Sample Code

The following are pieces of code written in nesC, an extension of C that is designed to be used with embedded systems especially TinyOS, an operating system for wireless sensor networks. They are core code of my time synchronization that resulted in the publication "Energy-Efficient Gradient Time Synchronization for Wireless Sensor Networks" and my Master's degree thesis.

```
#include "TimeSyncMsg.h"

configuration TimeSyncC
{
  uses interface Boot;
  provides interface Init;
  provides interface StdControl;
  provides interface GlobalTime<TMilli>;
  provides interface TimeSyncInfo;
}

implementation
{
  components new TimeSyncP(TMilli);

  GlobalTime      =   TimeSyncP;
  StdControl      =   TimeSyncP;
  Init            =   TimeSyncP;
  Boot            =   TimeSyncP;
  TimeSyncInfo    =   TimeSyncP;

#ifdef TIMESYNC_TOSSIM
  components ActiveMessageC;
  TimeSyncP.RadioControl   ->  ActiveMessageC;
  TimeSyncP.AMSend         ->  ActiveMessageC.AMSend[AM_TIMESYNCMSG];
  TimeSyncP.Receive        ->  ActiveMessageC.Receive[AM_TIMESYNCMSG];
#else
  components TimeSyncMessageC as ActiveMessageC;
  TimeSyncP.RadioControl     ->  ActiveMessageC;
  TimeSyncP.AMSend           ->  ActiveMessageC.TimeSyncAMSendMilli[AM_TIMESYNCMSG];
  TimeSyncP.Receive          ->  ActiveMessageC.Receive[AM_TIMESYNCMSG];
  TimeSyncP.TimeSyncPacket   ->  ActiveMessageC;
#endif
  components HilTimerMilliC;
  TimeSyncP.LocalTime        ->  HilTimerMilliC;

  components new TimerMilliC() as TimerC;
  TimeSyncP.SendTimer ->  TimerC.Timer;

  components new TimerMilliC() as DriftCaptureDelay;
  TimeSyncP.DriftCaptureDelay ->  DriftCaptureDelay;

  components new TimerMilliC() as DriftCapturePeriod;
  TimeSyncP.DriftCapturePeriod ->  DriftCapturePeriod;

  components new TimerMilliC() as CompensateUnitDriftPos;
  TimeSyncP.CompensateUnitDriftPos ->  CompensateUnitDriftPos;

  components new TimerMilliC() as CompensateUnitDriftNeg;
  TimeSyncP.CompensateUnitDriftNeg ->  CompensateUnitDriftNeg;

  components LedsC;
  TimeSyncP.Leds  ->  LedsC;
}
```

The above piece of code shows the wiring and modular style of nesC. If you want to use a code module, you need to wire it to your code. My code mainly wires Active Message interfaces that make it easy to implement message transmission and reception over a wireless network, and timer interfaces that supports multiple fine-grained milli-second clocks.

```
event message_t* Receive.receive(message_t* msg, void* payload, uint8_t len)
{
  #ifdef TIMESYNC_DEBUG   // this code can be used to simulate multiple hopsf
          uint8_t incomingID = (uint8_t)((TimeSyncMsg*)payload)->nodeID;
          int8_t diff = (incomingID & 0x0F) - (TOS_NODE_ID & 0x0F);
          if( diff < -1 || diff > 1 )
              return msg;
          diff = (incomingID & 0xF0) - (TOS_NODE_ID & 0xF0);
          if( diff < -16 || diff > 16 )
              return msg;
  #endif

  //not currently processing and TimeSyncPacket is valid
  if  ( (state & STATE_PROCESSING) == 0
    #ifndef TIMESYNC_TOSSIM
           && call TimeSyncPacket.isValid(msg)
    #endif
    )
  {
       uint32_t localTime, globalTime;
              message_t* old = processedMsg;
                // old <-- processedMsg <-- msg
              processedMsg = msg;

  #ifdef TIMESYNC_TOSSIM
       globalTime = localTime = call GlobalTime.getLocalTime();
  #else
       globalTime = localTime = call TimeSyncPacket.eventTime(msg);   // MAC timestamping
            the packet and convert the event time to local of receiver
  #endif

     call GlobalTime.local2Global(&globalTime);
  ((TimeSyncMsg*)(payload))->receiveEventLocalTime = localTime;
     ((TimeSyncMsg*)(payload))->receiveEventGlobalTime = globalTime;
  printf("TimeSync: \t\nnodeID = %u \t\nglobalTime = %lu\n", TOS_NODE_ID, globalTime);

      state |= STATE_PROCESSING;
            post processMsg();
      return old;
      }
      return msg;
}
```

The above piece of code, processing upon reception of a new time sync packet, shows how I design the code implementation so that it can be used with different scenarios and settings. For example, if the makefile has been marked TIMESYNC DEBUG, this code will simulate multi hop wireless networks using node identification numbers as a controller. If the makefile has been marked TIMESYNC TOSSIM, a clock module for a TinyOS simulation called TOSSIM, which differs from that of sensor nodes or motes, will be used. Moreover, *post processMsg();* shows my implementation of nesC tasks. Tasks in TinyOS are a form of deferred procedure call (DPC), which enable a program to defer a computation or operation until a later time. TinyOS tasks run to completion and do not pre-empt one another. These two constraints mean that code called from tasks runs synchonously with respect to other tasks.