call	コマンドの実行
signal	イベントの実行
post	実行キューにタスクを入れる
task	バックグランドで実行させる機能
includes	ヘッダファイルの挿入
競合状態の自動防止のサポート	
async	非同期で実行されるコマンドやイベントに使用
atomic	式が分割されずに(割り込まれないで)実行されることを保証
norace	nesCが検知した競合状態のワーニングを排除

分割処理 – リクエストと終了シーケンス

- 非ブロッキングコマンドでオペレーション開始
- 継続/アイドル
- イベントは完了を報告



スプリット-フェーズを使用するとき

処理時間が可変

- ハードウェアI/Oの処理
  - ADC変換開始 -> 変換完了
- 低速デバイス
  - フラッシュメモリ, バッファの書き込み -> 完了

非同期や複雑な処理

- 通信スタックにメッセージを送信して、.sendDoneまで処理を続けます。

メッセージパケットの送信 – GenericComm

データパケットの送信方法

- TinyOSの通信コンポーネントGenericCommを使用します。

GenericComm の2種類のパケット送信

- UARTポート
- 無線

仕様

- 目的のノードアドレスを指定
- 予約されたノードアドレス
  - Broadcast 0xFFFF
  - UARTチャネル 0x007E
- 近隣の特定のノードIDに直接送信

MyApp.ncのコンフィグレーション

```
implementation {
  components Main, MyAppM, TimerC, LedsC, Photo, GenericComm as
Comm;

  Main.StdControl -> TimerC.StdControl;
  Main.StdControl -> MyAppM.StdControl;
  Main.StdControl -> Comm.Control;
…
  MyAppM.SendMsg -> Comm.SendMsg[AM_XSXMSG];
…
```

GenericComm（Commにエイリアス）はComm.Control（StdControl）インタフェースで接続されます。
MyAppMモジュールはComm.SendMsgインタフェースの1つのインスタンスに接続されます。
AM_XSXMSGはアクティブメッセージタイプを特定します。

- この値は複数のメッセージを見分けるのに使用します。

nesCインタフェース -- SendMsg

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

SendMsgインタフェースは1つのコマンド指定

- Send

1つのイベント

- sendDone

正しいパラメータとsendコマンドをメッセージ呼び出しに送ります。
sendDone イベントはメッセージを送信した後に受け取ります。

nesCインタフェース -- SendMsg

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

SendMsgインタフェースで使用するデータ構造はTOS_Msg構造体で定義します。

nesCデータ構造 -- TOSMsg

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

- addr – 目的のアドレス
- type – アクティブメッセージタイプ（ここではAM_XSXMSG）
- group – プログラミングのときに指定されたグループID
- length – ペイロードの長さ
- data – 可変長のペイロード領域（センサデータ）

MyApp_SensorM.nc – アプリケーションデータのペイロード

```
 command result_t StdControl.init() {
…
    // Initialize the message packet with default values
    atomic {
      pack = (XDataMsg *)&(msg_buffer.data);
      pack->xSensorHeader.board_id = SENSOR_BOARD_ID;
      pack->xSensorHeader.node_id = TOS_LOCAL_ADDRESS;
      pack->xSensorHeader.rsvd = 0;
        }
…
```

TOS_Msgのデータ領域はアプリケーションのペイロードを格納する場所です。
上記のMyAppM.ncモジュールから抜粋したコードはセンサアプリケーションのためのTOS_MSGのペイロードの初期化する方法を示しています。

MyAppM.nc – アプリケーションデータのペイロード

```
  void task SendData()
  {
    call PhotoControl.stop();

    if (sending_packet) return;
    atomic sending_packet = TRUE;

        // send message to UART (serial) a
        if (call
SendMsg.send(TOS_UART_ADDR,sizeof(XDataMsg
SUCCESS)
          sending_packet = FALSE;
…
  event result_t SendMsg.sendDone(TOS_MsgPtr msg, result_t success)
{
        call Leds.greenToggle();
        atomic sending_packet = FALSE;
…
```

sendDataタスクは最初に光センサコンポーネントのstopコマンドを呼び出しているところに注意してください。これはセンサを使用していないときにパワーを節約するためです。

メッセージの送信中(sending_packet = TRUE)はリターンします。これはsendDoneイベントがまだ呼ばれていないこと意味しますので待たなくてはなりません。

MyAppM.nc – アプリケーションデータのペイロード

```
  void task SendData()
  {
    call PhotoControl.stop();

    if (sending_packet) return;
    atomic sending_packet = TRUE;

        // send message to UART (serial) port
        if (call
SendMsg.send(TOS_UART_ADDR,sizeof(XDataMsg),&msg_buffer) !=
SUCCESS)
          sending_packet = FALSE;
…
  event result_t SendMsg.sendDone(TOS_MsgPtr msg, result_t success)
{
        call Leds.greenToggle();
        atomic sending_packet = FALSE;
…
```

目的のノードアドレスに送るには SendMsg.sendコマンドをコールします。この場合はTOS_UART_ADDRと送信するメッセージパケットのポインタを渡します。

MyAppM.nc – アプリケーションデータのペイロード

```
  void task SendData()
  {
    call PhotoControl.stop();

    if (sending_packet) return;
    atomic sending_packet = TRUE;

        // send message to U
        if (call
SendMsg.send(TOS_UART_ADDR,
SUCCESS)
          sending_packet = FALSE;
…
  event result_t SendMsg.sendDone(TOS_MsgPtr msg, result_t success)
{
        call Leds.greenToggle();
        atomic sending_packet = FALSE;
…
```

最後にSendMsg.sendDoneイベントはパケットを送信したら呼び出されます。再びタイマが起動したときは、今までの処理を最初から行う準備が出来ています。

nesCのキーワード – 実装

基本的な実装のキーワード	
call	コマンドの実行
signal	イベントの実行
post	実行キューにタスクを入れる
task	バックグランドで実行させる機能
includes	ヘッダファイルの挿入
競合状態の自動防止のサポート	
async	非同期で実行されるコマンドやイベントに使用
atomic	式が分割されずに(割り込まれないで)実行されることを保証
norace	nesCが検知した競合状態のワーニングを排除

atomic キーワード

atomic キーワードは処理に割り込みが入るのを防ぐために使用します。

- 競合状態を防止

使用する時

- asyncイベントハンドラでグローバル変数を更新するために必要です。
- 他のファンクションやタスクで変数を参照するときにatomicブロックを使用しなければなりません。

nesCコンパイラはグローバル変数を使用しているときにwarningメッセージを出します。

- 例:

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


共有ステートとatomic

atomic キーワード:

- 全ての割り込みを無効にします。
- atomicに続く{...}で囲んだブロック全体に効きます。
- セマフォでリソースをロックするのに役に立ちます。
- *注意して使用してください。*

Atomic ブロック

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

コンパイル:

```
    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 構文: Tinyos-1.x/tos/system/TimerM.nc

```
command result_t Timer.stop[uint8_t id]() {

        if (id>=NUM_TIMERS) return FAIL;
        if (mState&(0x1L<<id)) { // if the timer is running
           atomic mState &= ~(0x1L<<id);
           if (!mState) {
               setIntervalFlag = 1;
           }
            return SUCCESS;
        }
        return FAIL; //timer not running
    }
```


レビュー – パラメータ付インタフェース (1 of 2)

同じコンポーネントを複数で利用する場合

- 1つのTimerコンポーネントに対して多くの利用者が違うイベントを要求します。
- Timerが起動したときにどのコンポーネントに合図しますか?

TinyOSの扱い

- コンポーネントインタフェースインスタンスを複数持てるようにします。
- パラメータでそれらのうちの1つを特定します。

レビュー – パラメータ付インタフェース (2 of 2)

パラメータ付インタフェースで、コンポーネントは1つのインタフェースにつき複数のインスタンスが提供出来ます。

パラメータの番号(インデックス)はコンパイル時の値によって設定されます。

```
provides interface Timer[uint8_t id];
```

インスタンスの総数は実装により制限されます。

nesC コンセプト – ユニークなインスタンス

インスタンスパラメータはどのようにして使用しますか？
`unique("astring")`を使用

- 引数で与えられた文字列から8ビットの識別子を作成します。
- 複数のコンポーネントが次のように書くと
  `timer[unique("Timer")]`
  それぞれのタイマ設定に対応するシグナルを受け取ることが保証されます。

全てのコンポーネントが同じ"`string`"を使用しなければなりません。

nesC コンセプト – Timer パラメータ付インタフェースのワイアリング

パラメータ付インタフェースを提供

Tinyos-1.x/apps/CntToLedsandRfm.nc

```
configuration CntToLedsAndRfm {
}
implementation {
  components Main, Counter, IntToLeds,
IntToRfm, TimerC;

  Main.StdControl -> Counter.StdControl;
  Main.StdControl -> IntToLeds.StdControl;
  Main.StdControl -> IntToRfm.StdControl;
  Main.StdControl -> TimerC.StdControl;
  Counter.Timer->TimerC.Timer[unique("Timer")];
  IntToLeds <- Counter.IntOutput;
  Counter.IntOutput -> IntToRfm;
}
```

ユニークなTimerの番号を生成

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

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

implementation {
uint32_t mState;
uint8_t queue_size;
uint8_t queue[NUM_TIMERS];
```


ファンアウト（Fan-Out）

コンポーネントは複数に繋がれます。
コマンドやイベントは繋がれたコンポーネント間を流れます。
呼び出しの順番は保証されません。

Tinyos-1.x/apps/CntToLedsAndRfm.nc

```
configuration CntToLedsAndRfm {
}
implementation {
  components Main, Counter, IntToLeds, IntToRfm, TimerC;
  Counter.IntOutput -> IntToLeds;
  Counter.IntOutput -> IntToRfm;
}
```


nesCのキーワード – 実装

基本的な実装のキーワード	
`call`	コマンドの実行
`signal`	イベントの実行
`post`	実行キューにタスクを入れる
`task`	バックグランドで実行させる機能
`includes`	ヘッダファイルの挿入
競合状態の自動防止のサポート	
`async`	非同期で実行されるコマンドやイベントに使用
`atomic`	式が分割されずに（割り込まれないで）実行されることを保証
`norace`	nesCが検知した競合状態のワーニングを排除

async 対 sync

`async` コード

- 少なくとも1つの割り込みハンドラから呼び出されるコードです。

`sync` コード

- タスクからのみ呼び出されるコマンド、イベント、あるいはタスクです。

非同期コードでの共有ステート（メモリ）の更新は潜在的なデータアクセス競合です。

- nesCは`async`キーワードで宣言されなかった非同期なコードの`command`や`event`に対してのエラーを出します。
- これによって割り込みハンドラ内で安全に実行出来ないコードが不注意で呼ばれないように出来ます。

非同期イベントは分割処理を使用して出来る限り早く処理を終え、他の割り込み処理を行えるようにしなければなりません。

- 遅延処理のためにタスクを`Post`します。

async 構文

`async`属性はコマンドやイベントが非同期処理の一部に使われることを示します。
`async`処理はタスクをポストすることで処理を分割します。

- タスクは常に同期処理です。

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

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

`async`イベントでタスクをポストして処理を分割

nesCのキーワード – 実装

基本的な実装のキーワード	
`call`	コマンドの実行
`signal`	イベントの実行
`post`	実行キューにタスクを入れる
`task`	バックグランドで実行させる機能
`includes`	ヘッダファイルの挿入
競合状態の自動防止のサポート	
`async`	非同期で実行されるコマンドやイベントに使用
`atomic`	式が分割されずに（割り込まれないで）実行されることを保証
`norace`	nesCが検知した競合状態のワーニングを排除

norace キーワード

NesCコンパイラは競合条件が起こらないグローバル変数に対してもワーニングメッセージを出します。
ワーニングは`norace`キーワードを使用することで無効に出来ます。

- Example:

```
norace uint16_t voltage;
```

クイズ

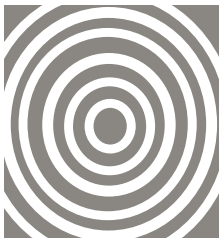
1. ハードウェア割り込みが生成するのはどんなタイプのイベントですか？
2. データアクセス競合とは何ですか？またこれを防ぐにはどのようにすれば良いですか？
3. タスクは割り込むことが出来ますか？
4. スケジューラをタスクをどのように処理しますか？
5. 分割処理とは何ですか？何故重要なのですか？
6. atomicコードについて特別なことは何ですか？

Crossbow

実習: MyApp_Sensor

ステップ

- Makefile
- Makefile.component
- トップレベルのアプリケーションコンフィグレーション
- トップレベルのモジュール
- アプリケーションのコンパイルとMoteへインストール
- nesC の自動ドキュメンテーションの実習



MyApp_Sensor – コンパイルとプログラムのインストール

1. MyApp_Sensorアプリケーションをコンパイル
   MakeXbowlocalファイルの設定
   DEFAULT_LOCAL_GROUPとRADIO_CHANNELのみ
- Moteにプログラムをインストール
- LEDのパターンを観察。どのようになっていますか？
  - 赤、緑、黄のLEDが毎秒ごとに点滅しています。

LED color	Indication
Red	1秒ごとのタイマでイベントが起動
Yellow	光センサをサンプル
Green	基地局にセンサメッセージを送信

MyApp_Sensors

4. Cygwinウィンドウで *XServe* を開始します。
   - デスクトップのCygwinのアイコンをダブルクリックしてCygwinを開きます。
   - MIB510/520を使用している場合の入力
     - `xserve -s=COM<#>`
   - MIB600を使用している場合の入力
     - `xserve -i=<hostname>:<port>`

ノードからのMTS310パケットが見えます。

nesCの自動ドキュメンテーションの実習

タスク: nesC 自動ドキュメンテーションユーティリティを使用してアプリケーションの構成を勉強してください。

1. Programmer's NotepadでMyApp_Sensorアプリケーションのコンフィグレーションを開きます。
   - tools > shellを選択して、make <*platform*> docs
2. Internet Explorerなどのwebブラウザでドキュメントファイルを開きます。
   - <MoteWorksインストールディレクトリ>¥cygwin¥opt¥MoteWorks¥doc¥nesdoc¥<*platform*>¥index.html

Crossbow

Q & A: センサアプリケーションMyApp_Sensorを作成

目標

- センサボードから光データを読むMoteファームウェアを作成
- シリアルポートにセンサデータを含むメッセージを送信