2014

PROOF OF CONCEPT PROTOTYPE FOR A RAILROAD PEDESTRIAN WARNING SYSTEM USING WIRELESS SENSOR NETWORKS

Puneeth Ramesh
Michigan Technological University

PROOF OF CONCEPT PROTOTYPE FOR A RAILROAD PEDESTRIAN WARNING SYSTEM USING WIRELESS SENSOR NETWORKS

By

Puneeth Ramesh

A REPORT

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

In Electrical Engineering

MICHIGAN TECHNOLOGICAL UNIVERSITY

2014

This report has been approved in partial fulfillment of the requirements for the Degree of MASTER OF SCIENCE in Electrical Engineering

Department of Electrical and Computer Engineering

Report Advisor:	*Dr. Roger M. Kieckhafer*
Committee Member:	*Dr. Zhaohui Wang*
Committee Member:	*Dr. Pasi Lautala*
Department Chair:	*Dr. Daniel R. Fuhrmann*

Table of Contents

List of Figures

Abstract

Wireless sensor network is an emerging research topic due to its vast and ever-growing applications. Wireless sensor networks are made up of small nodes whose main goal is to monitor, compute and transmit data. The nodes are basically made up of low powered microcontrollers, wireless transceiver chips, sensors to monitor their environment and a power source. The applications of wireless sensor networks range from basic household applications, such as health monitoring, appliance control and security to military application, such as intruder detection.

The wide spread application of wireless sensor networks has brought to light many research issues such as battery efficiency, unreliable routing protocols due to node failures, localization issues and security vulnerabilities. This report will describe the hardware development of a fault tolerant routing protocol for railroad pedestrian warning system. The protocol implemented is a peer to peer multi-hop TDMA based protocol for nodes arranged in a linear zigzag chain arrangement. The basic working of the protocol was derived from Wireless Architecture for Hard Real-Time Embedded Networks (WAHREN).

1. Introduction

The main objective of the project was to build a proof of concept prototype to warn the pedestrians on railroad tracks about an oncoming train. In the US, on an average 500 people are injured or killed every year due to pedestrian railroad accidents [11]. Most of these accidents happen in blind spots or due to negligence. A warning system can potentially warn victims from oncoming trains. The warning systems should be capable of functioning even in areas such as tunnels, canyons and bridges.

1.1. Network architecture

This prototype uses a Time Division Multiple Access (TDMA) based wireless sensor network protocol based on the concepts of Wireless Architecture for Hard Real-Time Embedded Networks (WAHREN) [1]. This protocol uses peer-to-peer communication. The data from each node is relayed to its neighbors until it finally reaches the end node via a multi-hop relay. The nodes are arranged as shown in figure 1. This arrangement allows each node to communicate with two of its immediate neighbors on either side, thus providing a fault tolerant way of relaying data. Even if there is a single node failure on either side of a node the data still gets transferred. The data from each node will be available at the end nodes. For example in figure 1 data from all the nodes can be read at either node 1 or node 7. The protocol uses a TDMA based approach where each node is assigned a time slot for transmitting. The nodes use their time slots to either send their own data or to relay data from neighboring nodes. The data is transferred in systolic broadcast method.

Basic functions of the production system:

- In the production system, all nodes are equipped with sensors to detect a train, while minimizing false detection due to natural phenomena or pranksters.
- Each detection is relayed peer-to-peer to all nodes
- Each node can decide whether to issue an alert based on the distance (number of hops) to the nearest detection.

The prototype was built using a eZ430-RF2500 [2] development board from Texas Instruments (TI). The development tool has a MSP430F2274 [3] microcontroller and

CC2500 2.4-GHz wireless transceiver [4]. To demonstrate the proper working of the protocol each node is equipped with a Passive Infrared motion sensor (PIR sensor). The status of each node's sensor is relayed to the end nodes. The prototype software was developed to support sixteen nodes; however production software could support many more.

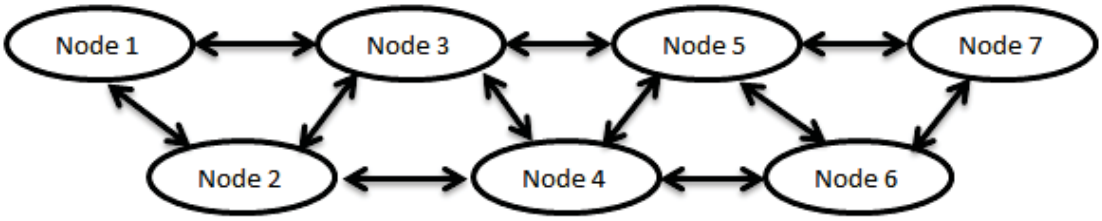


Figure 1. Figure showing the node arrangement.

The protocol is fault tolerant to single node failures at random locations. The network can still reliably route data from one end to the other even with multiple single node failures at random locations. For example, in figure 2, if nodes 3 and 6 fail, data is still reliably routed from node 1 to node 7 through nodes 2, 4 and 5.

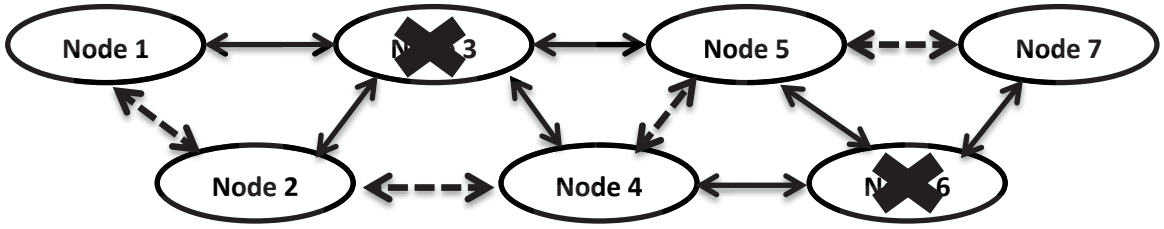


Figure 2. Network with multiple node failures.

1.2. Proof-of-Concept vs. Production System

The proof-of-concept system presented herein comprises the same basic subsystems and functions as would be needed in a production system. Although it varies considerably in the details of how the subsystems are implements, all of the principles are transferable and adaptable to the needs of a production system.

A production system would be required to detect a train, forward that data a mile or more down the track, and issue an audio/visual alert sufficient to attract a trespasser's attention well in advance of the arrival of the train. The proof of concept system is a much smaller

system, designed to detect motion of a human being in a building, forward that data up-to 100 meters down the hallway, and issue a visible alert to a researcher who is looking for it.

The system comprises four basic subsystems: a Sensor subsystem, a Software-Configurable Radio subsystem, an Annunciator subsystem, and a Power Supply subsystem. These subsystems are modular, and can be independently modified or adapted to in-the-field requirements and conditions.

1.2.1. Sensor Subsystem

The proof-of-concept system employs a simple Passive Infra-Red (PIR) sensor to detect motion of a human. By contrast, the production system would require a more sophisticated suite of sensors to detect a train, while being insensitive to false alarms, especially those induced by pranksters. Development of the production sensor suite is beyond the scope of this study.

1.2.2. Software-Configurable Radio Subsystem

The Radio subsystem comprises a low-cost, low-power micro-controller and an RF transceiver. Operation of this subsystem is completely dictated by the micro-controller software.

The definition and operation of the entire network is completely controlled by the micro-controller software, which can be easily adapted to any production system requirements. The primary focus of this study is demonstrating the feasibility of the network for peer-to-peer forwarding detection data to all nodes in the system.

The proof-of-concept transceiver operates in the unlicensed 2.4 GHz band, whereas a production model would employ RF bands licensed to the railroads.

1.2.3. Annunciator Subsystem

The proof-of-concept Annunciator is simply an LED visible to a passer-by. A Production model subsystem would need to be bright, loud, and annoying enough to catch the attention of a distracted trespasser (e.g. one who is texting on a smart-phone while wearing earphones

and a hooded parka). Development of the production annunciator is beyond the scope of this study.

1.2.4. Power Supply subsystem

Each proof-of-concept node is powered by a pair of AAA batteries. However, a production model would require a self-contained renewable power supply. Several alternatives exist, including solar, wind, and/or vibration-based sources. One advantage of this system is that the vast majority of power is consumed when a train is present. Development of a production Power Supply subsystem is beyond the scope of this study.

2. Background

The routing protocol implemented in the prototype was derived from WAHREN [1]. The WAHREN routing protocol was designed specifically for highly reliable message delivery over fixed networks and for hard real-time deadlines [1]. This protocol was designed for linear topologies like figure 1. This ideally suits the requirements for sensor placement along the railroad tracks, since they are essentially a fixed linear topology.

The WAHREN protocol was also designed to deliver messages in fixed time, where the delays for all packets are time bounded [1]. This is also a very important requirement for pedestrian warning systems. The warning message has to reach the pedestrian in time; hence delivery time of the packets is critical. The protocol uses a TDMA based approach for on time delivery of a node's packet and for fairness of medium access. WAHREN was also designed to withstand single node failures at random locations. This helps reliably route data from one end to the other even in the presence of faults [1]. The protocol also works well with gentle curves in the topology, and can be adapted to sharp corners.

3. Basic Operation

This section will describe in detail the working of the protocol. The operation of each node can be divided into two modes: time synchronization mode and normal mode. The protocol is based on TDMA, hence time synchronization is important. Each node is given its own time slot. There are sixteen time slots in one window, where each time slot is 4ms. Hence the entire window will be 64ms as shown in figure 3.

When each node is powered on, it goes through the time synchronization mode, and once it is done with time synchronization, it enters the normal mode and remains there indefinitely. During the time synchronization mode, each node synchronizes its clock to its neighbors clocks and picks a node address (which corresponds to an available time slot). When the first node is turned on it has no neighbors, so it does not make any corrections to its clock counts and just picks the first time slot. But when the second node is turned on, it first checks to see if there are transmitting nodes, picks an available time slot, and then synchronizes with the first node's clock. Once synchronized, it starts transmitting. Once node 2 starts transmitting, node 1 detects the presence of node 2 and it also makes corrections to its clock counts. This helps bring the clock counts of both the nodes as close as possible. Now, if node 3 is turned on, it does the same; it first synchronizes with both nodes 1 and 2 and then starts transmitting. All the nodes turned on follow the same procedure. Once each node synchronizes, it enters normal mode; this is when it starts transmitting.

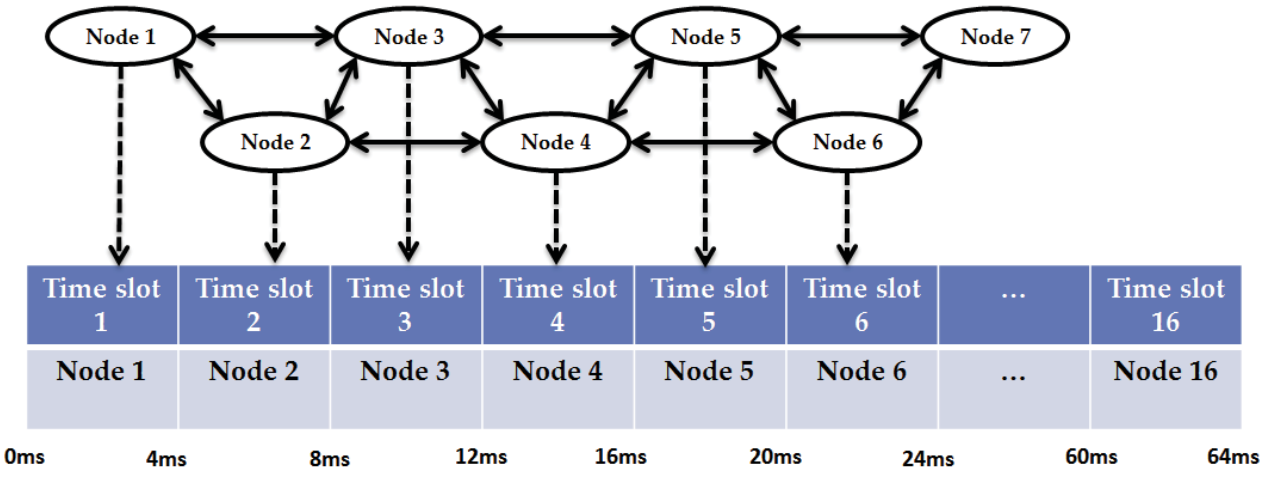


Figure 3. Figure showing one complete window.

When a node enters normal mode it does three operations:

(1) During its time slot it transmits its data or relays data received from its neighbors.

(2) During the remaining time slots it receives data from its immediate two neighbors on either side.

(3) And at the end of each window, it performs clock count corrections based on the arrival times of its neighbor's packets.

The node remains in a low-power sleep mode in between these operations. Each node will transmit its data at the middle of its time slot; for example, node one will transmit its data at 2 msec (its time slot is from 0 – 4 msec) and node 2 will transmit its data at 6 msec (its time slot is from 4 – 8msec), and so on. Figure 1, shows that each node can communicate with two of its neighbors on either side. This ensures that even if one node fails, data is still relayed to the end nodes. The data payload of each node is 4 bytes, where 2 bytes carry the upstream data and 2 bytes carry the downstream data. This ensures that the same data is available at both the ends.

The prototype consists of sixteen nodes. Each node transmits its own sensor data every sixteenth window (during Window-0) and they continuously relay data received from their

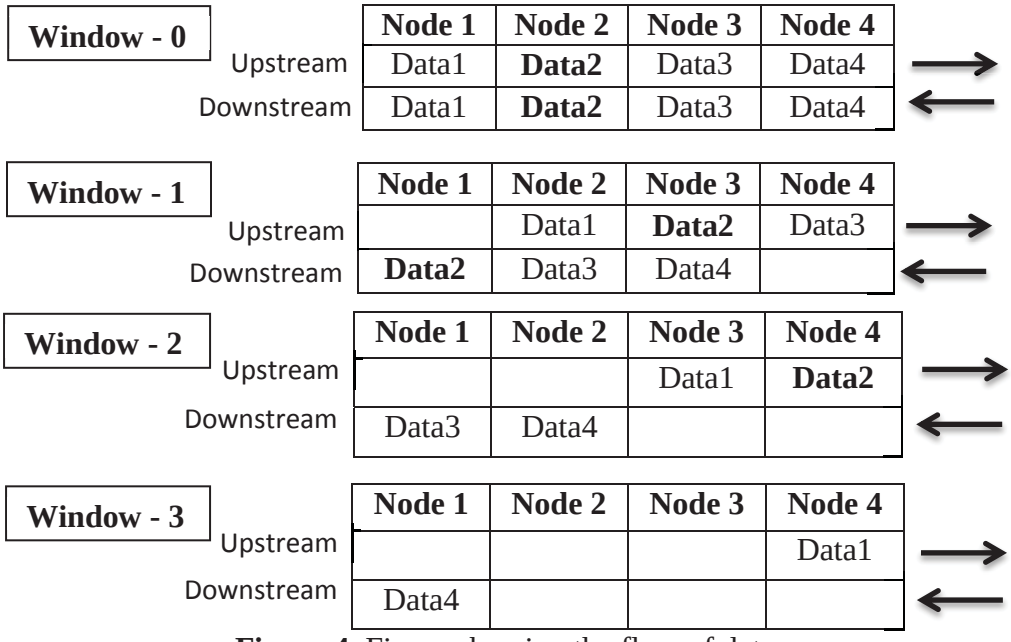


Figure 4. Figure showing the flow of data.

during the remaining fifteen windows (Window-1 to Window-15). Figure 4 illustrates how data gets relayed to the end nodes. Each node transmits its own data during window-0. This is later on relayed by its neighbors to the end nodes. For example, Node 2 transmits its data ("Data2") during its time slot in Window-0. This is later on relayed by Node 3 and Node 1 in Window-1 and by Node 4 in Window-2.

When each node receives a packet from its neighbors it records the arrival time of the packet. For example in figure 1, node 1 can receive packets from both node's 2 and 3 and records the arrival time of the packets from both nodes. The difference in the expected arrival time and the actual arrival time is used to find the error in the clock counts. The average error is used to correct the nodes clock counts. This correction is done at the end of the window. The MSP430's clock is not sourced from a crystal oscillator hence the clock generators are neither exceptionally accurate nor stable; hence the clock counts of all the nodes cannot be perfectly synchronized. There will be a non-zero offset while making corrections.

In the pedestrian warning system, each and every node is equipped with a sensor to detect the presence of a train. As soon as a train is detected the warning signal is relayed ahead of the train as shown in the figure 5. The warning signal is forwarded to ±N nodes from the detection; the distance to forward the alert can be dependent on the speed of the train. The distance to forward the alert should also provide the pedestrian enough time to move away from the tracks. In the image shown in figure 5 the warning signal reaches the pedestrian before the train does, providing the pedestrian enough time to move away from the tracks.

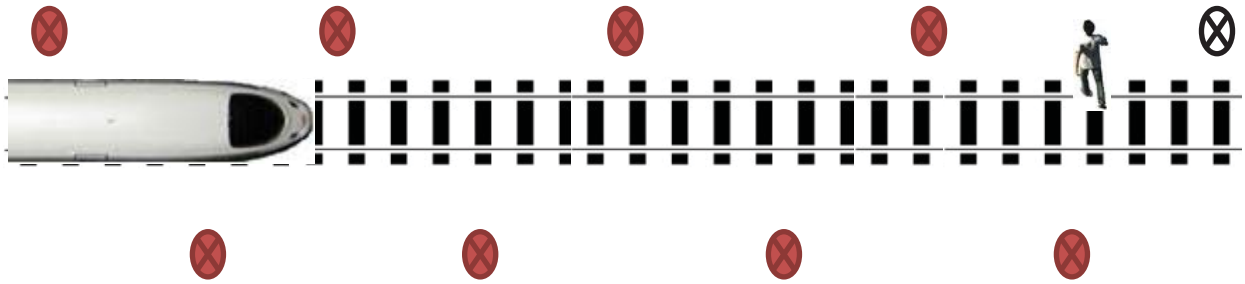


Figure 5. Operation of railway pedestrian warning system.

4. Hardware implementation

4.1. Hardware used:

1. EZ430-RF2500 development board [2].
    - MSP430F2274 microcontroller [3].
    - CC2500 2.4-GHz wireless transceiver [4].
2. PIR Sensor (#555-28027) from parallax [16].
3. LED for indication.

4.2. Node's hardware design

This section describes in detail the hardware implementation of the prototype. The eZ430-RF2500 development board was used to implement the controller and for wireless communication. The eZ430-RF2500 is an MSP430 wireless development tool. The development board features a MSP430F2274 microcontroller and CC2500 2.4-GHz wireless transceiver [2]. The IAR embedded workbench [17] was used to program and debug the MSP430F2274 microcontroller. The schematic of the prototype is shown in Figure 6 and actual images of the prototype's top and front views are shown in Figure 7.

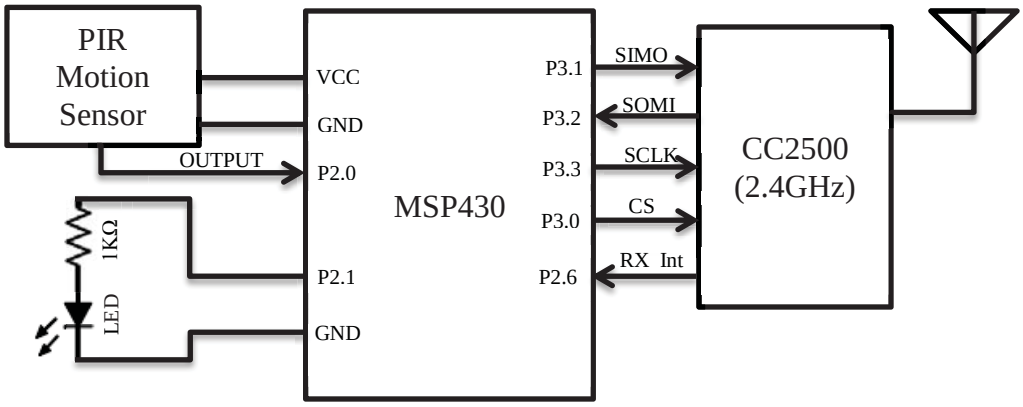


Figure 6. Block diagram of the hardware implementation.

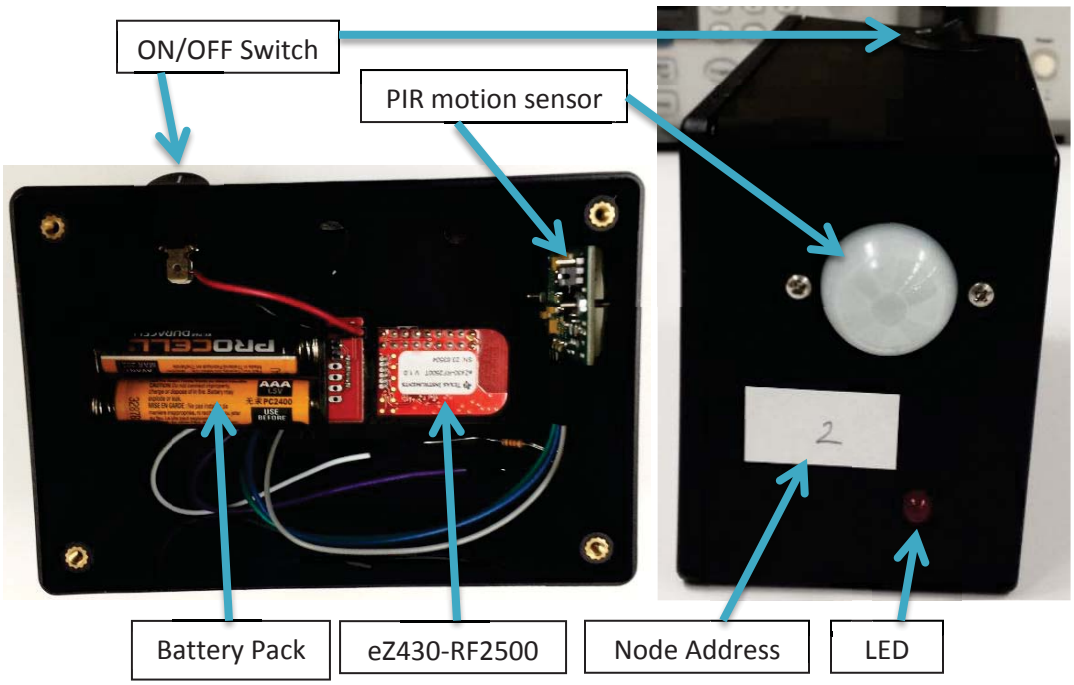


Figure 7. Top and front views of the prototype node.

The MSP430F2274 is 16-bit RISC architecture rated to operate between 1.8V and 3.6V. The prototype is powered by 2 AAA batteries. The MSP430's internal clock can be configured to run at speeds up to 16 MHz [3]. For our application the master clock (MCLK) is configured to run at 8 MHz. The MCLK is the clock source for the CPU and a sub-master clock (SMCLK), configured to run at 1 MHz. SMCLK is the clock source used for the timers and for SPI communication with the CC2500. The timer is configured in UP mode. It is programmed to generate an interrupt at the end of each window, which is 64 msec. The timer is also used to trigger an interrupt during the node's transmit time slot. The node transmits its data during this interrupt. MSP430 external port pin P2.0 is used to read the sensor status, and port pin P2.1 is used to toggle the external LED, to signal an alert.

The MSP430 uses SPI to communicate with the CC2500 transceiver. Pins (P3.0 to P3.3) are configured for SPI communication. P3.0 is used as chip select, P3.1 is used as slave in master out (SIMO), P3.2 is used as slave out master in (SOMI) and P3.3 is used as clock source (SCLK) from master to slave. Pins P3.0 to P3.3, P2.6 and P2.7 are internally connected to the CC2500. The SPI's clock source is derived from the SMCLK. This is further divided by 2 resulting in an SPI data rate of 500 Kbps. Pins P2.6 and P2.7 are

connected to the CC2500's GDO0 and GDO2 pins. These pins are user configurable pins, used to configure the CC2500 to generate an interrupt when a packet is received. Once SPI is configured, the CC2500 can be configured with the required settings.

The CC2500 is configured to transmit at a data rate of 250 Kbps. The CC2500 transmits in the 2.4 GHz frequency band. The base frequency starts from 2.433 GHz. The channel spacing is configured to 199.95 Khz. The frequency has to be picked carefully, since Wi-Fi also uses the same frequency band. For our demonstration specifically channel number 89 was used. This corresponds to a carrier frequency of 2450.79 MHz.

Carrier frequency = Base frequency + (Channel spacing * Channel number)

= 2.433 GHz + (199.95 Khz * 89)

= 2450.79 MHz.

This frequency falls into the gap between Wi-Fi channels 6 and 11. The CC2500 is configured to transmit a 4 byte preamble and a 32 bit sync word before transmitting the payload. The payload transmitted is 7 bytes long. This is appended with a 16 bit CRC for error detection. The CC2500 has on-chip support for CRC handling and sync word detection [4]. Minimum shift keying (MSK) is used to modulate the data transmitted. Carrier sense is disabled, and the GDO0 pin is configured to generate an interrupt whenever a valid packet is received. The data is transmitted as a broadcast to all nodes in the neighborhood. Data transmitted with a receiver address of 0xFF is considered a broadcast. The actual address of the transmitting node is included in the payload.

4.3. Receiver node's design

The receiver node used to collect data from any node in the network consists of the MSP430F2274 and the eZ430's USB debugger card. The receiver node collects data from a targeted node and sends it to a host computer via UART (universal asynchronous receiver/transmitter). The receiver node is configured to receive packets at the same frequency and modulation as the network nodes. The receiver node is configured to send the received packets to the host over UART at 9600 bauds with no flow control, no parity check and with 1 stop bit.

5. Software implementation

5.1. Algorithms

This section will describe in detail the software development for the prototype.

5.1.1. Procedure for clock correction

The first thing that has to be done before starting to transmit is to synchronize clock counts with the neighboring nodes. This is done using the startup sync function "startup_sync()". This function makes sure that the node's clock count is as close as possible to its neighboring nodes and also assigns a node address. In this prototype, the node address is hardcoded for each node. To achieve time synchronization, each node's clock counts have to match the neighbor's clock counts as closely as possible. This is done for 16 windows. The window size is not updated now; this will be done once transmission starts. During startup sync the node will not be transmitting, it only listens to other nodes. The code for updating the clock counts to match the neighbors clock counts is shown below. The flow chart can be found in Appendix A (flow chart 4).

```
while(total_window_cnt<16)                  // Sync for 16 windows
{
  __bis_SR_register(LPM1_bits + GIE);       // Sleep till interrupt
  if(packet_received_flg)                   // Received packet is valid
  {
    if(packet_rx[0] == (node_add-1))        // Sync to the node before me
    {
      time_error = packet_rx[1];            // Grab received time

      // Calculate error with received time and expected time
      time_error -= (half_slot_size+(slot_size*(packet_rx[0]-1)));

      if( time_error > 0 )                  // If error is positive
        while(TAR<time_error);              //Wait for TAR to reach error
      TAR -= time_error;                    // Update TAR value

      if(packet_rx[2] == (node_add-1))
        window_count = 0;             // Match window count with neighbor
    }
    else if(packet_rx[0] == node_add)
    {
       startup_sync();                  // Restart startup sync
    }
    packet_received_flg = 0;                // Clear flag
  }
}
```

The “startup_sync()” function also matches the window count with its neighbor (this helps the node decide when to transmit its data). At the end of this function the node address is fixed and the clock error is as small as possible. Before the node starts transmitting just to make sure that the sync was proper, each node performs a sync check using the function “sync_check()”. This function makes sure that the startup sync went on smoothly. It also makes sure that the error was minimized before starting to transmit; this is done for 3 windows. If the error is large then the function again calls the startup sync. The code uses a function to find the minimum error; this function’s working will be discussed later. The code is shown below and the flow chart can be found in Appendix A (flow chart 5). Once done with startup sync and sync check, the node can start transmitting data.

```
while(total_window_cnt<3)              // Check for 3 windows
{
  __bis_SR_register(LPM1_bits + GIE); // Sleep till interrupt

  TI_CC_SPIStrobe(TI_CCxxx0_SIDLE);   // Initialize CC2500 in Idle mode
  TI_CC_SPIStrobe(TI_CCxxx0_SRX);     // Initialize CC2500 in RX mode

  time_error = 0;                     // Clear time error
  error_buff[0]= min_error(error_buff[0],error_buff[1],0);//Get min err
  error_buff[1]= min_error(error_buff[2],error_buff[3],2);//Get min err
  time_error = (error_buff[0] + error_buff[1])/3;// Take the average err
  if((time_error<25) && (time_error>-25))     // If error is within +-25
  {
    correct_error();                  // Correct clock errors
  }
  else
  {
    startup_sync();                   // Restart startup sync
  }
}
```

5.1.2. Procedure for normal operating mode

The node enters the normal operating mode once it is done with startup sync. It remains in this mode indefinitely. At the beginning of this mode, transmission is enabled. After that the node enters a sleep mode until it receives a packet, detects the end of the window, or begins a transmission. First, what happens when a packet is received, is explained. As soon as a packet is received, the MSP430 gets an interrupt from the CC2500 transceiver which pulls it out of sleep mode, and interrupt service routine “f_RxData_ISR()” is executed. This function collects information from the received packet and goes back to sleep. At the beginning of the interrupt service routine the arrival time of the packet is captured. This

timestamp is used to find the error in clock counts. Now the received packet is uploaded from the CC2500 via SPI communication. The packet received has the format shown in figure 8.

rx[0]	Broadcast address (0xFF)
rx[1]	Transmitting Node address
rx[2]	Upstream data (Node address + Sensor data)
rx[3]	Upstream data (Error data)
rx[4]	Downstream data (Node address + Sensor data)
rx[5]	Downstream data(Error data)

Figure 8. Table showing the received packet structure.

Once a packet is received, two things are done. First, the error in the packet arrival time with respect to the expected nominal arrival time is determined. This gives the clock offset with respect to our current clock count. This is stored in a buffer (error_buff), and once all the errors from our neighboring nodes are collected. This buffers information can be used to make clock count corrections at the end of the window. This will be explained later. A sample of how the error is calculated is shown below.

```
error_buff[2]= packet_rx[1];        // Copy timestamp

// Calculate error and store in buffer
error_buff[2] -= (HLF_SLOT_SIZE+(SLOT_SIZE*(packet_rx[0]-1)));
```

The second thing that has to be done is to store the upstream and the downstream data in a queue so that they can be forwarded during this node's transmit slot. The queue consists of three sub queues – the *current* queue carrying data to be transmitted during the current time slot, the *next* queue carrying data to be transmitted during the next time slot and the *future* queue carrying data to be transmitted during the time slot after the next time slot. Once the queue is updated and the error is calculated and stored in the error buffer, the node can go back to sleep until the next interrupt. This procedure is repeated every time a packet is received. The process of storing data in the queues as they are received is described next. The flow chart can be found in Appendix A (flow chart 2).

The shifting between queues can be better explained using the figures 9-12. Data received from the node's immediate neighbors (node address ± 1) are stored in the *next* queue as shown in figures 9 and figure 11, this will be transmitted during this node's time slot in the next window. Similarly, data received from the node's second immediate neighbors (node

address ± 2) are stored in the future queue as shown in figures 10 and figure 12, this will be transmitted during this node's time slot in the window after the next window.

If data received is from the node above (node address+1), the downstream data is copied to the queue as shown in figure 9.

Index	[0]	[1]	[2]	[3]
Current queue				
Next queue			Downstream data [0]	Downstream data [1]
Future queue				

Figure 9. Data stored in the next queue.

If data received is from the node above that (node address+2), the downstream data is copied to the queue as shown in figure 10.

Index	[0]	[1]	[2]	[3]
Current queue				
Next queue				
Future queue			Downstream data [0]	Downstream data [1]

Figure 10. Data stored in the future queue.

If data received is from the node below (node address-1), the upstream data is copied to the queue as shown in figure 11.

Index	[0]	[1]	[2]	[3]
Current queue				
Next queue	Upstream data [0]	Upstream data [1]		
Future queue				

Figure 11. Data stored in the next queue.

If data received is from the node below that (node address-2), the upstream data is copied to the queue as shown in figure 12.

Index	[0]	[1]	[2]	[3]
Current queue				
Next queue				
Future queue	Upstream data [0]	Upstream data [1]		

Figure 12. Data stored in the future queue.

The queue indices are updated at the end of every window during normal mode. The *current* queue index is redirected to the *next* queue, the *next* queue index is redirected to the

future queue and the *future* queue index is redirected to the *current* queue like a loop. This is done with the code shown below.

```
// Increment the queue indices
curr_que_idx =(curr_que_idx>=2)?((curr_que_idx+1)-3):(curr_que_idx+1);
next_que_idx =(next_que_idx>=2)?((next_que_idx+1)-3):(next_que_idx+1);
futr_que_idx =(futr_que_idx>=2)?((futr_que_idx+1)-3):(futr_que_idx+1);
```

What happens every time the transmissions interrupt arrives is described next. This interrupt is triggered every time at the middle of the nodes time slot. So every node will transmit its data during this interrupt. The data packet transmitted has the structure shown in figure 13.

packet_tx[0]	**Packet length not including the length field**
packet_tx[1]	**Broadcast address (0xFF)**
packet_tx[2]	**Node address**
packet_tx[3]	**Current queue[0]**
packet_tx[4]	**Current queue[1]**
packet_tx[5]	**Current queue[2]**
packet_tx[6]	**Current queue[3]**

Figure 13. Table showing the transmitted packet structure.

The packet shown in figure 13 is transmitted during the node's time slot. The current queue has the upstream and the downstream data that has to be relayed on to the neighboring nodes. The name of the array holding the packet is "packet_tx[]".

The data structure of the payload for both upstream and downstream is shown in figure 14. The first bit is used to send the 1-bit sensor data (motion detected or not), the next 7 bits are used for the node address and the next 8 bits (1-bit used for sign and 7-bits used for magnitude of node's clock offset) are used to send the node's current clock offset. The error is saturated to ±127 before transmitting.

Bit position	**15**	**14-8**	**7**	**6-0**
Contents	Sensor data	Node address	Error sign	Current error

Figure 14. Payload Data in upstream and downstream.

The node transmits its own data when the window count is 0 and just relays data received from the neighboring nodes for the rest of the windows. The code for this is shown below. The flow chart can be found in Appendix A (flow chart 1).

```
if(window_count == 0)
{
  packet_tx[3] = ((P2IN<<7)|node_add);            // Node address
  packet_tx[4] = error_data;                      // Current Error
  packet_tx[5] = ((P2IN<<7)|node_add);            // Node address
  packet_tx[6] = error_data;                      // Current Error
}
else
{
  packet_tx[3] = packet_queue[curr_que_idx][0];   // Transmit data from
  packet_tx[4] = packet_queue[curr_que_idx][1];   // queue
  packet_tx[5] = packet_queue[curr_que_idx][2];
  packet_tx[6] = packet_queue[curr_que_idx][3];
}
```

Once the data is transmitted the current queue's values are reset and the node goes back to sleep mode until the next interrupt.

The third type of interrupt happens at the end of every window. At the end of the window, the node is interrupted from sleep mode so that it can make corrections to the clock counts. This keeps the nodes synchronized to each other. All this is performed in the normal mode function "normal_mode()".

The average error is determined by picking up the minimum error from either side of the node's neighbors. The two minimum errors are averaged and the average of those is sent over to the error correction function "correct_error()". The code for this is shown below. The minimum error is determined using the min error function "min_error()".

```
error_buff[0]= min_error(error_buff[0],error_buff[1],0);//Get min error
error_buff[1]= min_error(error_buff[2],error_buff[3],2);//Get min error
time_error = (error_buff[0] + error_buff[1])/3;// Take the average error
correct_error();                    // Correct clock errors
```

The min error function has three input parameters – two errors from the error buffer and the index. This function then returns the minimum among the two errors passed to it. Each node first sends the errors of the nodes below its node address and then sends the errors of nodes above its node address. This gives two errors, one above and one below the node address. The node then takes the average of these by summing them and dividing the sum by 3 (we are dividing by 3 to include the current node, whose error will be zero).

The min error function's code is shown below. The errors are first checked if they are negative. If they are negative the absolute value is taken. Then the minimum error among the

two absolute values is returned. If both the errors are 0xFFF (the default value) that means no packets were received, the function will just return zero.

```
int32_t min_error(int32_t error_1, int32_t error_2, uint8_t index)
{
  if(error_1<0)                  // If first error is negative
    error_1*=-1;                 // Get ABS value
  if(error_2<0)                  // If second error is negative
    error_2*=-1;                 // Get ABS value
  if(error_1>error_2)            // If 1st error is greater than 2nd
  {
    return error_buff[index+1]; // return second error
  }
  else
  {
    if(error_1 != 0xFFF)
      return error_buff[index]; //return first error
    else
      return 0;                  //return zero
  }
}
```

Now that the average error is calculated, it will be sent to the error correction function "correct_error()". This error will be used to correct the clock counts. The error correction code is shown below. In the error correction function, the window size is updated every 16th window (the window_count resets every 16th window). This is done by adding the error to the nominal window size, and the slot size is updated, the half slot size and the interrupt trigger values to reflect the changes made to the window size. Since the window correction is done only once every 16th window the node will just correct the TAR (counter value) during the remaining time. At the end of the error correction function, the error buffer values are reset to 0xFFF. The flow chart can be found in Appendix A (flow chart 3).

```
if(window_count == 0)
{
  time_error = time_error>>1;
  window_size = 16000 + time_error;    // Add error to the window size
  slot_size = window_size >> 4;        // slot size = window size/16
  half_slot_size = slot_size >> 1;     // Half slot size= slot size/2

  TACCR0 = window_size;                // Update TACCR0 with new values
  TACCR1 = half_slot_size + (slot_size*(node_add-1))- tx_delay;
}
else
{
  if( time_error > 0 )                 // If error is positive
    while(TAR<time_error);             // wait for TAR to reach error
  TAR -= time_error;                   // Update TAR value
}
```

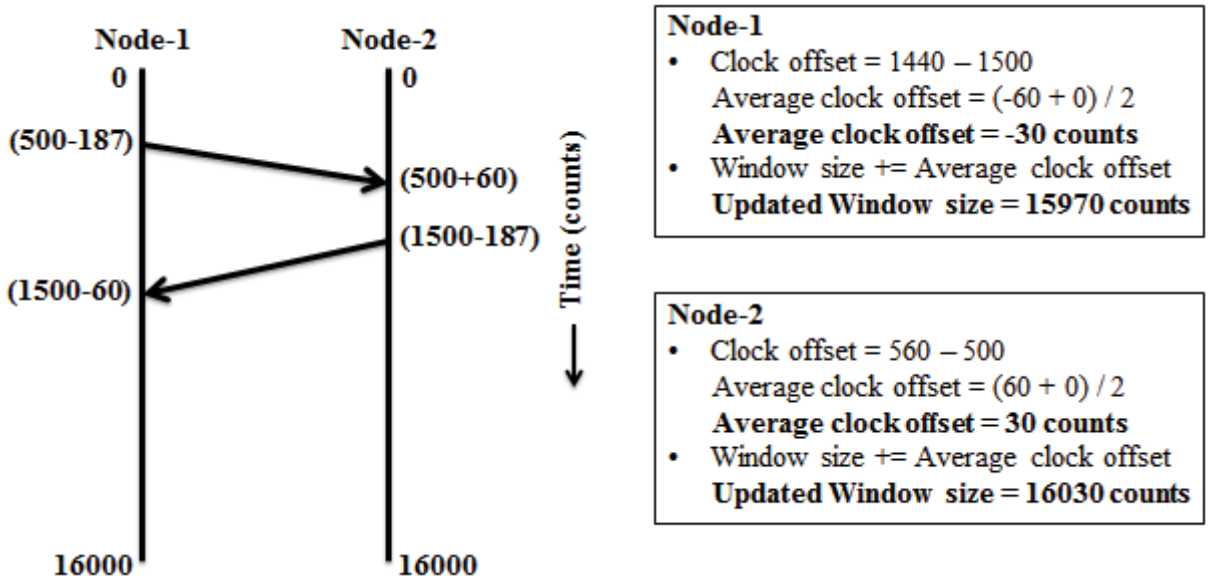


Figure 15. Clock corrections between 2 nodes.

For example, figure 15 shows clock correction procedure for 2 nodes. In this example, node-2's clock is 60 counts faster than node-1's clock. Which means when node-1's clock reaches 16000 counts, node-2's clock would have rolled over and reached 60 counts. The total transmission time is 187 counts. Each node transmits ahead of time, so that the receiver receives it at the expected arrival time. So node-1 would transmit its packet when its counter reaches (500-187) counts, this would arrive at node-2 at (500+60) counts. Similarly, node-2's packet would arrive at node-1 at (1500-60) counts. Then both the nodes calculate the clock offset as shown in the figure 15. the average clock offset is then added to the window size. The node's then count towards the updated window size in the next window.

5.2. Receiver node software implementation

This section describes in detail the receiver node's software development and the software application used to monitor the nodes' activity. The receiver node's CC2500 is also configured to generate an interrupt every time a packet is received successfully. Once the packet is received, the receiver node compares the node address of the received packet with the target node address. If the received node address matches the target node address, the rest of the packet is processed to be sent over the UART and to the USB debugger. The eZ430's USB debugger is hard coded by the manufacturer to send data at 9600 bauds [5]. Due to this limitation either upstream or downstream data can be sent over the UART, but not both.

The data to be sent over UART consists of the node's address, its sensor status and the node's corresponding clock count offset with respect to the nominal. The node address and the sensor status are masked before they are sent over UART because some of the node addresses correspond to control commands in the UART. To avoid this error, the empty bit fields are masked before transmitting. Similarly the node's corresponding error is a random value which could also match some of the control commands in the UART; hence it is split into 2 bytes (upper and lower nibble) and masked before transmitting. At the end of transmission, a new one word line character "\n" is transmitted. This is transmitted to indicate the end of transmission and also acts as a reference to separate the data packets at the computer's end. The stream flag is used to switch between upstream and downstream. The code snippet is shown below.

```
if(stream_flag == 1)
{
  while ( !(IFG2 & UCA0TXIFG)) {};// Wait while Tx Buff not empty
  UCA0TXBUF = (rx[2] | 0x60); // send byte address and sensor data
  while ( !(IFG2 & UCA0TXIFG)) {};// Wait while Tx Buff not empty
  UCA0TXBUF = ((rx[3] & 0x0F)| 0x60); // send lower half of error
  while ( !(IFG2 & UCA0TXIFG)) {};// Wait while Tx Buff not empty
  UCA0TXBUF = (((rx[3] & 0xF0)>>4)| 0x60);// send upper half of error
  while ( !(IFG2 & UCA0TXIFG)) {};// Wait while Tx Buff not empty
  UCA0TXBUF = '\n';               // send new line char
}
else
{
  while ( !(IFG2 & UCA0TXIFG)) {};// Wait while Tx Buff not empty
  UCA0TXBUF = (rx[4] | 0x60); // send byte address and sensor data
  while ( !(IFG2 & UCA0TXIFG)) {};// Wait while Tx Buff not empty
  UCA0TXBUF = ((rx[5] & 0x0F)| 0x60); // send lower half of error
  while ( !(IFG2 & UCA0TXIFG)) {};// Wait while Tx Buff not empty
  UCA0TXBUF = (((rx[5] & 0xF0)>>4)| 0x60);// send upper half of error
  while ( !(IFG2 & UCA0TXIFG)) {};// Wait while Tx Buff not empty
  UCA0TXBUF = '\n';               // send new line char
}
```

5.3. Software development of the monitoring GUI

This section gives an overview of the software used to monitor each node's activity on a host computer. The tool helps monitor the sensor data and the error in each node. This provides real time information of each node's activity. This tool is designed to monitor data from all sixteen nodes. The tool can be used to tap into any target node and collect data. This is possible as the nodes relay data in both directions. This provides the flexibility to collect

data from the first node, the last node or any node in-between. The tool was designed and programed using C#. The tool is designed to be used in tandem with a receiver node (code described in the section 6.2). The receiver node is programed to collect data from a target node; the collected data is then sent over to the tool in the computer via UART communication. The tool also sends instructions back to the receiver node. The instruction could be the desired target node or the upstream or downstream data from the desired target node. Further details on how to use the tool can be found in the user manual (Appendix B).

6. Testing and validation

This section validates the working of the prototype. To test the protocol, sixteen nodes were built. All sixteen nodes were laid out as shown in figure 1. The nodes were monitored from node 1's upstream data through the receiver node. The nodes stay synchronized for extended period of time, despite significant differences in clock frequencies. The results of the experiments conducted are described in this section.

6.1. Steady state output for all 16 nodes

The steady state output of all 16 nodes was monitored for a few minutes and the sync offset for each node with respect to nominal clock counts was buffered for the entire duration. 269 samples were accumulated, then the minimum, maximum and the average offset was calculated for each node and results were plotted as shown in figure 16.

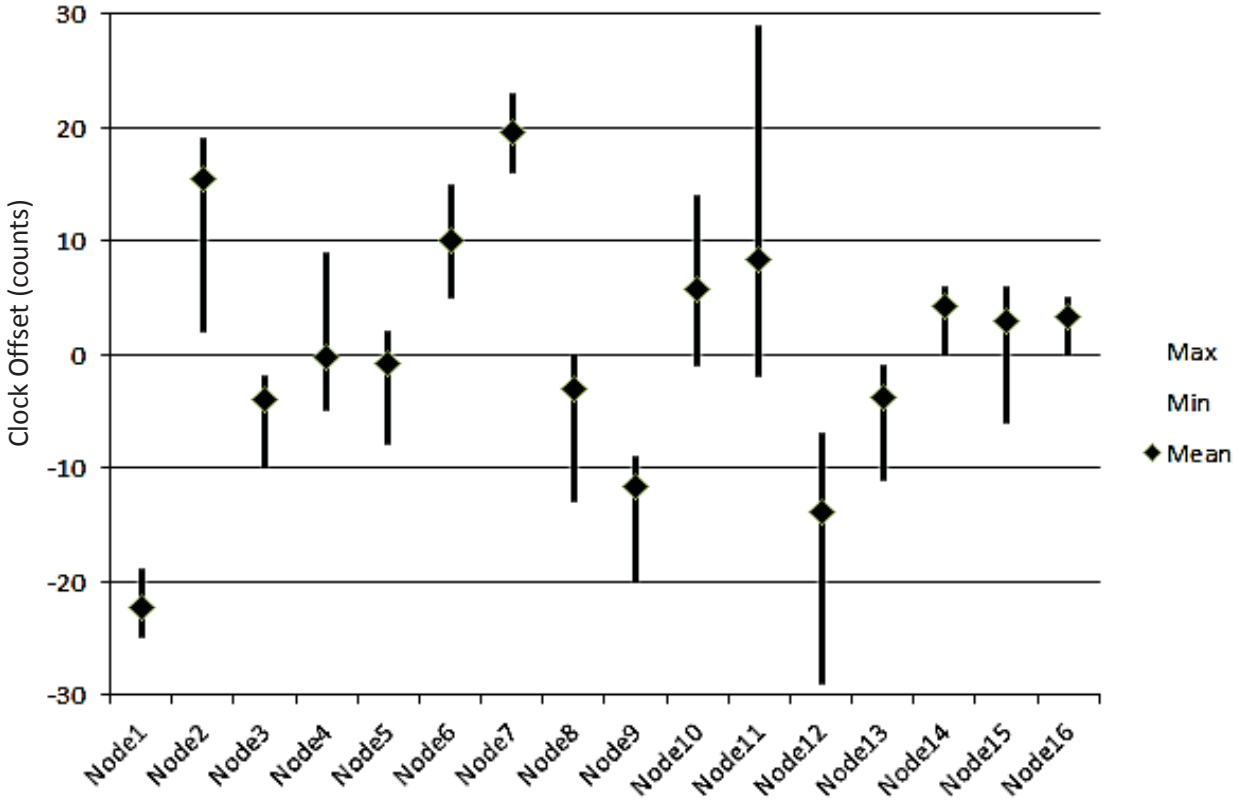
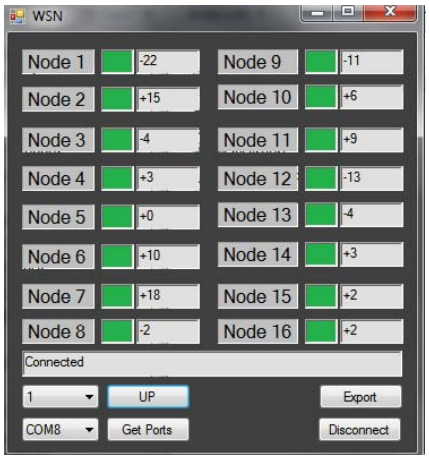


Figure 16. Minimum, maximum and average sync offsets for all 16 nodes.



	Max	Min	Mean
Node1	-19	-25	-22.3454
Node2	19	2	15.46429
Node3	-2	-10	-4.03553
Node4	9	-5	-0.33505
Node5	2	-8	-0.83673
Node6	15	5	10.04103
Node7	23	16	19.54124
Node8	0	-13	-3.02094
Node9	-9	-20	-11.6875
Node10	14	-1	5.817708
Node11	29	-2	8.383838
Node12	-7	-29	-13.8883
Node13	-1	-11	-3.79793
Node14	6	0	4.190722
Node15	6	-6	2.989637
Node16	5	0	3.289474

Figure 17. GUI showing each and every node's activity (left). Where green indicates that the node is up and running. Clock offset for each node (right).

In figure 16, the x-axis is the node number and the y-axis is the sync offset with respect to the nominal in clock counts. Each clock count is equal to 4 microseconds. The zero on the y-axis is the nominal count. Each node's minimum, maximum and the average clock count offset is shown in the graph in figure 16. It can be shown in the graph that the clock offset is bounded within a limit for each node. This clock offset is applied to the node's window size while making clock corrections. As expected, the clock offsets are non-zero as there will be constant variation from the clock source. A negative offset indicates that the node's clock is slower than its neighbors. Similarly a positive offset indicates that the node's clock is faster than its neighbors.

6.2. Network's response with node 2 turned off

This test was performed to demonstrate the impact of a missing node. Once the nodes reached steady state, node 2 was turned off. The table in figure 18 shows node 2's four neighbors offsets when node 2 was turned off and then turned back on. Node 1 and node 3 are dependent on node 2 for their clock synchronization; this can be seen in figure 18. As soon as node 2 is turned off, the magnitude of node 1's offset decreases and node 3's offset increases. Now both node 1 and node 3 are synchronizing with each other.

Node1	-19	-19	-19	-19	-15	-14	-16	-14	-14	-16	-22	-19	-19
Node2	17	18	18								26	15	19
Node3	-1	-1	-1	-1	2	4	3	4	4	3	-2	0	-1
Node4	1	2	1	2	6	6	7	4	4	5	-1	2	-1
Node5	3	1	2	2	5	5	6	6	6	7	0	2	3

Figure 18. Table showing clock offsets for the first 5 nodes.

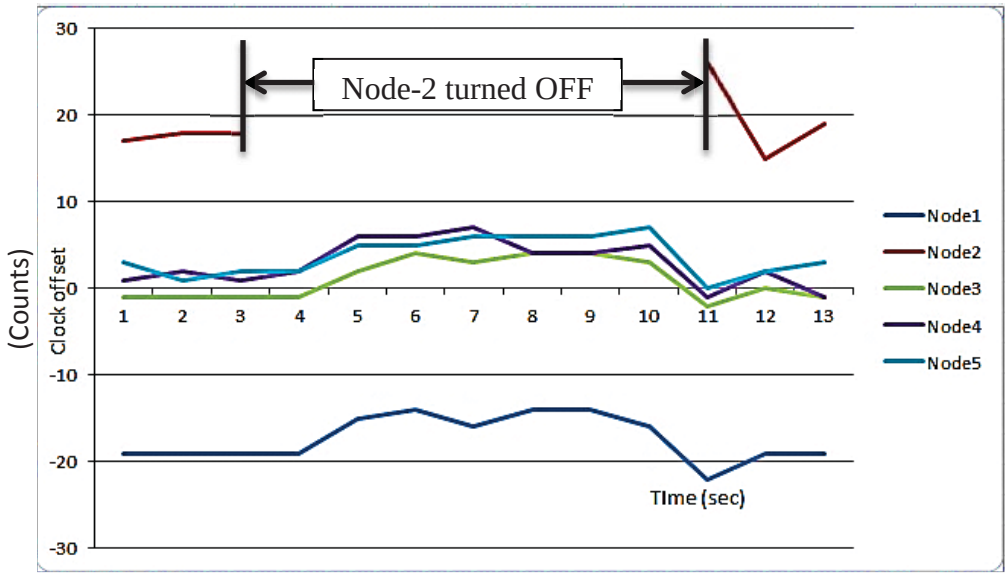


Figure 19. Graph showing the change in offsets as node 2 is turned off.

Once node 2 is turned back on, the magnitude of node 1's offset increases and node 3's offset decreases. The nodes go back to the same state as they were before. Figure 19 shows a graph of the offsets plotted from the table in figure 18. When node 2 is turned off, the offsets of node 4 and 5 also increase because node 4 is dependent on node 3 (as this is the only node to the left of node 4) and both of node 5's neighbors to the left have increased their offsets. This validates the working of the prototype for single node failure.

6.3. Network's response with node 2 and node 4 turned off

This test was performed to demonstrate the impact of a two alternate missing node. Once the nodes reached steady state, node 2 was turned off and after reaching steady state node 4 was turned off. The same results as the last test can be observed when node 2 was turned off (section 7.2). As soon as node 4 is turned off, both node 3's and node 5's offsets increase. This shows that node 3 and 5 were synchronized to node 4 while it was still on. The graph is shown in figure 21.

Node1	-18	-14	-14	-15	-14	-15	-14	-12	-12	-14	-12
Node2	17										
Node3	-1	3	3	4	3	4	3	5	5	3	5
Node4	2	5	5	6	5	6	5				
Node5	2	5	5	5	7	6	7	7	7	8	7

Figure 20. Table showing clock offsets for the first 5 nodes.

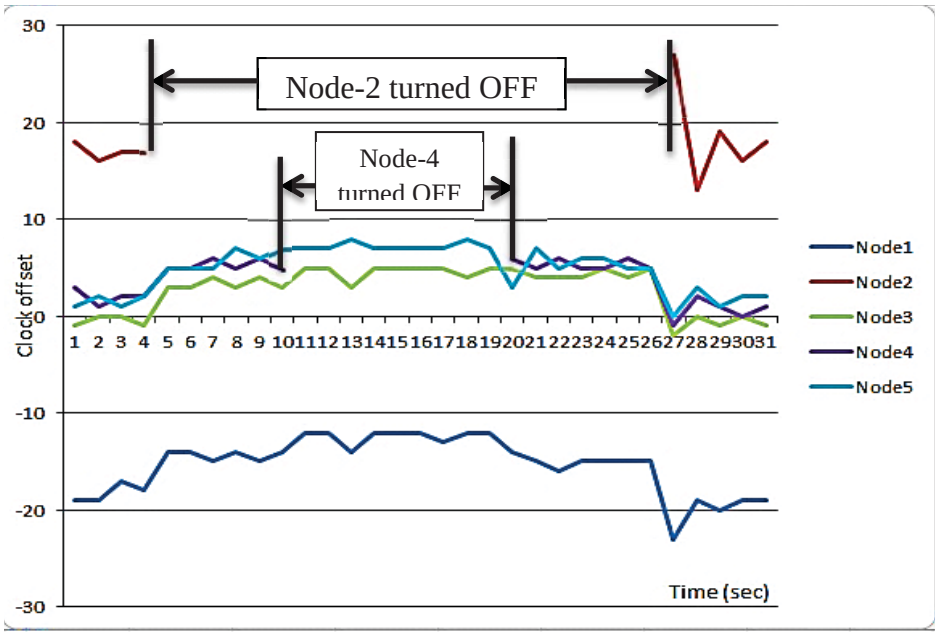


Figure 21. Graph showing the change in offsets as node 2 and 4 are turned off.

6.4. Network's response with alternate nodes turned off

This test was performed with all 16 nodes. To demonstrate the reliability of the network alternate nodes were turned off. Figure 22 shows the data received from node 1 when all even numbered nodes are turned off. This shows that even with single node failure, data is still routed to the end nodes. Figure 23 shows that time synchronization is still achieved even with alternate nodes off.

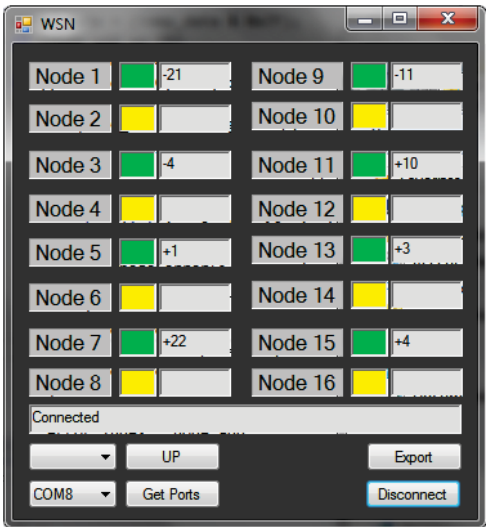


Figure 22. GUI showing the data received from node 1.

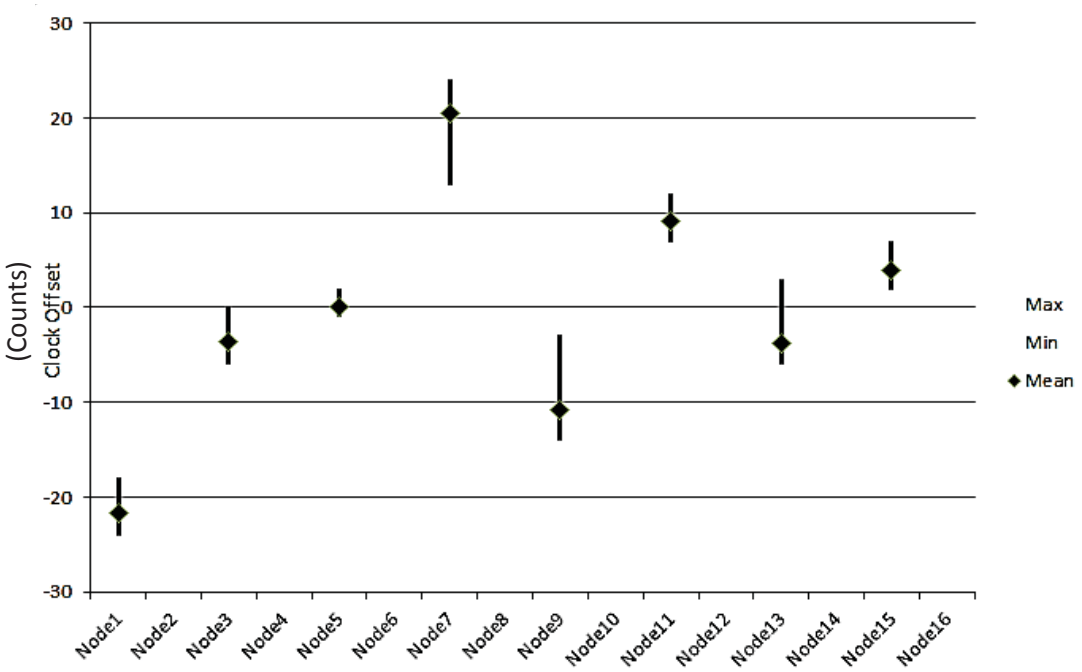


Figure 23. Min, max and average sync offsets when alternate nodes are turned off.

6.5. Network's response with 2 adjacent nodes turned off

This test was performed with all 16 nodes. When two adjacent nodes are turned off, this breaks the network into two halves. For example, in figure 24, nodes 6 and 7 were turned off; this breaks the link and all the data after node 6 is no longer available at node 1. This partitions the network into 2 sub-networks (node1 to node5 and node8 to node16). The table in figure 25 shows the loss of data once both nodes 6 and 7 are turned off.

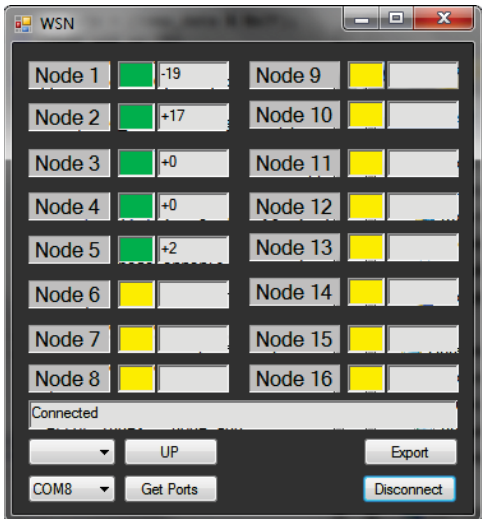


Figure 24. GUI showing the data received from node 1.

A	X	Y	Z	AA	AB	AC	AD	AE	AF	AG
Node1	-20	-22	-23	-22		-22	-22	-23	-19	-19
Node2	11	15	14	14	14	14	15	14	16	17
Node3	-2	-4	-4	-4	-4	-4		-1	0	-2
Node4	2	-1	0	0	0	-1	3	3	3	3
Node5	0	-1	0	-1	-1	-1	2	2	2	2
Node6	10									
Node7	20	20	20	21	20	20				
Node8	-1	-3	-4	-4	-5	-4				
Node9	-12	-11	-13	-11	-11	-12				
Node10	5	4	5	4	6					
Node11	10	9	9	9	8					
Node12	-14	-13	-12	-14	-15					
Node13	-5	-6	-5	-4	-6					
Node14	3	4	2	4	3					
Node15	4		3	1	3					
Node16	3	3	3	2	3					

Figure 25. Table showing clock offsets received from node 1.

7. Design constrains and compromises

Now that the system has been described in its entirety, this section discusses in detail some constrains faced while designing the prototype.

7.1. Design considerations for production unit

The prototype was built to demonstrate that the concept can be used to provide a reliable warning system for pedestrians. The prototype used PIR motion sensors to detect the presence of an object, but can be replaced with a more sophisticated detection method to detect the presence of a train, while minimizing false detections. The prototype uses LED's to notify the pedestrians, this can also be replaced with other warning systems such as lights and horns. The prototype was designed to forward alert to ±1 node; this can be extended to ±N nodes. The prototype software was designed to accommodate sixteen nodes; this can also be further extended by reusing the time slots.

The initial design of the prototype was to make the nodes pick their addresses automatically. The nodes at startup would listen to their neighbors and based on that would pick an empty time slot and an address corresponding to the time slot. This can be implemented in the production nodes because in the field the nodes can clearly hear only their neighbors. Thus they can detect empty time slots and pick those time slots. In the

prototype, a starting node could hear several other nodes due to the short distance involved. Thus, the nodes were hardcoded with their addresses and pick the time slots corresponding to their addresses.

7.2. Other design constrains

The receiver node uses a UART to communicate with the user's computer. The eZ430-RF2500's USB debugger was used to communicate with the computer. Unfortunately, the UART communication rate is fixed to 9600 baud [5]. This resulted in limiting the maximum number of bytes that could be transmitted in one time slot. At the rate of 9600 bauds it will take 3.3 msec to transmit 4 bytes over UART and the time slot for each node was 4 msec. To send both the upstream and the downstream data, 8 bytes had to be transmitted (it takes 6.6 msec to send 8 bytes). Given the fact that only 4 bytes can be sent within each time slot the GUI is limited to displaying only the upstream or the downstream data. This limitation is only when the user tries to receive data from nodes other than 1 and 16. All the nodes can be monitored from the end nodes. For node 1, only the downstream is sufficient and for node 16 only the upstream is sufficient.

Each node's data consists of the node address, sensor status and the node's clock offset. The node address and the sensor status can be packed into one byte but the clock offset had to be split into two bytes. The node's clock offset is a random number and because of that it would sometimes match the ASCII control codes, resulting in undesirable output at the receiver end. To avoid this issue the offset is split into 2 parts (upper nibble and the lower nibble). This is then packed into 2 bytes and masked before transmitting. At the end of each node's data a delimiter (new-line character) has to be sent; this is sent to help the receiver separate the node's data. All this put together is 4 bytes in length.

7.3.Choosing the proper frequency for communication

The CC2500 uses the 2.4 GHz band which is commonly used by many devices, including Wi-Fi routers. Routers commonly use channels 1 (2.401-2.423 GHz), 6 (2.426-2.448 GHz) and 11 (2.451-2.473 GHz) [7] as shown in figure 26. But some routers use channels other than the ones mentioned above. The frequency for the prototype was chosen experimentally to start at 2.4508 GHz. This is in between Wi-Fi channels 6 and 11. Due to the fact that the

channel could still be busy, carrier sense had to be disabled because this would delay the transmission, which is not acceptable in a time critical application.

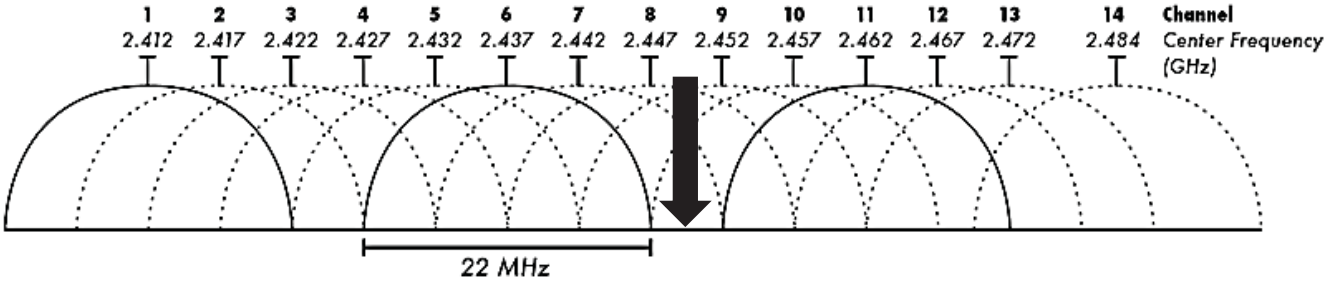


Figure 26. Wi-Fi Channel spectrum [6].

7.4. Total packet transmission time

The protocol relies on TDMA which makes time synchronization critical. Since the clock offset is determined based on the received packets arrival time the transmission delay has to be included. The transmission delay can be determined analytically as shown below.

Total transmission time:

$$T_{Tx} = T_{SPI} + T_{WTx}$$

Where:

T_{Tx} = Total transmission time.

T_{SPI} = MSP to CC2500 SPI transfer time at 500 Kbps.

T_{WTx} = Wireless transmission time at 250 Kbps.

SPI transfer time:

$$T_{SPI} = L_{Tx\_FIFO} + L_D + L_{Tx\_stb}$$

T_{SPI} = (1-byte + 7-bytes + 1-byte) at 500 Kbps.

T_{SPI} = (9-bytes * 8-bits) / 500K = 0.144 msec.

Where:

L_D = Length of payload data in bytes.

L_{Tx_FIFO} = Length of Tx FIFO register's address in bytes.

L_{Tx_stb} = Length of transmission strobe command in bytes.

Wireless transmission time:

$$T_{WTx} = L_{Pre} + L_{Sync} + L_D + L_{CRC}$$

T_{WTx} = (4-bytes + 32-bits + 7-bytes + 16-bits) at 250 Kbps

T_{WTx} = (17-bytes * 8-bits) / 250k = 0.544 msec.

Where:

L_{Pre} = Length of preamble in bytes.

L_{Sync} = Length of sync word in bits.

L_{CRC} = Length of CRC in bits.

$$T_{Tx} = T_{SPI} + T_{WTx}$$

T_{Tx} = (0.144 + 0.544) = 0.688 msec.

The above calculation does not include the time required for the CC2500 to modulate the data, CRC calculation and the propagation time in air. Experimentally, the average total time to transmit a packet was determined to be 0.748 msec. This delay is included in the packet TX interrupt. The packet is transmitted ahead of the delay time so that it arrives at the exact time expected at the receiver. In the field, propagation delay of 5.2 ms/mile has to be added to the total transmission time. The CC2500, when in receive mode, occasionally stops receiving. To overcome this issue, the receive strobe is retransmitted to the CC2500 at the end of every window.

8. Conclusion

The prototype was successfully built and demonstrated. The algorithm used for this project was also validated by the performance of the prototype. The protocol shows reliable assurance for hard real time data communication even with node failures at random locations.

The prototype can be developed into a production module with slight modifications to warn pedestrians about oncoming trains. The production model can be easily scaled to longer distances by reusing the time slots. The network deployment can be variable. For example, in areas where accidents are prone to happen, network deployment can be denser than in areas with less human activity. In places such as railway stations, bridges and curved tracks the network deployment can be denser. A denser deployment can easily catch the pedestrian's attention. In areas where there is less or no human activity and in places where there is clear visibility of oncoming trains the network deployment can be less dense.

One of the main requirements of a railway warning system is that the warning signal reaches the pedestrian well before the train does. To ensure this, the warning signal should travel faster than the train. In the prototype, it takes 1.024 seconds for the warning signal to travel from one end to the other. So if the prototype nodes are placed 50 feet apart, the warning signal travels at a speed of 499.38 mph, which is much faster than a train's speed. The warning signals speed can be improved further by reducing the time slots width or by using faster data rates between nodes. It is estimated that the propagation speed can be at-least quadrupled with minimal changes to the system software.

The prototype is designed to forward an alert to one immediate neighbor on either side. In the production model, the alert distance (number of hops from detection) can be determined based on the speed of the detected train. Thus, the alert distance can be larger for a train with higher speed and smaller for a train with lower speed. The speed of the train can be easily calculated as we know the distance between two nodes and the time it takes to travel from one node to the next. In order to avoid false detections due to track maintenance vehicles, the nodes can be programmed to receive a wireless command from the maintenance unit to inhibit detection of that vehicle, while still retaining its ability to detect oncoming trains.

In a 2nd order power chain, such as the proof-of-concept system, when 2 adjacent nodes fail, the network is partitioned into 2 sub-networks. The two sub-networks still relay data within themselves from one end to the other end. Once either of the failed nodes recovers, the two sub-networks can automatically recombine to restore the complete network.

If this level of redundancy proves insufficient, the system can be configured as a higher order power chain. For example, in a 3rd order chain, each node communicates with its 3 nearest neighbors on either side as shown in figure 27. This would require only minor changes to the software and that the nodes are placed such that each node is within communication range of its 3 immediate neighbors on either side.

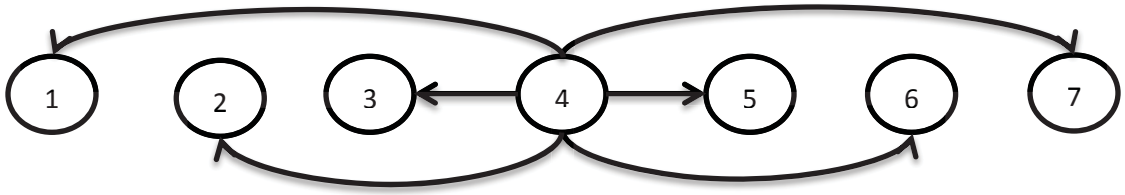


Figure 27. Node-4 communicating with 3 immediate neighbors on either side.

The total cost of each proof-of-concept unit was $58.28. The most expensive components are the ez430-rf2500 development board ($40) [18] and the PIR ($9.89) motion sensor [19]. If the nodes are placed 50 feet apart from each other, there will be 105 nodes per mile. The total hardware cost would then be $6,120 per mile. Naturally, a production system would be more costly, due to the Sensor, Annunciator, and Power Supply subsystems. Nonetheless, it is reasonable to expect the system cost to be in the range of a few tens of thousands of dollars per mile.

For future work, the nodes should be capable of predicting available node addresses at startup. Rather than hard coding the address they should be able to pick their addresses based on the information received from the neighboring nodes. It can also be used in applications where direct line of sight communication is not possible, like in tunnels, canyons and mines.

9. Reference

[1] R.M. Kieckhafer, “Design of a Real-Time Wireless Network for the Northern Pierre Auger Observatory”, the Pierre Auger Collaboration, June 28, 2010.

[2] Texas instruments, eZ430-RF2500 Development Tool User guide. SLAU227A Mixed Signal Products, 2007. , Available: http://www.ti.com/lit/ug/slau227e/slau227e.pdf.

[3] Texas instruments, *MSP430x22x2, MSP430x22x4 Mixed Signal microcontroller Datasheet*, 2012. , Available: http://www.ti.com.cn/cn/lit/ds/symlink/msp430f2274.pdf.

[4] Texas instruments, *CC2500 Low-Cost Low-Power 2.4 GHz RF Transceiver Datasheet*, 2014. , Available: http://www.ti.com/lit/ds/symlink/cc2500.pdf.

[5] Texas instruments, Application Report SLAA378D, April 2011, Available: http://www.ti.com/lit/an/slaa378d/slaa378d.pdf.

[6] *Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, IEEE Standard 802.11, 2012.

[7] *Wi-Fi / WLAN Channels, Frequencies, Bands & Bandwidths* [online]. Accessed on: October 2014, Available: http://www.radio-electronics.com/info/wireless/wi-fi/80211-channels-number-frequencies-bandwidth.php.

[8] Microsoft Visual Studio 2013, *C# Programming Guide*. Accessed on: June 2014, Available: http://msdn.microsoft.com/en-us/library/67ef8sbd.aspx.

[9] Microsoft Developer Network, *SerialPort Class*. Accessed on: June 2014, Available: http://msdn.microsoft.com/en-us/library/system.io.ports.serialport% 28v =vs.110%29.aspx.

[10] Instructables Tutorial, *Serial Port Programming With .NET*. Accessed on: June 2014, Available: http://www.instructables.com/id/Serial-Port-Programming-With-NET/step1/Set-up-and-Open-the-Serial-Port/

[11] St. Louise Post-dispatch, *Hundreds die walking the tracks each year* [online]. Accessed on: September 2014, Available: http://www.stltoday.com/news/local/metro/hundreds-die-walking-the-tracks-each-year/a rticle_b9c8bbcc-f424-559a-a0c4-bee46f8d4fe7.html

[12] The Pierre Auger Cosmic Ray Observatory [online]. Accessed on: February 2014, Available: http://www.auger.org /index.html

[13] Branislav.K, et al, “The Flooding Time Synchronization Protocol”, *SenSys '04 Proceedings of the 2nd international conference on Embedded networked sensor systems*, Pages 39-49, Nov 2004.

[14] R.M. Kieckhafer, et al, “Exploiting Omissive Faults in Synchronous Approximate Agreement”, *IEEE Trans. on Computers*, VOL. 49, NO. 10, October 2000.

[15] Saurabh G, et al, “Timing-sync Protocol for Sensor Networks”, *SenSys '03 Proceedings of the 1st international conference on Embedded networked sensor systems*, Pages 138 – 149, 2003.

[16] Motion sensor, *PIR Sensor (#555-28027) Datasheet*, Parallax Inc., 2012.

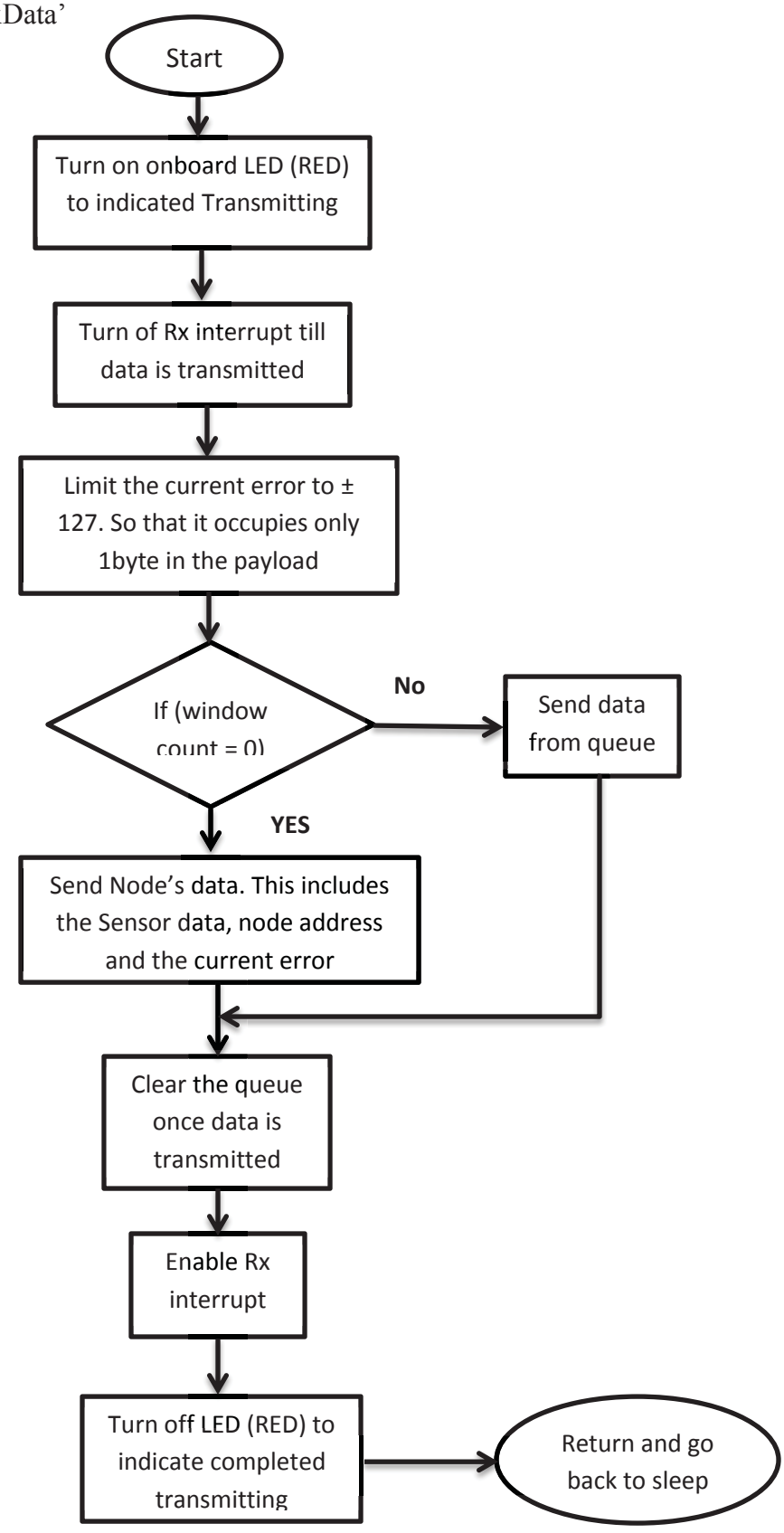
[17] IAR Embedded Workbench, *Compiler and Debugger toolchain for microcontrollers*, IAR systems.

[18] Texas instruments, eZ430-RF2500 Development Tool, Available: <http://www.ti.com/tool/ez430-rf2500t#buy>

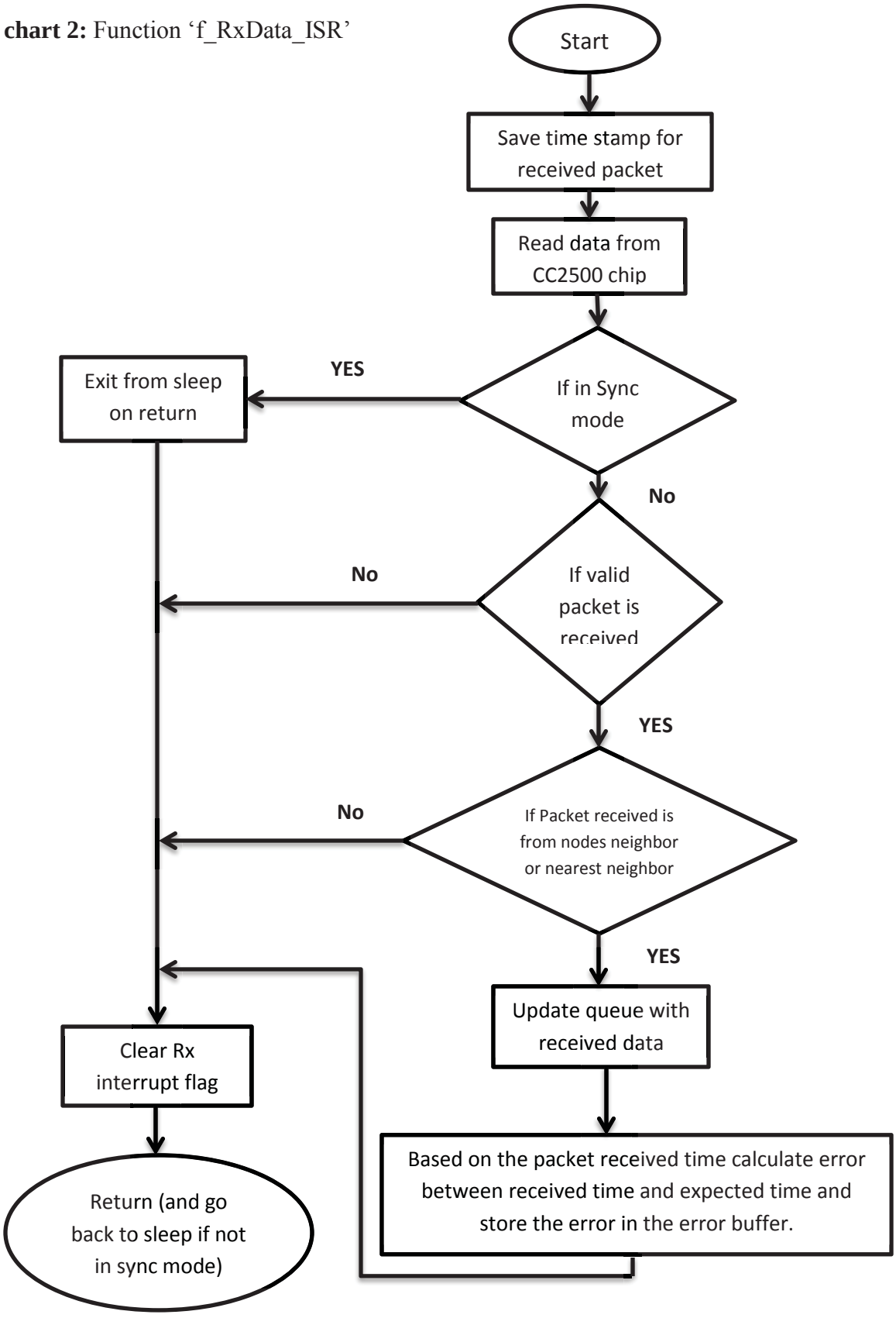
[19] Motion sensor, PIR Sensor (#555-28027), Parallax Inc., 2012. Available: <http://www.digikey.com/product-search/en?KeyWords=55528027>

Appendix A

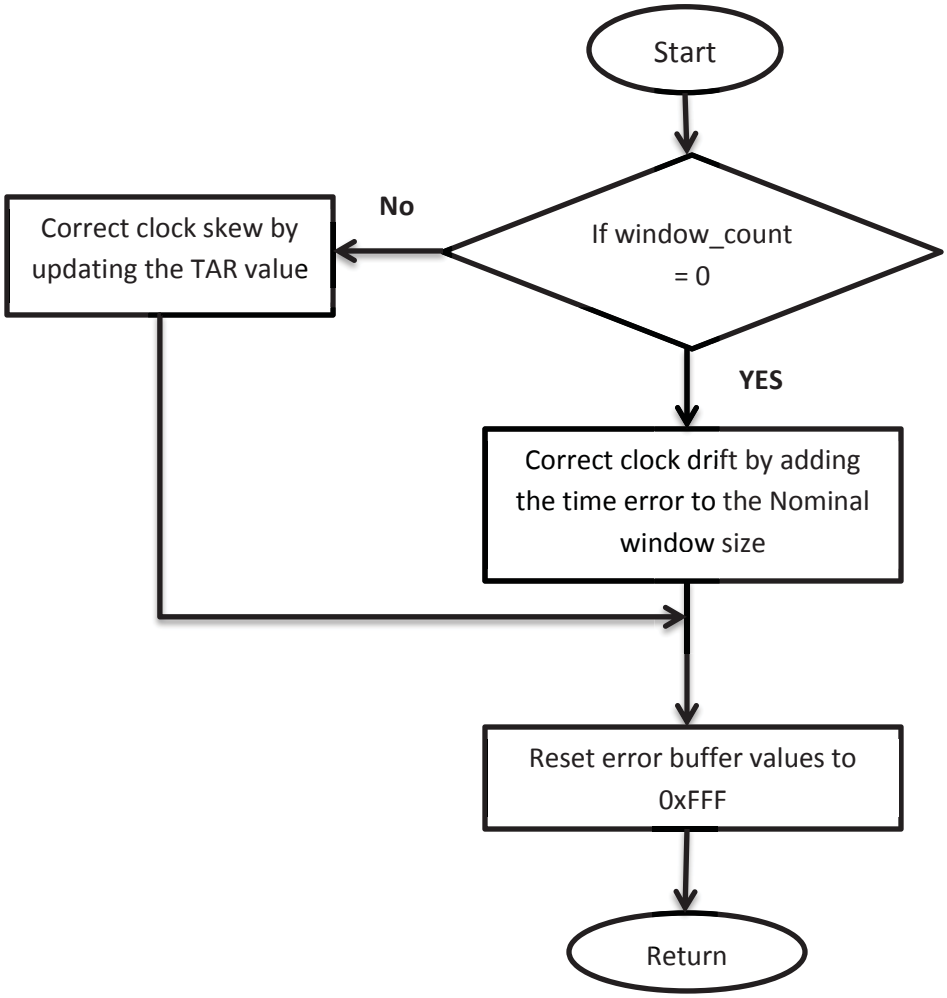
Flow chart 1: Function 'f_TxData'



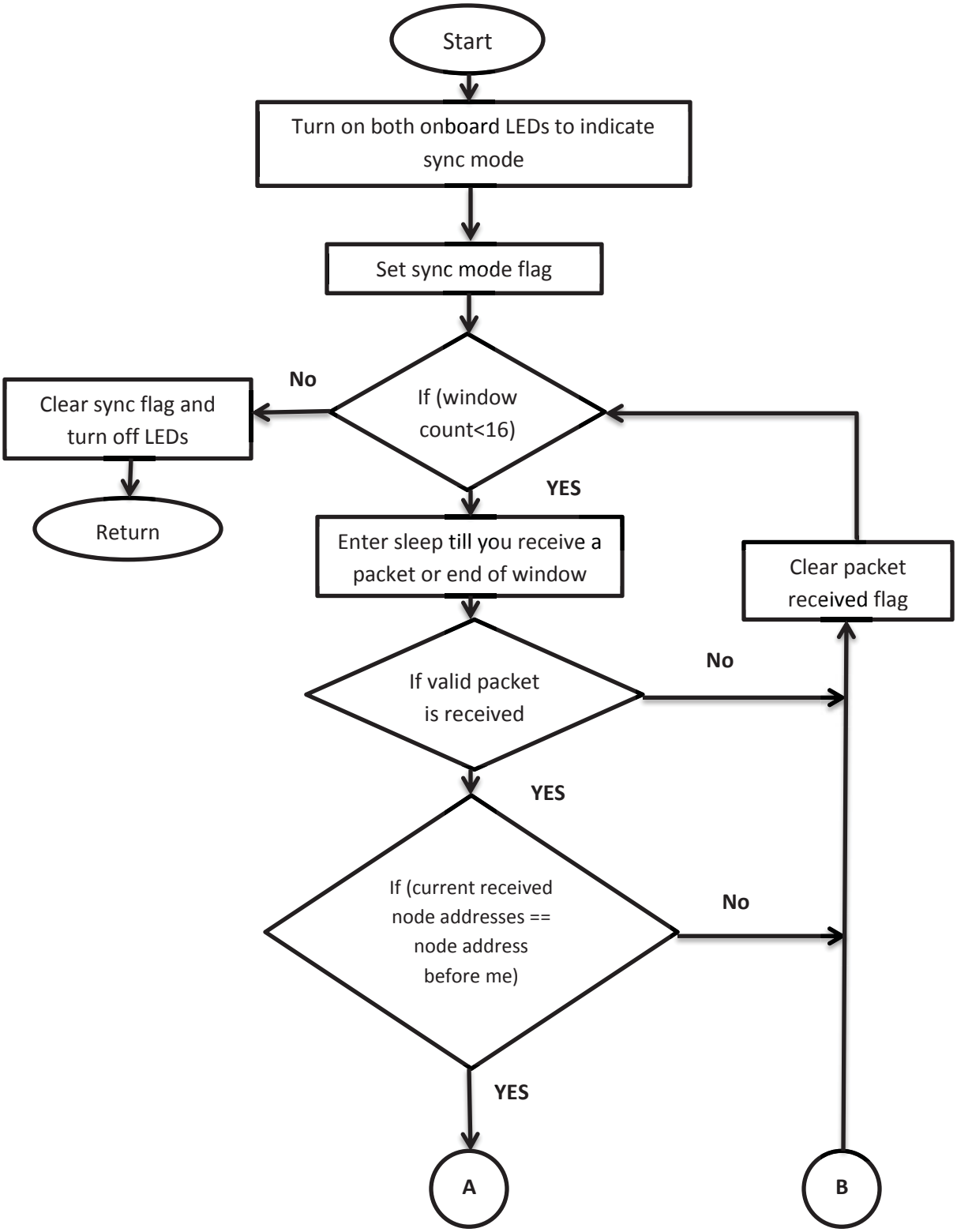
Flow chart 2: Function 'f_RxData_ISR'

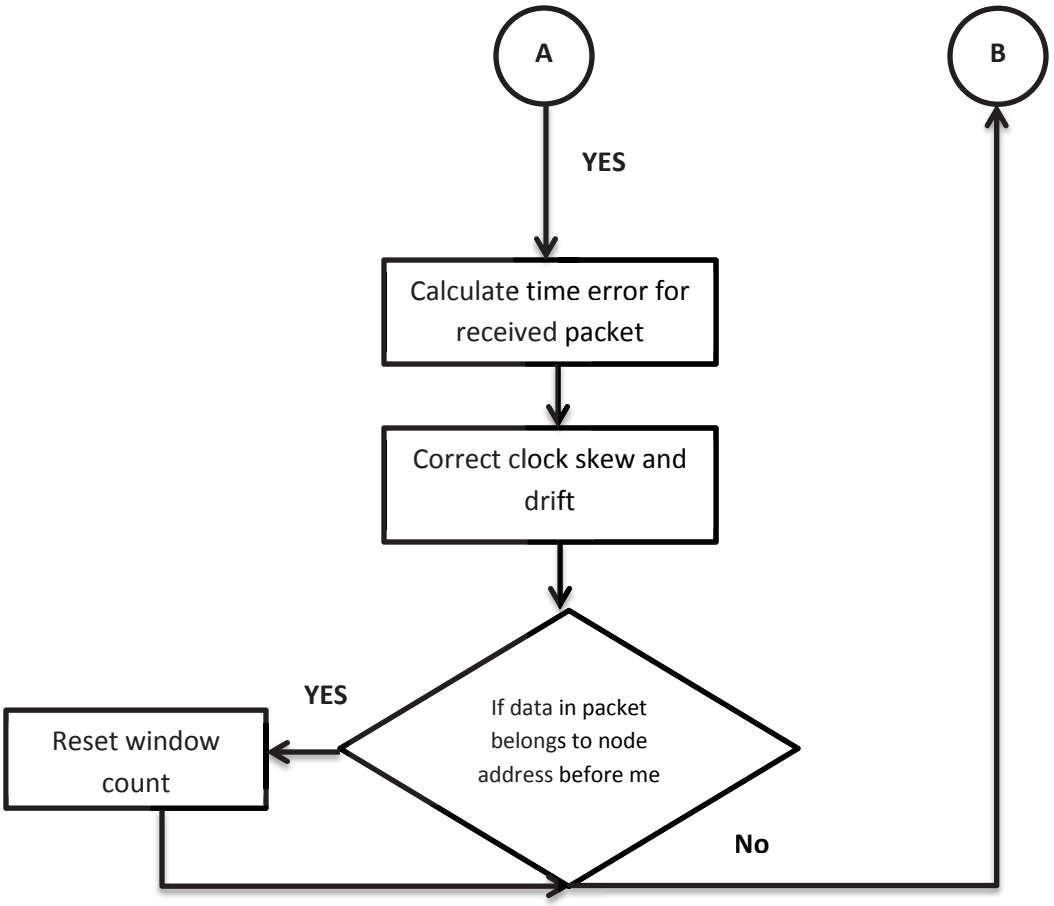


Flow chart 3: Function ‘correct_error’

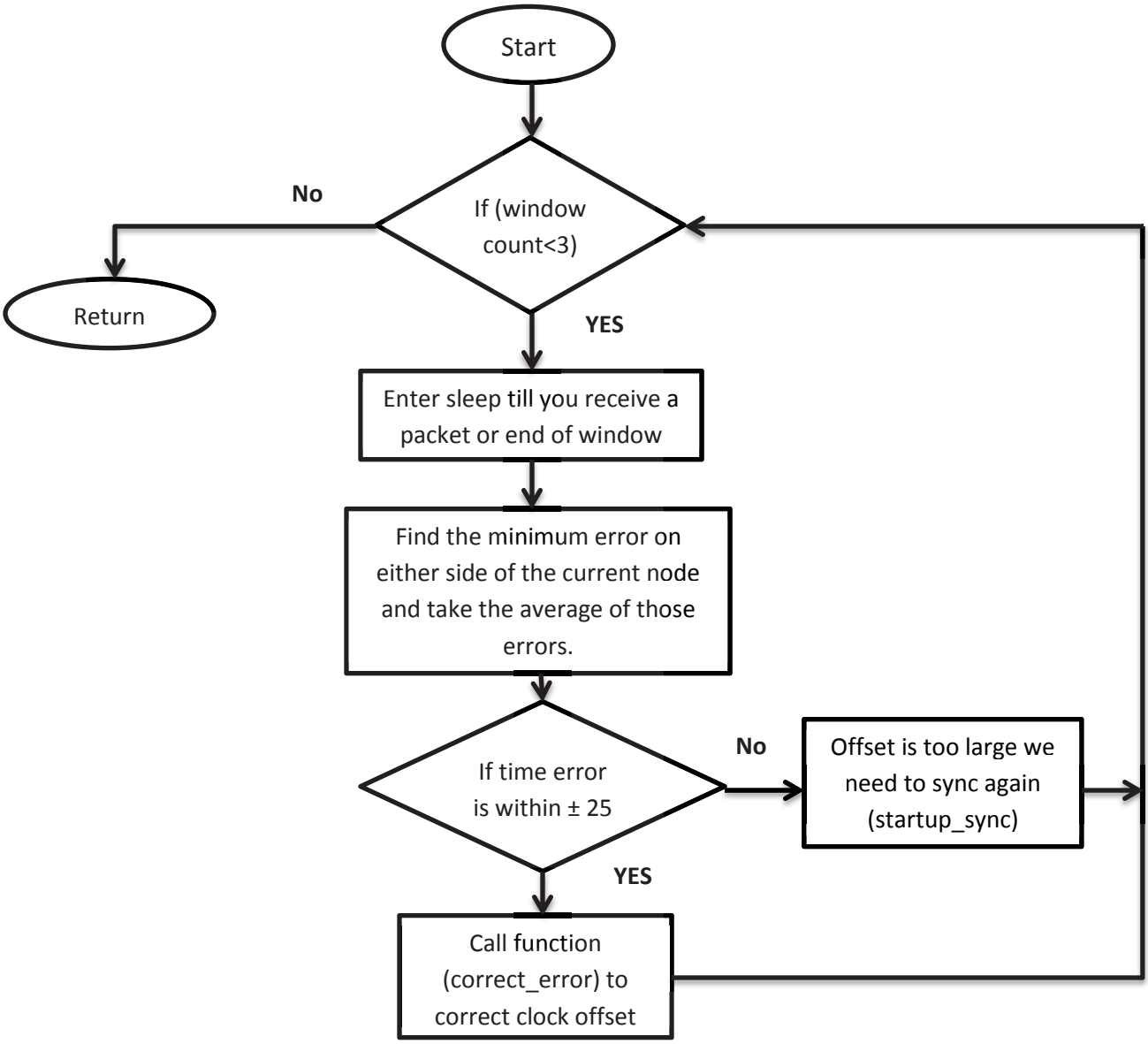


Flow chart 4: Function 'startup_sync'

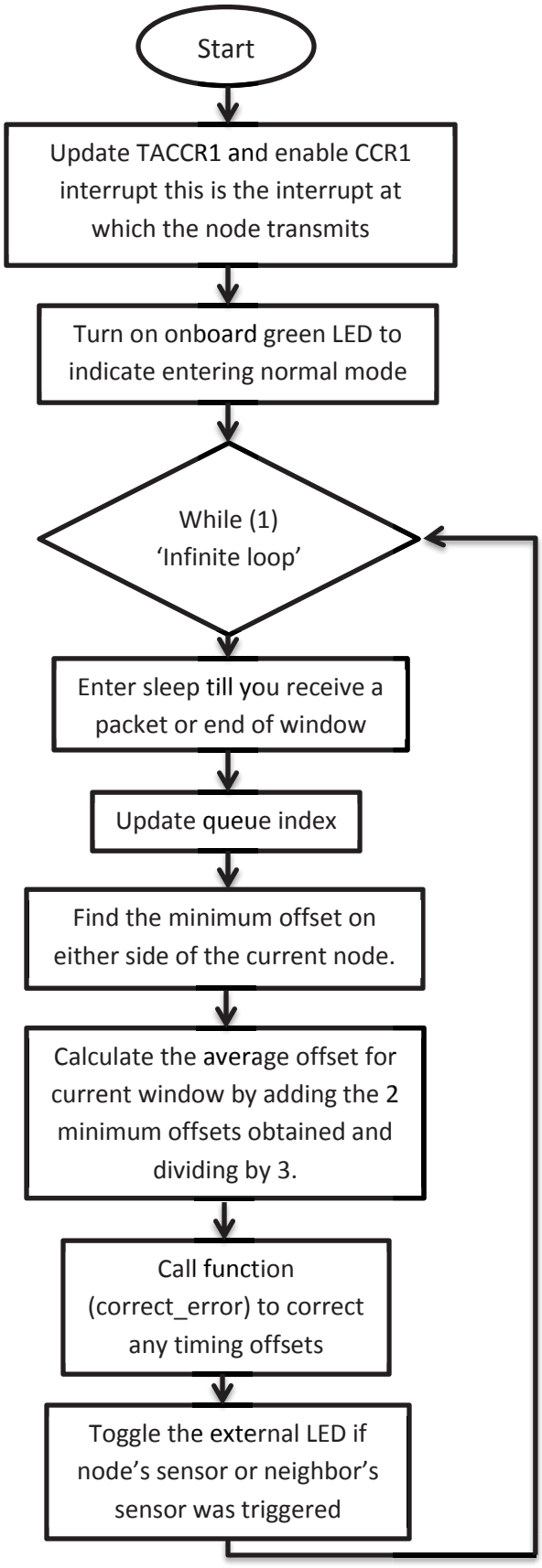



A
B
YES
Calculate time error for received packet
Correct clock skew and drift
If data in packet belongs to node address before me
YES
Reset window count
No

Flow chart 5: Function ‘sync_check’



Flow chart 6: function ‘normal_mode’



Appendix B

User Manual

Puneeth Ramesh

October, 31, 2014

Contents

1. Introduction

This document describes in detail the PC software used to monitor each node's activity. The tool helps monitor the sensor data and the clock offsets in each node. This provides real time information of each node's activity. The tool is designed to monitor data from all 16 nodes. The tool can be used to tap into any node and collect data. This is possible as the nodes relay data in both directions. This provides the flexibility to collect data from the first node, the last node or any node in-between. The tool was designed and programed using C#. The tool is designed to be used in tandem with a eZ430-Rf2500 receiver node. The receiver node is programed to collect data from a target node and send the collected data to the tool via a UART to a USB port. The tool also sends instructions back to the receiver node. The instruction could be the desired target node or the upstream or downstream data from the desired target node. This document is a guide on how to use the tool to monitor and collect error data from the nodes.

2. Overview of the tool

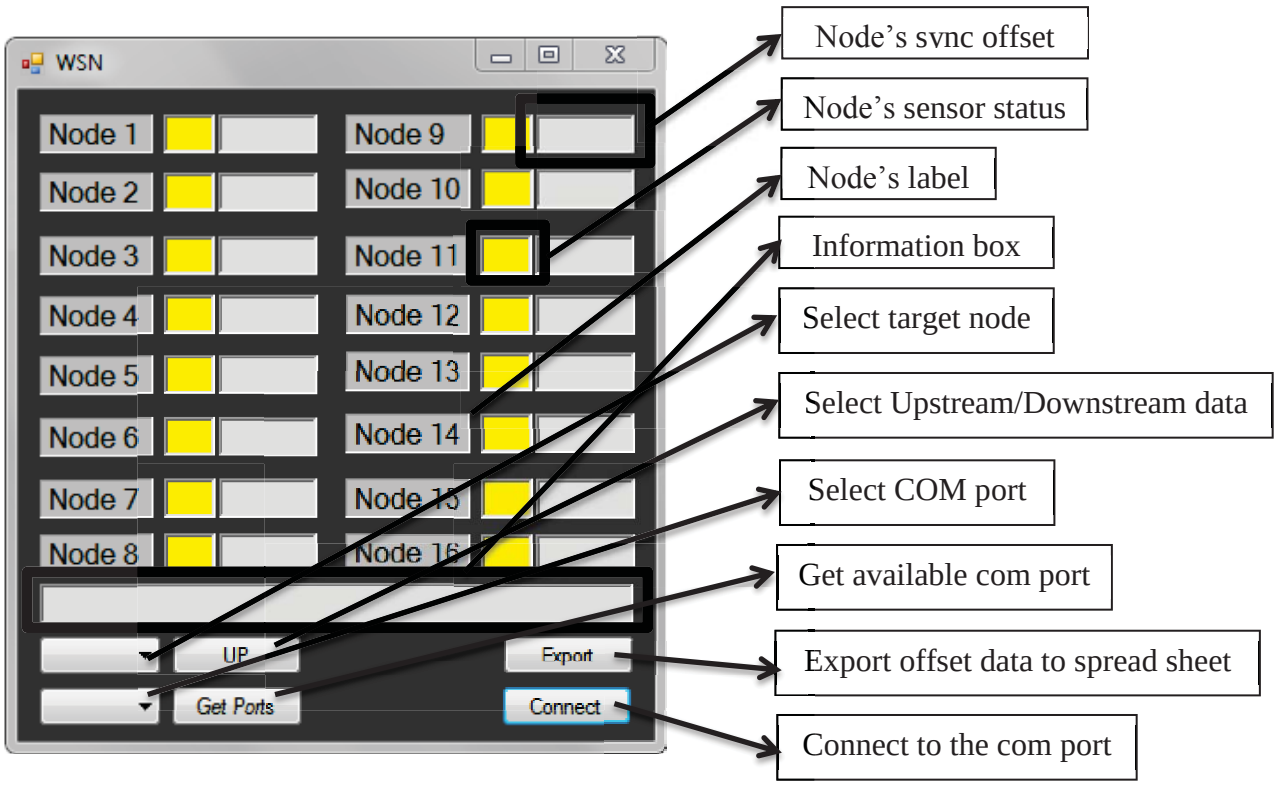


Figure 1. Basic layout of the tool's GUI.

3. Basic operation

This section will describe how to use the tool. The basic layout of the graphical user interface (GUI) is shown in figure 1. Before we start using the GUI we need to program the receiver node with the receiver code using IAR embedded workbench; only then we can communicate with the receiver node. Once the receiver node is up and running we can now start the tool by double clicking on the **"WSN.exe"** file. First the tool needs to connect to the receiver node. Since serial communication (UART) is being used, we need to connect to the appropriate COM port on the PC.

3.1 Programming the network nodes and the receiver node.

This manual assumes that the user has IAR and the required drivers installed on his computer. To program the nodes the user needs 2 files (**"node.c"** and **"wireless.h"**). Place both the files in the same project folder and select the device under project options as shown in figure 2. Before programming the device, update the node address in the main function as shown in figure 3. The node address has to be changed manually for each and every node. Then click on the **"Download and Debug"** button to program the device as shown in figure 4.

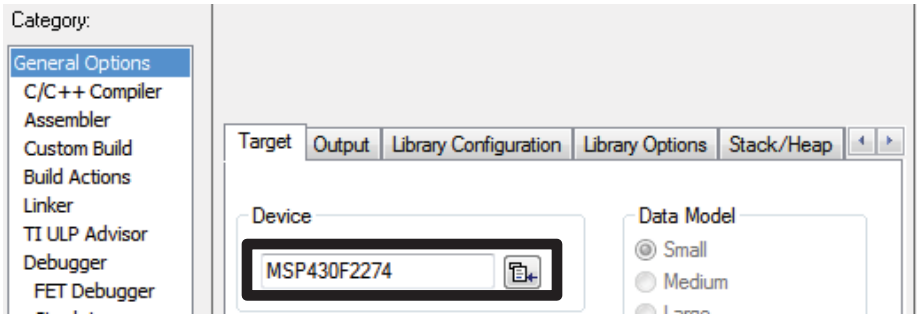


Figure 2. Select the device under project options.

```
void main()
{
  window_size = WIN_SIZE;              // Initialize window size ~64ms
  slot_size = SLOT_SIZE;               // Initialize slot size   ~4ms
  half_slot_size = HLF_SLOT_SIZE;      // Initialize half slot size
  node_add = 1;                        // Set node address
```

Figure 3. Update the Node address in the main function of the code.

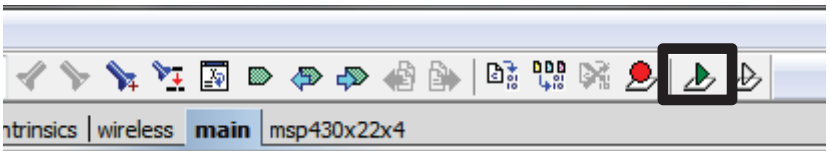


Figure 4. Click on the download and debug button to program the device.

The receiver node is programmed in the same way. The receiver node also requires two files (**"receiver_node.c"** and **"wireless.h"**) to program it. The wireless.h file is the same for both.

3.2 Connecting to the receiver node

First, plugin the receiver node to a USB port and then to find the receiver node's COM port. On your desktop click start. In the search bar type Device Manager. Then open Device Manager. This will give you the window shown in figure 5. Under ports we can find the receivers' COM port. In figure 5 the COM port is COM21.

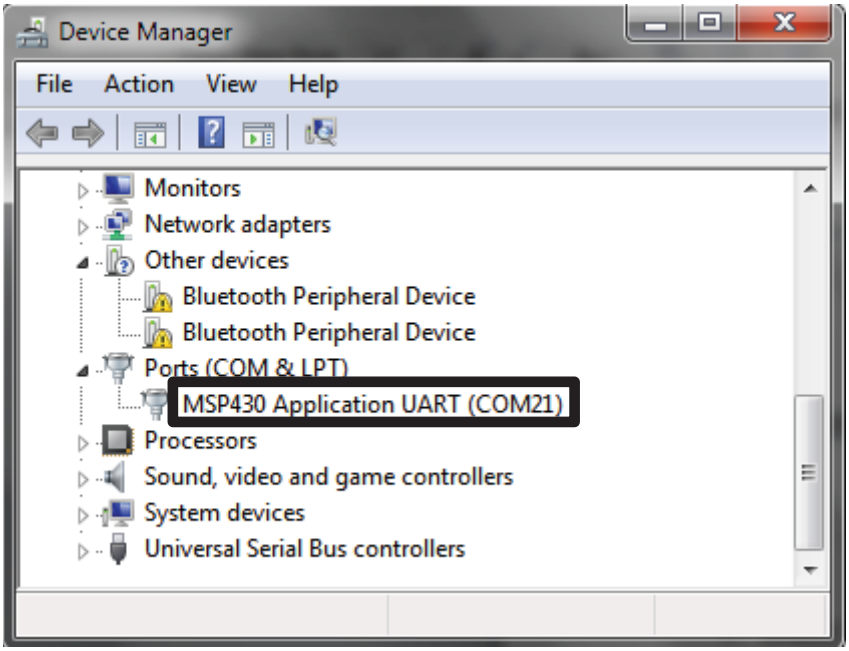


Figure 5. Device Manager showing active COM ports.

Once the COM port is known we can now connect to the receiver node. On the tool's GUI click the "Get Ports" button to generate a list of active COM ports on your PC and select the receivers' COM port as shown in figure 6.

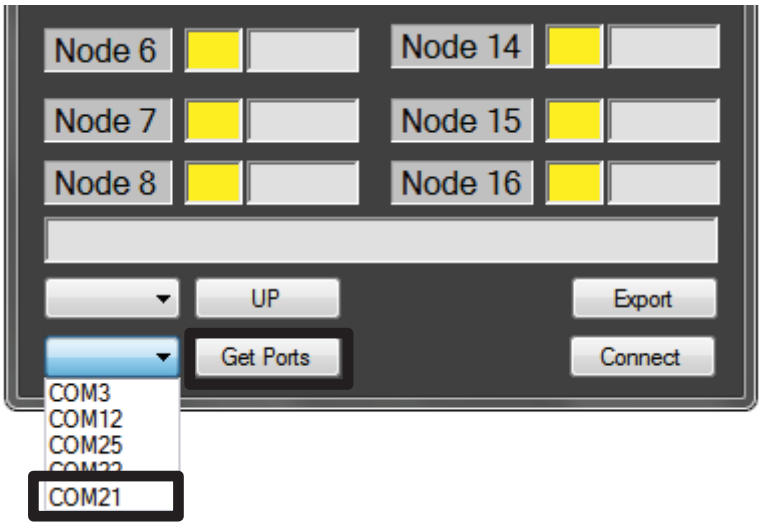


Figure 6. Selecting COM port on the GUI.

Once we select the COM port we can now hit the "Connect" button to establish a connection. Once the connection is established we get a confirmation in the information box as shown in figure 7. We cannot establish a connection without selecting the COM port. This will cause an error message on the information box.

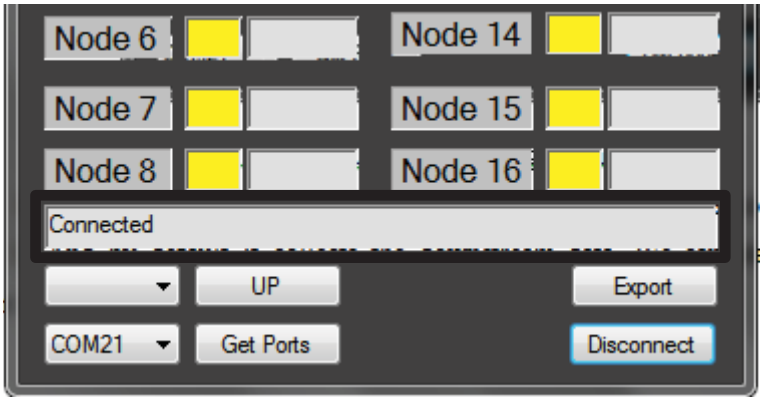


Figure 7. After establishing connection.

3.3 Changing the target node

Once the connection is established by default the tool starts collecting data from node 1. And by default it collects the upstream data and starts monitoring the sensor data. We can change the target node by selecting a different node from the drop down list next to the "UP" button as shown in figure 8. This can only be done once a

connection is established. Changing the target node before establishing a connection will result in a warning message in the info box.

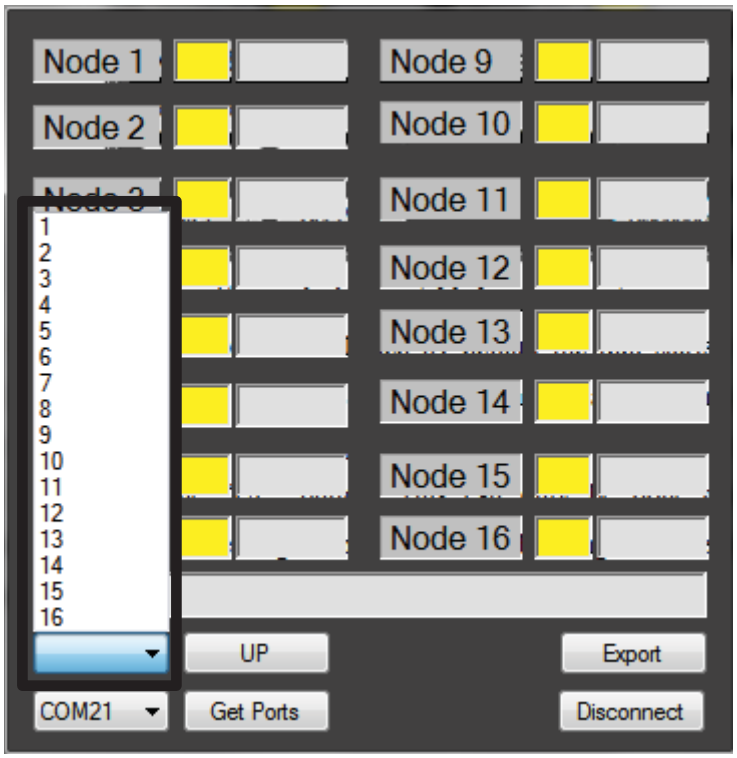


Figure 8. Changing the target node.

3.4 Changing streams

We can change streams by clicking the "UP/DOWN" button, as shown in figure 9. It is useful to change streams when our target node is the 16th node, as all the data flows downstream.

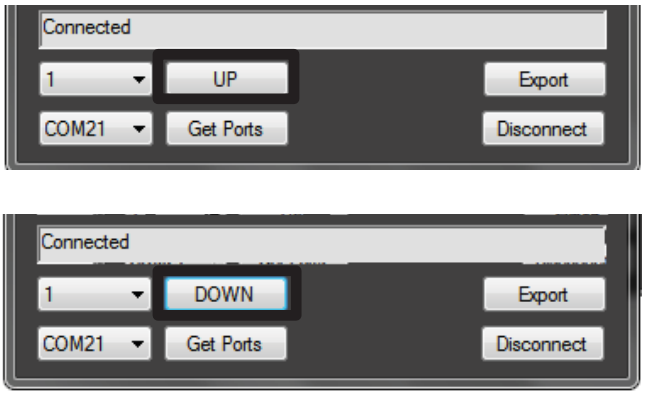


Figure 9. Collecting Upstream/Downstream data.

3.5 Export collected error data to spread sheet

The tool has the capacity to store the previous 300 sync offsets for all 16 nodes. This can be dumped into a spread sheet. Later on the data can be plotted. To save the data to a spread sheet we just need to hit the “Export” button as shown in figure 10. This will create or overwrite a csv file with the name “Output.csv”. It is recommended to disconnect before exporting the data.

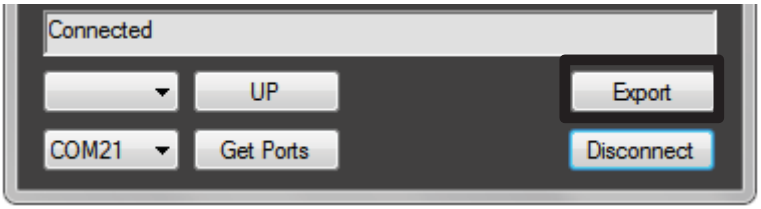


Figure 10. Export to spread sheet.

The data saved to the spread sheet is as shown in figure 11. This can be plotted to see the variation with respect to nominal.

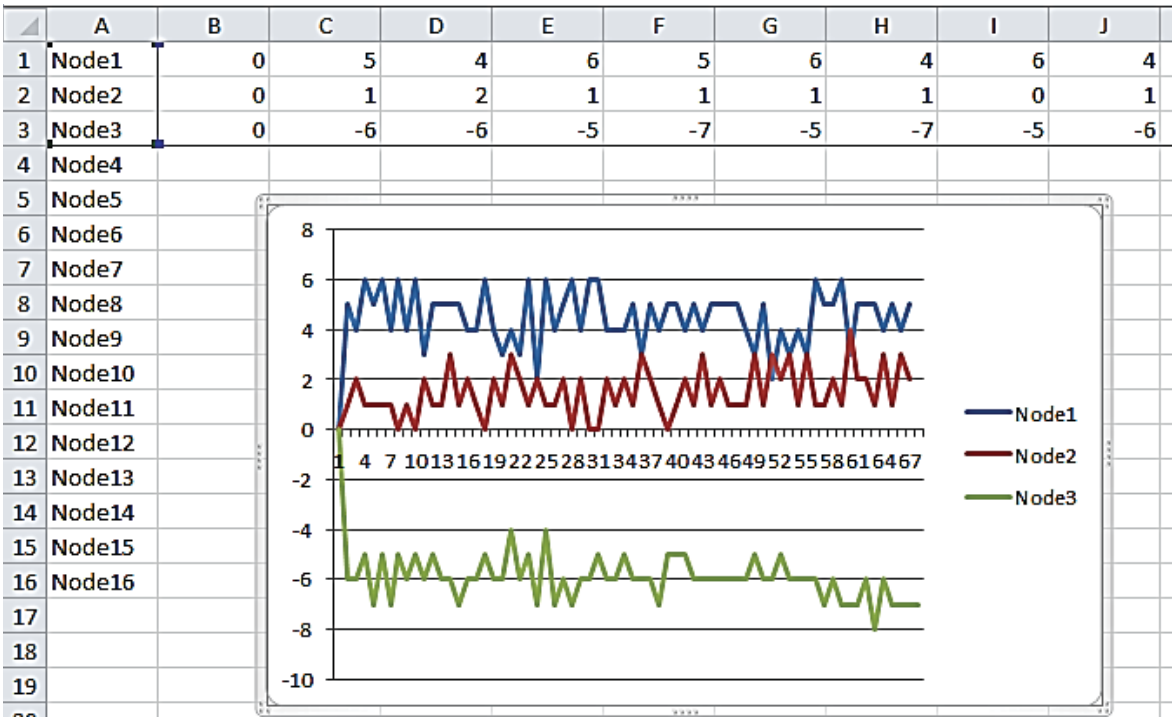


Figure 11. Exported data.

3.6 Monitoring sensor data

The sensor data from each node can be monitored from the small color window next to the node labels as shown in figure 12. The meaning of each color is shown in figure 13.

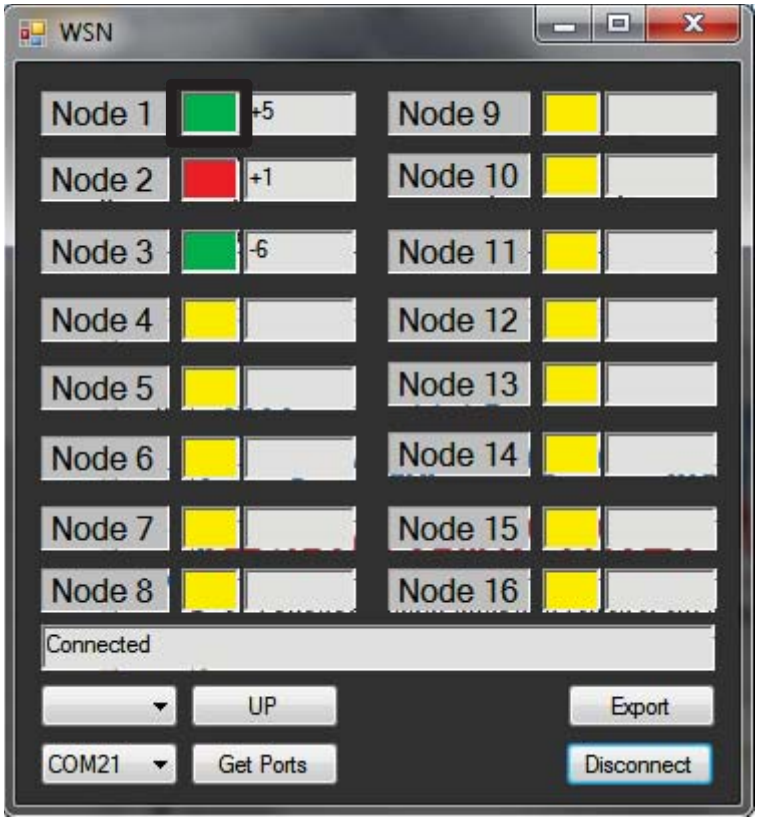


Figure 12. Monitoring sensor data.

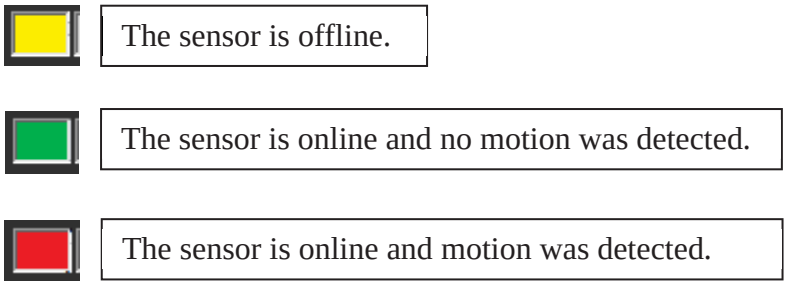


Figure 13. Sensor status.

3.7 Disconnecting the receiver node

To disconnect the receiver node from the tool hit the “Disconnect” button (as shown in figure 14) or closing the window with the “X” button at the top will also safely disconnect the receiver node (close’s the COM port). Caution: Force closing the tool from task manager or any other tool will lock the COM port.

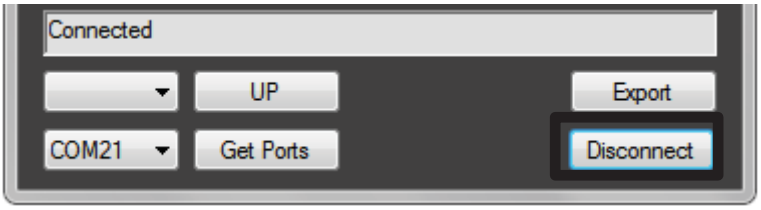


Figure 14. Disconnecting the receiver node.

4. References

[1] Microsoft Visual Studio 2013, *C# Programming Guide*. Accessed on: June 2014, Available: http://msdn.microsoft.com/en-us/library/67ef8sbd.aspx.

[2] Microsoft Developer Network, *SerialPort Class*. Accessed on: June 2014, Available: http://msdn.microsoft.com/en-us/library/system.io.ports.serialport% 28v =vs.110%29.aspx.

[3] Instructables Tutorial, *Serial Port Programming With .NET*. Accessed on: June 2014, Available: http://www.instructables.com/id/Serial-Port-Programming-With-NET/step1/Set-up-and-Open-the-Serial-Port/

[4] Texas instruments, *MSP430x22x2, MSP430x22x4 Mixed Signal microcontroller Datasheet*, 2012. , Available: http://www.ti.com.cn/cn/lit/ds/symlink/msp430f2274.pdf.

[5] Texas instruments, *CC2500 Low-Cost Low-Power 2.4 GHz RF Transceiver Datasheet*, 2014. , Available: http://www.ti.com/lit/ds/symlink/cc2500.pdf.

[6] Texas instruments, eZ430-RF2500 Development Tool User guide. SLAU227A Mixed Signal Products, 2007. , Available: http://www.ti.com/lit/ug/slau227e/slau227e.pdf.

Appendix C

Network Node Source Code:

```
#include "msp430x22x4.h"                // chip-specific macros & defs
#include "stdint.h"                     // MSP430 data type definitions
#include "wireless.h"                   // Wireless setup and func definition
#define WIN_SIZE 16000                  // Window size 64ms
#define SLOT_SIZE 1000                  // Slot size 4ms
#define HLF_SLOT_SIZE 500               // Hlaf slot size

uint8_t node_add = 0;                             // Node Address
uint8_t curr_que_idx,next_que_idx,futr_que_idx;// Queue Index
uint8_t packet_received_flg = 0, sync_flg = 0; // Packet rx flag
uint8_t neigh1_flg = 0, neigh2_flg = 0, neigh3_flg = 0, neigh4_flg = 0;
uint16_t window_count = 0,total_window_cnt;    // Window counts
uint16_t packet_rx[6],tx_delay = 187;   // Rx packet and transmition delay
uint8_t packet_tx[7],packet_queue[3][4],rx[6];
int32_t time_error, error_buff[4];
uint16_t window_size,slot_size,half_slot_size;
uint8_t loop_i,loop_j;                   // Loop index variable

// ============================================================
// Function transmits data using helper function "RFSendPacket"
// Args:  none
// Retn:  none
// Flow chart 1
// ============================================================
void f_TxData(void)
{
  P1OUT |= 0x01;                          // Turn on Red LED
  TI_CC_GDO0_PxIE &= ~TI_CC_GDO0_PIN;    // Disable int on end of packet
  uint8_t packet_size = 7,error_data = 0;// Packet length

  // Copy the current error and saturate it if greater than 1 byte
  if((time_error<=127)&&(time_error>=-127))
  {
      // Error is within 1 byte so we can copy it.
    error_data = (time_error>=0)?time_error:(0x80|(time_error*-1));
  }
    else if(time_error>127)
  {
    error_data = 127;                 // Upper limit (+127)
  }
    else
  {
    error_data = 0xFF;                // Lower limit (-127)
  }

  packet_tx[0] = 6;                     // packet lng excluding lng field
  packet_tx[1] = 0xFF;                  // Broadcast Address
  packet_tx[2] = node_add;              // Node address

  if(window_count == 0)
  {
    packet_tx[3] = ((P2IN<<7)|node_add);// Sensor data + Node address
```

```
    packet_tx[4] = error_data;          // Current error
    packet_tx[5] = ((P2IN<<7)|node_add);// Sensor data + Node address
    packet_tx[6] = error_data;          // Current error
  }
    else
  {
    packet_tx[3] = packet_queue[curr_que_idx][0];       // Transmit data from
    packet_tx[4] = packet_queue[curr_que_idx][1];       // queue
    packet_tx[5] = packet_queue[curr_que_idx][2];
    packet_tx[6] = packet_queue[curr_que_idx][3];
  }

  RFSendPacket(packet_tx, packet_size); // Send data

  packet_queue[curr_que_idx][0] = 0xFF; // Clear queue
  packet_queue[curr_que_idx][1] = 0xFF; // Clear queue
  packet_queue[curr_que_idx][2] = 0xFF; // Clear queue
  packet_queue[curr_que_idx][3] = 0xFF; // Clear queue

  TI_CC_GDO0_PxIE |= TI_CC_GDO0_PIN;    // Enable int on end of packet
  P1OUT &= ~0x01;                       // Turn off Red LED
  return;                               // Go back to sleep

}

//-------------------------------------------------------------------
// Function receives data from CC2500 chip when Rx interrupt is
// generated. The Rx interrupt is generated once CC2500 receives the
// last packet. Function uses the helper function 'RFReceivePacket'
// Args:  None
// Retn:  None
// Flow chart 2
//-------------------------------------------------------------------
#pragma vector=PORT2_VECTOR
__interrupt void f_RxData_ISR(void)
{
   uint8_t len = 6;                     //Receive 6 bytes
   uint8_t status[2];                   // Buffer to store status data

   packet_rx[1] = TAR;                  // Store time of received packet
   packet_received_flg = 0;             // Clear flag

   if(TI_CC_GDO0_PxIFG & TI_CC_GDO2_PIN)
      packet_received_flg = RFReceivePacket(rx,&len,status); //Fetch packet

   packet_rx[0] = rx[1];             // Grab transmitters node addr from pkt
   packet_rx[2] = (0x1F & rx[2]);    // Grab payloads node address from pkt

   if(sync_flg == 1)                            // If in sync mode
      __bic_SR_register_on_exit(LPM1_bits);     // Wake up on exit
   else if(packet_received_flg)
   {

     if(rx[1] == (node_add+1))                  // check addr for neighbors
     {
       packet_queue[next_que_idx][2] = rx[4];   // Store data in queue
       packet_queue[next_que_idx][3] = rx[5];   // Store data in queue
```

```
  error_buff[2]= packet_rx[1];       // Copy timestamp
  // Calculate error and store in buffer
  error_buff[2] -= (HLF_SLOT_SIZE+(SLOT_SIZE*(packet_rx[0]-1)));
  if((node_add+1)==(0x1F & rx[4]))
  {
    if((0x80 & rx[4])==0x80)               // Check if sensor is ON
      neigh1_flg = 1;                      // Set flag
    else
      neigh1_flg = 0;                      // Reset flag
  }
}
else if(rx[1] == (node_add+2))             // check addr for neighbors
{
  packet_queue[futr_que_idx][2] = rx[4];   // Store data in queue
  packet_queue[futr_que_idx][3] = rx[5];   // Store data in queue
  error_buff[3]= packet_rx[1];             // Copy timestamp

  // Calculate error and store in buffer
  error_buff[3] -= (HLF_SLOT_SIZE+(SLOT_SIZE*(packet_rx[0]-1)));
  if((node_add+2)==(0x1F & rx[4]))
  {
    if((0x80 & rx[4])==0x80)               // Check if sensor is ON
      neigh3_flg = 1;                      // Set flag
    else
      neigh3_flg = 0;                      // Reset flag
  }
}
else if(rx[1] == (node_add-1))             // check addr for neighbors
{
  packet_queue[next_que_idx][0] = rx[2];   // Store data in queue
  packet_queue[next_que_idx][1] = rx[3];   // Store data in queue
  error_buff[1]= packet_rx[1];             // Copy timestamp

  // Calculate error and store in buffer
  error_buff[1] -= (HLF_SLOT_SIZE+(SLOT_SIZE*(packet_rx[0]-1)));
  if((node_add-1)==(0x1F & rx[2]))
  {
    if((0x80 & rx[2])==0x80)               // Check if sensor is ON
      neigh2_flg = 1;                      // Set flag
    else
      neigh2_flg = 0;                      // Reset flag
  }
  if(packet_rx[2] == 1)
    window_count = node_add - 2;
}
else if(rx[1] == (node_add-2))             // check addr for neighbors
{
  packet_queue[futr_que_idx][0] = rx[2];   // Store data in queue
  packet_queue[futr_que_idx][1] = rx[3];   // Store data in queue
  error_buff[0]= packet_rx[1];             // Copy timestamp

  // Calculate error and store in buffer
  error_buff[0] -= (HLF_SLOT_SIZE+(SLOT_SIZE*(packet_rx[0]-1)));
  if((node_add-2)==(0x1F & rx[2]))
  {
    if((0x80 & rx[2])==0x80)               // Check if sensor is ON
      neigh4_flg = 1;                      // Set flag
```

```
          else
            neigh4_flg = 0;                       // Reset flag
        }
      }
    }
    TI_CC_GDO0_PxIFG &= ~TI_CC_GDO0_PIN; // After pkt RX, reset intrp flag.

    return;                                       // Go back to sleep
}

//-------------------------------------------------------------------
// Func:  Pull CPU out of sleep mode during transmition and at the
// end of the window.
// Args:  None
// Retn:  None
//-------------------------------------------------------------------
#pragma vector=TIMERA1_VECTOR
__interrupt void f_TimerAISR(void)
{
    switch (__even_in_range(TAIV, 10))
    {
       case TAIV_TAIFG:                   // Handle TAR rollover -> 0 IRQ
          window_count++;                 // Increament window count
          if(window_count == 16)
            window_count = 0;             // Clear window count
          total_window_cnt++;             // Increament window count
          TACTL &= ~TAIFG;                // Clear flag
          __bic_SR_register_on_exit(LPM1_bits);  // wake up on exit
          break;
       case TAIV_TACCR1:                  // Chnl 1 IRQ
         f_TxData();                      // Transmit data
         break;
       case TAIV_TACCR2:                  // ignore chnl 2 IRQ
       default:                           // ignore everything else
    }
    return;
}

//====================================================================
// Setup the Ports, Clocks and Wireless config.
// Args:  none
// Retn:  none
//====================================================================
void f_setup(void)
{
    WDTCTL = WDTPW + WDTHOLD;             // halt watchdog

    volatile uint16_t delay;              // Variable for delay

    // This is just a software delay
    for(delay=0;delay<650;delay++);

    // Setup clock system
    BCSCTL1 = CALBC1_8MHZ;                // set DCO freq.
    BCSCTL2 |= DIVS_3;                    // SMCLK = MCLK/8 (1MHz)
    DCOCTL  = CALDCO_8MHZ;                // set MCLK to 8MHz
```

```
  // TimerA config
  TACCR0 = window_size;                 // ~64msec
  TACTL = TASSEL_2 |ID_2 | MC_1 | TAIE;// SMCLK, Interrupts,Up mode,div by 4

  //Port config
  P1DIR |= 0x03;                        // activate LEDs
  P2DIR |= 0x02;                        // activate LEDs
  P1OUT &= ~0x03;                       // Clear LEDs

  // Wireless Initialization
  TI_CC_SPISetup();                     // Initialize SPI port
  P2SEL = 0;                            // Sets P2.6 & P2.7 as GPIO
  TI_CC_PowerupResetCCxxxx();           // Reset CCxxxx
  writeRFSettings();                    // Write RF settings to config reg
  TI_CC_GDO0_PxIES |= TI_CC_GDO0_PIN;  // Int on falling edge (end of pkt)
  TI_CC_GDO0_PxIFG &= ~TI_CC_GDO0_PIN; // Clear flag
  TI_CC_GDO0_PxIE |= TI_CC_GDO0_PIN;   // Enable int on end of packet
  TI_CC_SPIStrobe(TI_CCxxx0_SRX);       // Initialize CCxxxx in RX mode.

  // This is just a software delay
  for(delay=0;delay<650;delay++);

  P1OUT = 0x02;            // Turn on green led to indicate setup is done.
}

// ============================================================
// Function returns the minimum error
// Args:  Input errors and the error buffer index
// Retn:  minimum error
// ============================================================
int32_t min_error(int32_t error_1, int32_t error_2, uint8_t index)
{
  if(error_1<0)                 // If first error is negative
    error_1*=-1;                // Get ABS value

  if(error_2<0)                 // If second error is negative
    error_2*=-1;                // Get ABS value

  if(error_1>error_2)           // If first error is greater than second
  {
    return error_buff[index+1];     // return second error
  }
  else
  {
    if(error_1 != 0xFFF)
      return error_buff[index];     //return first error
    else
      return 0;
  }
}

// ============================================================
// Function used to correct Clock drift and Clock skew
// Args:  none
// Retn:  none
// Flow chart 3
// ============================================================
```

```
void correct_error(void)
{

  if(window_count == 0)
  {
    time_error = time_error>>1;
    window_size = 16000 + time_error;       // Add error to the window size
    slot_size = window_size >> 4;           // slot size = window size/16
    half_slot_size = slot_size >> 1;        // Half slot size= slot size/2

    TACCR0 = window_size;                   // Update TACCR0 with new values
    TACCR1 = half_slot_size + (slot_size*(node_add-1))- tx_delay;
  }
  else
  {
    if( time_error > 0 )                    // If error is positive
      while(TAR<time_error);                // wait for TAR to reach error
    TAR -= time_error;                      // Update TAR value
  }

  for(loop_i=0;loop_i<4;loop_i++)           // Reset error buffer
    error_buff[loop_i] = 0xFFF;

  return;
}

// ============================================================
// Function is used at startup to sync with neighbouring node
// Args:  none
// Retn:  none
// Flow chart 4
// ============================================================
void startup_sync(void)
{
  P1OUT |= 0x03;                              // Turn on both LEDs
  sync_flg = 1;                               // Enable sync flag
  total_window_cnt = 0;                       // Clear window count

  while(total_window_cnt<16)                // Sync for 16 windows
  {
    __bis_SR_register(LPM1_bits + GIE);       // Sleep till interrupt
    if(packet_received_flg)                   // Received packet is valid
    {
      if(packet_rx[0] == (node_add-1))        // Sync to the node before me
      {
        time_error = packet_rx[1];            // Grab received time

        // Calculate error with received time and expected time
        time_error -= (half_slot_size+(slot_size*(packet_rx[0]-1)));

        if( time_error > 0 )                  // If error is positive
          while(TAR<time_error);              // wait for TAR to reach error
        TAR -= time_error;                    // Update TAR value

        if(packet_rx[2] == (node_add-1))
          window_count = 0;                 // Match window count with neighbour
      }
```

```
      packet_received_flg = 0;                  // Clear flag
    }
  }

  sync_flg = 0;                                 // Clear sync flag
  P1OUT &= ~0x03;                               // Turn off LEDs
}


// ============================================================
// Function is used to check if synced properly with
// neighbouring nodes at startup.
// Args:  none
// Retn:  none
// Flow chart 5
// ============================================================
void sync_check()
{
  total_window_cnt = 0;                 // Clear window count

  while(total_window_cnt<3)             // Check for 3 windows
  {
    __bis_SR_register(LPM1_bits + GIE); // Sleep till interrupt

    TI_CC_SPIStrobe(TI_CCxxx0_SIDLE);   // Initialize CCxxxx in Idle mode.
    TI_CC_SPIStrobe(TI_CCxxx0_SRX);     // Initialize CCxxxx in RX mode.

    time_error = 0;                     // Clear time error
    error_buff[0]= min_error(error_buff[0],error_buff[1],0);//Get min error
    error_buff[1]= min_error(error_buff[2],error_buff[3],2);//Get min error
    time_error = (error_buff[0] + error_buff[1])/3;// Take the average error

    if((time_error<25) && (time_error>-25))     // If error is within +-25
    {
      correct_error();                  // Correct clock errors
    } else {
      startup_sync();                   // Restart startup sync
    }
  }
}

// ============================================================
// Function executed during normal mode. Main purpose of the
// function is to correct errors at the end of each window.
// Args:  none
// Retn:  none
// Flow chart 6
// ============================================================
void normal_mode(void)
{
  static uint8_t toggle_led = 0;

  // Update TACCR1 with new sizes
  TACCR1 = half_slot_size + (slot_size*(node_add-1))- tx_delay;
  TACCTL1 = CCIE;               // enable CCR1 interrupt (Tx interrupt)
  P1OUT |= 0x02;                        // Turn on Green LED
```

```
  while(1)
  {
    __bis_SR_register(LPM1_bits + GIE); // Enter sleep till interrupt

    TI_CC_SPIStrobe(TI_CCxxx0_SIDLE);   // Initialize CCxxxx in Idle mode.
    TI_CC_SPIStrobe(TI_CCxxx0_SRX);     // Initialize CCxxxx in RX mode.

    // Increment the queue indices
    curr_que_idx = (curr_que_idx>=2)?((curr_que_idx+1)-3) : (curr_que_idx+1);
    next_que_idx = (next_que_idx>=2)?((next_que_idx+1)-3) : (next_que_idx+1);
    futr_que_idx = (futr_que_idx>=2)?((futr_que_idx+1)-3) : (futr_que_idx+1);

    time_error = 0;                     // Clear time error
    error_buff[0]= min_error(error_buff[0],error_buff[1],0);//Get min error
    error_buff[1]= min_error(error_buff[2],error_buff[3],2);//Get min error
    time_error = (error_buff[0] + error_buff[1])/3;// Take the average error
    correct_error();                    // Correct clock errors

    // Check if node's sensor or neighbours sensor is triggered
    if((P2IN&0x01)|(neigh1_flg == 1)|(neigh2_flg == 1)|\
      (neigh3_flg == 1)|(neigh4_flg == 1))
    {
      if(P2IN&0x01)
        toggle_led = 1;         // Toggle every 64ms
      else
        toggle_led ^= 0x01;     // Toggle every 128ms

      if(toggle_led)
        P2OUT ^= 0x02;          // Toggle the LED
      else
        P2OUT &= ~0x02;         // Turn off the LED
    }
  }
}

// ============================================================
// This is the main function. Calls the setup function to
// configure the ports and other settings. Once setup is done
// startup sync is performed.After syncing we enter normal
// operating mode.
// Args:  none
// Retn:  none
// ============================================================
void main()
{
  window_size = WIN_SIZE;             // Initialize window size ~64ms
  slot_size = SLOT_SIZE;              // Initialize slot size   ~4ms
  half_slot_size = HLF_SLOT_SIZE;     // Initialize half slot size
  node_add = 1;                       // Set node address

  curr_que_idx = 0;                     // Initial value for queue index
  next_que_idx = 1;                     // Initial value for queue index
  futr_que_idx = 2;                     // Initial value for queue index

  // Clear the queue (0xFF means its empty). This is where the data is stored
  for(loop_i=0; loop_i<3;loop_i++)
    for(loop_j=0;loop_j<4;loop_j++)
```

```
      packet_queue[loop_i][loop_j] = 0xFF;

  for(loop_i=0;loop_i<4;loop_i++)       // Clear error buffer
      error_buff[loop_i] = 0xFFF;

  f_setup();                            // Setup Ports and wireless settings
  startup_sync();                       // Perform startup sync
  sync_check();                         // Check if sync was successful
  normal_mode();                        // Enter normal operating mode
}
```

Appendix D

Receiver Node Source Code:

```
#include "msp430x22x4.h"                // chip-specific macros & defs
#include "stdint.h"                     // MSP430 data type definitions
#include "wireless.h"                   // Wireless setup and func definition

uint8_t packet_received_flg = 0;        // Received pck flag
uint8_t rx[6];                          // Received packet
uint8_t RX_node = 1,stream_flag = 0;    // Target node address and stream

#pragma vector=USCIAB0RX_VECTOR
__interrupt void IsrUartEcho(void)
//-------------------------------------------------------------------------
// Func:  ISR receives a command byte from the user and updates the
//        stream direction (Upstream/Downstream)and the target node
//        address
// Args:  None
// Retn:  None
//-------------------------------------------------------------------------
{
    RX_node = UCA0RXBUF;                // Receive character
    stream_flag = ((RX_node & 0x80)>>7);// Upstream/Downstream
    RX_node &= 0x7F;                    // Update the target node add
    return;
}

//-------------------------------------------------------------------------
// Function receives data from CC2500 chip when Rx interrupt is
// generated. The Rx interrupt is generated once CC2500 receives the
// last packet. Function uses the helper function 'RFReceivePacket'
// Args:  None
// Retn:  None
//-------------------------------------------------------------------------
#pragma vector=PORT2_VECTOR
__interrupt void f_RxData_ISR(void)
{
   uint8_t len = 6;                     //Receive 6 bytes
   uint8_t status[2];                   // Buffer to store status data

   packet_received_flg = 0;         // Clear flag

   if(TI_CC_GDO0_PxIFG & TI_CC_GDO2_PIN)
      packet_received_flg = RFReceivePacket(rx,&len,status); //Fetch packet


   if(packet_received_flg)              // Check if valid packet is received
   {
      if(rx[1] == RX_node)              // check target address
      {
        P1OUT ^= 0x03;                  // Toggle LED
        if(stream_flag == 1)            // Downstream
        {
          while ( !(IFG2 & UCA0TXIFG)) {};// wait: while Tx Buff not empty
          UCA0TXBUF = (rx[2] | 0x60);     // send node addr and sensor data
```

```
            while ( !(IFG2 & UCA0TXIFG)) {};// wait: while Tx Buff not empty
            UCA0TXBUF = ((rx[3] & 0x0F)| 0x60);// send lower half of error
            while ( !(IFG2 & UCA0TXIFG)) {};// wait: while Tx Buff not empty
            UCA0TXBUF = (((rx[3] & 0xF0)>>4)| 0x60);// send upper half of error
            while ( !(IFG2 & UCA0TXIFG)) {};// wait: while Tx Buff not empty
            UCA0TXBUF = '\n';               // send new line char
          } else {
            while ( !(IFG2 & UCA0TXIFG)) {};// wait: while Tx Buff not empty
            UCA0TXBUF = (rx[4] | 0x60);     // send byte addr and sensor data
            while ( !(IFG2 & UCA0TXIFG)) {};// wait: while Tx Buff not empty
            UCA0TXBUF = ((rx[5] & 0x0F)| 0x60);// send lower half of error
            while ( !(IFG2 & UCA0TXIFG)) {};// wait: while Tx Buff not empty
            UCA0TXBUF = (((rx[5] & 0xF0)>>4)| 0x60); // send upper half of err
            while ( !(IFG2 & UCA0TXIFG)) {};// wait: while Tx Buff not empty
            UCA0TXBUF = '\n';               // send new line char
          }
        }
      }

      TI_CC_GDO0_PxIFG &= ~TI_CC_GDO0_PIN; // After pkt RX, reset intrp flag.
      return;
}

//-------------------------------------------------------------------
// Func:  Pull CPU out of sleep mode during transmition and at the
// end of the window.
// Args:  None
// Retn:  None
//-------------------------------------------------------------------
#pragma vector=TIMERA1_VECTOR
__interrupt void f_TimerAISR(void)
{
   switch (__even_in_range(TAIV, 10))
   {
      case TAIV_TAIFG:                  // Handle TAR rollover -> 0 IRQ
         TACTL &= ~TAIFG;               // Clear flag
         __bic_SR_register_on_exit(LPM1_bits);  // wake up on exit
         break;
      case TAIV_TACCR1:                 // Chnl 1 IRQ
        break;
      case TAIV_TACCR2:                 // ignore chnl 2 IRQ
      default:                          // ignore everything else
   }
   return;
}

//====================================================================
// Setup the Ports, Clocks and Wireless config.
// Args:  none
// Retn:  none
//====================================================================
void f_setup(void)
{
   WDTCTL = WDTPW + WDTHOLD;            // halt watchdog

   volatile uint16_t delay;             // Variable for delay
```

```
   // This is just a software delay
   for(delay=0;delay<650;delay++);

   // Setup clock system
   BCSCTL1 = CALBC1_8MHZ;                // set DCO freq.
   BCSCTL2 |= DIVS_3;                    // SMCLK = MCLK/8 (1MHz)
   DCOCTL  = CALDCO_8MHZ;                // set MCLK to 8MHz

   // Setup UART
   P3SEL = 0x30;                         // P3.4,5 = USCI_A0 TXD/RXD
   UCA0CTL1 |= UCSSEL_2;                 // UART uses SMCLK
   UCA0BR0 = 104;                        // 1MHz 9600
   UCA0BR1 = 0;                          // 1MHz 9600
   UCA0MCTL = UCBRS0;                    // Modulation UCBRSx = 1
   UCA0CTL1 &= ~UCSWRST;                 // Init. USCI state machine
   IE2 |= UCA0RXIE;                      // Enable USCI_A0 RX IRQ

   // TimerA config
   TACCR0 = 16000;                       // ~64msec
   TACTL = TASSEL_2 |ID_2 | MC_1 | TAIE;// SMCLK, Interrupts,Up mode,div by 4

   //Port config
   P1DIR |= 0x03;                        // activate LEDs
   P2DIR |= 0x02;                        // activate LEDs
   P1OUT &= ~0x03;                       // Clear LEDs

   // Wireless Initialization
   TI_CC_SPISetup();                     // Initialize SPI port
   P2SEL = 0;                            // Sets P2.6 & P2.7 as GPIO
   TI_CC_PowerupResetCCxxxx();           // Reset CCxxxx
   writeRFSettings();                    // Write RF settings to config reg
   TI_CC_GDO0_PxIES |= TI_CC_GDO0_PIN;  // Int on falling edge (end of pkt)
   TI_CC_GDO0_PxIFG &= ~TI_CC_GDO0_PIN; // Clear flag
   TI_CC_GDO0_PxIE |= TI_CC_GDO0_PIN;   // Enable int on end of packet
   TI_CC_SPIStrobe(TI_CCxxx0_SRX);       // Initialize CCxxxx in RX mode.

   // This is just a software delay
   for(delay=0;delay<650;delay++);

   P1OUT = 0x02;           // Turn on green led to indicate setup is done.
}


// ===========================================================
// Function executed during normal mode. Main purpose of the
// function is to remain in sleep till an interrupt is
// requested.
// Args:  none
// Retn:  none
// ===========================================================
void normal_mode(void)
{
  while(1)
  {
    __bis_SR_register(LPM1_bits + GIE); // Enter sleep till interrupt

    TI_CC_SPIStrobe(TI_CCxxx0_SIDLE);   // Initialize CCxxxx in Idle mode.
```

```
    TI_CC_SPIStrobe(TI_CCxxx0_SRX);     // Initialize CCxxxx in RX mode.
  }
}

// ============================================================
// This is the main function. Calls the setup function to
// configure the ports and other settings. Once setup is done
// it enters normal operating mode.
// Args:  none
// Retn:  none
// ============================================================
void main()
{
  f_setup();                            // Setup Ports and wireless settings
  normal_mode();                        // Enter normal operating mode
}
```

Appendix E

GUI Source Code:

```
// Include preprocessor directives
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.IO;
using System.IO.Ports;
using System.Threading;

namespace WSN
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();

            // Update the node address combo list
            for(int i=1;i<17;i++)
                comboBox2.Items.Add(i);
        }

        private int connect_flag = 0;       // Connect flag
        private int stream_flag = 0;        // Stream flag
        private string Rxdata;              // Received packet
        private int tx_data = 1;            // Data to transmit
        private int node_add;               // Node address
        private int sensor_data;            // Sensor status
        private int error_sign;             // Sign +/- of error
        private int error_data;             // Sync offset

        // These counter values are used as a trigger
        // to update the GUI
        private byte data_counter1 = 0;
        private byte data_counter2 = 0;
        private byte data_counter3 = 0;
        private byte data_counter4 = 0;
        private byte data_counter5 = 0;
        private byte data_counter6 = 0;
        private byte data_counter7 = 0;
        private byte data_counter8 = 0;
        private byte data_counter9 = 0;
        private byte data_counter10 = 0;
        private byte data_counter11 = 0;
        private byte data_counter12 = 0;
        private byte data_counter13 = 0;
        private byte data_counter14 = 0;
```

```
  private byte data_counter15 = 0;
  private byte data_counter16 = 0;
  private int average_loop = 0;

  // Buffer to store the clock offsets
  private byte[,] node_error = new byte[16, 300];
  private byte[] error_temp = new byte[16]; // Temp buffer
  private int error_loop = 0, error_index = 0; //offset index

//=====================================================================
// Function: Update COM port list combo box
//=====================================================================

  private void button1_Click(object sender, EventArgs e)
  {
      string[] ports = SerialPort.GetPortNames(); //Get port list
      foreach (string port in ports)
      {
          comboBox1.Items.Add(port); // Populate the combo list
      }

  }


//=====================================================================
// Function: Change Upstream/Downstream settings
//=====================================================================
  private void button2_Click(object sender, EventArgs e)
  {
      if (stream_flag == 0)
      {
          button2.Text = "DOWN";       // Downstream
          stream_flag = 1;             // Set stream flag
          tx_data |= 0x80;             // Set the stream bit
      }
      else
      {
          button2.Text = "UP";         // Upstream
          stream_flag = 0;             // Set stream flag
          tx_data &= 0x7F;             // Clear the stream bit
      }

      byte[] buffer = new byte[] {Convert.ToByte(tx_data)};
      try
      {   // Send data via UART
          serialPort1.Write(buffer, 0, 1);
      }
      catch (InvalidOperationException err)
      {
          infobox.Text = err.Message;  //Send error message
      }
  }

//=====================================================================
// Function: Target address selection function
//=====================================================================
```

```
  private void comboBox2_SelectedIndexChanged\
  (object sender, EventArgs e)
  {
      // Selected target node address
      string sel_node = comboBox2.SelectedItem.ToString();
      tx_data =(int) Decimal.Parse(sel_node); // Convert to decimal

      // Add stream info
      tx_data |= (stream_flag << 7);
      byte[] buffer = new byte[] {Convert.ToByte(tx_data)};
      try
      {   // Send data via UART
          serialPort1.Write(buffer, 0, 1);
      }
      catch(InvalidOperationException err)
      {
          infobox.Text = err.Message;  //Send error message
      }
  }

//=====================================================================
// Function: Connect to COM port button function
//=====================================================================
  private void connect_button_Click\
  (object sender, EventArgs e)
  {
      for (int i = 0; i < 16; i++)
          error_temp[i] = 0xFF; // Reset the error buffer

      if (connect_flag == 0)  // If Not connected
      {
          try
          {
              serialPort1.PortName =
                 comboBox1.SelectedItem.ToString();

              // Check if port is already open
               if (!serialPort1.IsOpen)
               {
                  serialPort1.Encoding =
                  System.Text.Encoding.GetEncoding(28591);
                  infobox.Text = "Connected";
                  serialPort1.Open();// Open COM port
                  connect_button.Text = "Disconnect";
                  connect_flag = 1; // set the connect flag
              }
              else
              {
                  infobox.Text = \
                  "Unable to Connect to port";
              }
          }
          catch (UnauthorizedAccessException err)
          {
              infobox.Text = err.Message;
          }
```

```
            catch (NullReferenceException)
            {
                infobox.Text = "Please choose a port";
            }
            catch (Exception)
            {
              infobox.Text = "Unable to Connect to port";
            }
        }
        else
        {
            try
            {
                serialPort1.Close();   // Close COM port
                infobox.Text = "Disconnected";
                connect_button.Text = "Connect";
                connect_flag = 0;      // Clear connect flag
            }
            catch (Exception)
            {
                infobox.Text = "Unable to Close to port";
            }
        }
    }

//=====================================================================
// Function: This function is executed when the close button is hit
//=====================================================================
    private void Form1_FormClosing(object sender,
    FormClosingEventArgs e)
    {
        if (serialPort1.IsOpen)
        {
            e.Cancel = true; //cancel the form closing
            Thread CloseDown = new Thread(new
            ThreadStart(CloseSerialOnExit));

            //close port in new thread to avoid hang
            CloseDown.Start();
        }

    }

    private void CloseSerialOnExit()
    {
        try
        {
            serialPort1.Close(); //close the serial port
        }

        catch (Exception ex)
        {
           //catch any serial port closing error messages
            MessageBox.Show(ex.Message);
        }

        //now close back in the main thread
```

```
        this.Invoke(new EventHandler(NowClose));
    }

    private void NowClose(object sender, EventArgs e)
    {
        this.Close(); //now close the form
    }

//=====================================================================
// Function: Function to Receive data from the COM port
//=====================================================================
    private void serialPort1_DataReceived(object sender,
    SerialDataReceivedEventArgs e)
    {
        // Receive data from com port
        Rxdata = serialPort1.ReadLine();
        // Update GUI
        this.Invoke(new EventHandler(UpdateGUI));
    }

//=====================================================================
// Function: This is where all the GUI stuff happens
//=====================================================================
    private void UpdateGUI(object s, EventArgs e)
    {
        int rx_length = Rxdata.Length;// Received packet length
        int temp_data;
        if (rx_length == 3)
        {
            node_add = (Rxdata[0] & 0x1F);          // Get Node address
            sensor_data = ((Rxdata[0] & 0x80)>>7);  // Get sensor status

            // Get clock offset
            temp_data = ((Rxdata[1] & 0x0F) | (Rxdata[2]<<4));
            error_sign = ((temp_data & 0x80) >> 7); // Get sign of offset
            error_data = (temp_data & 0x7F);        // Get offset

            if ((node_add > 0) && (node_add < 17))
            {
                // Update the highest address RX
                if (node_add > average_loop)
                    average_loop = node_add;

                if ((error_index > node_add) || (error_index/
                 == node_add))
                {
                    for (int i = 0; i < 16; i++)
                    {
                        // Copy to offset buffer
                        node_error[i, error_loop] = error_temp[i];
                        error_temp[i] = 0xFF;       // Clear temp buffer
                    }
                    error_loop++;
                    if (error_loop == 300)          // Reset buffer index
                    error_loop = 0;
                }
```

```
            // Copy offset to temp buffer
            error_temp[node_add - 1] = (byte)temp_data;
            error_index = node_add;
        }
    }
    else
        return;

    //Update node 1's GUI info
    if(node_add == 1)
    {
        data_counter1 = 0;

        // Update node's sensor data
        if (sensor_data == 1)
            pictureBox1.Image = WSN.Properties.Resources.red;
        else
            pictureBox1.Image = WSN.Properties.Resources.green;

         // Update node's error data
        if (error_sign == 1)
            box_node1.Text = "-" + error_data;
        else
            box_node1.Text = "+" + error_data;

    } else if(data_counter1==48)
    {
        pictureBox1.Image = WSN.Properties.Resources.yellow;
        box_node1.Text = "";
    } else {
        data_counter1++;
    }

    //Update node 2's GUI info
    if(node_add == 2)
    {
         data_counter2 = 0;

        // Update node's sensor data
        if (sensor_data == 1)
            pictureBox2.Image = WSN.Properties.Resources.red;
        else
            pictureBox2.Image = WSN.Properties.Resources.green;

         // Update node's error data
        if (error_sign == 1)
            box_node2.Text = "-" + error_data;
        else
            box_node2.Text = "+" + error_data;
    }
      else if(data_counter2==48)
    {
        pictureBox2.Image = WSN.Properties.Resources.yellow;
        box_node2.Text = "";
    } else {
        data_counter2++;
    }
```

```
//Update node 3's GUI info
if(node_add == 3)
{
     data_counter3 = 0;

    // Update node's sensor data
    if (sensor_data == 1)
        pictureBox3.Image = WSN.Properties.Resources.red;
    else
        pictureBox3.Image = WSN.Properties.Resources.green;

     // Update node's error data
    if (error_sign == 1)
        box_node3.Text = "-" + error_data;
    else
        box_node3.Text = "+" + error_data;
}
    else if(data_counter3==48)
{
    pictureBox3.Image = WSN.Properties.Resources.yellow;
    box_node3.Text = "";
} else {
    data_counter3++;
}

//Update node 4's GUI info
if(node_add == 4)
{
    data_counter4 = 0;

    // Update node's sensor data
    if (sensor_data == 1)
        pictureBox4.Image = WSN.Properties.Resources.red;
    else
        pictureBox4.Image = WSN.Properties.Resources.green;

     // Update node's error data
    if (error_sign == 1)
        box_node4.Text = "-" + error_data;
    else
        box_node4.Text = "+" + error_data;
}
else if(data_counter4==48)
{
    pictureBox4.Image = WSN.Properties.Resources.yellow;
    box_node4.Text = "";
} else {
    data_counter4++;
}

//Update node 5's GUI info
if(node_add == 5)
{
    data_counter5 = 0;

    // Update node's sensor data
```

```
    if (sensor_data == 1)
        pictureBox5.Image = WSN.Properties.Resources.red;
    else
        pictureBox5.Image = WSN.Properties.Resources.green;

     // Update node's error data
    if (error_sign == 1)
        box_node5.Text = "-" + error_data;
    else
        box_node5.Text = "+" + error_data;
}
else if(data_counter5==48)
{
    pictureBox5.Image = WSN.Properties.Resources.yellow;
    box_node5.Text = "";
} else {
    data_counter5++;
}

//Update node 6's GUI info
if(node_add == 6)
{
    data_counter6 = 0;

    // Update node's sensor data
    if (sensor_data == 1)
        pictureBox6.Image = WSN.Properties.Resources.red;
    else
        pictureBox6.Image = WSN.Properties.Resources.green;

     // Update node's error data
    if (error_sign == 1)
        box_node6.Text = "-" + error_data;
    else
        box_node6.Text = "+" + error_data;

}
else if(data_counter6==48)
{
    pictureBox6.Image = WSN.Properties.Resources.yellow;
    box_node6.Text = "";
} else {
    data_counter6++;
}

//Update node 7's GUI info
if(node_add == 7)
{
     data_counter7 = 0;
     // Update node's sensor data
     if (sensor_data == 1)
         pictureBox7.Image = WSN.Properties.Resources.red;
     else
          pictureBox7.Image = WSN.Properties.Resources.green;

     // Update node's error data
      if (error_sign == 1)
```

```
            box_node7.Text = "-" + error_data;
        else
            box_node7.Text = "+" + error_data;
}
else if(data_counter7==48)
{
    pictureBox7.Image = WSN.Properties.Resources.yellow;
    box_node7.Text = "";
} else {
    data_counter7++;
}

//Update node 8's GUI info
if(node_add == 8)
{
     data_counter8 = 0;

     // Update node's sensor data
     if (sensor_data == 1)
         pictureBox8.Image = WSN.Properties.Resources.red;
     else
         pictureBox8.Image = WSN.Properties.Resources.green;

     // Update node's error data
     if (error_sign == 1)
         box_node8.Text = "-" + error_data;
     else
         box_node8.Text = "+" + error_data;

}
else if(data_counter8==48)
{
    pictureBox8.Image = WSN.Properties.Resources.yellow;
    box_node8.Text = "";
} else {
    data_counter8++;
}

//Update node 9's GUI info
if(node_add == 9)
{
     data_counter9 = 0;

     // Update node's sensor data
     if (sensor_data == 1)
         pictureBox9.Image = WSN.Properties.Resources.red;
     else
         pictureBox9.Image = WSN.Properties.Resources.green;

     // Update node's error data
     if (error_sign == 1)
         box_node9.Text = "-" + error_data;
     else
         box_node9.Text = "+" + error_data;
}
else if(data_counter9==48)
{
```

```
    pictureBox9.Image = WSN.Properties.Resources.yellow;
    box_node9.Text = "";
} else {
    data_counter9++;
}

//Update node 10's GUI info
if(node_add == 10)
{
     data_counter10 = 0;

     // Update node's sensor data
     if (sensor_data == 1)
         pictureBox10.Image = WSN.Properties.Resources.red;
     else
         pictureBox10.Image = WSN.Properties.Resources.green;

     // Update node's error data
     if (error_sign == 1)
         box_node10.Text = "-" + error_data;
     else
         box_node10.Text = "+" + error_data;
}
else if(data_counter10==48)
{
    pictureBox10.Image = WSN.Properties.Resources.yellow;
    box_node10.Text = "";
} else {
    data_counter10++;
}

if(node_add == 11)
{
     data_counter11 = 0;

     // Update node's sensor data
     if (sensor_data == 1)
         pictureBox11.Image = WSN.Properties.Resources.red;
     else
         pictureBox11.Image = WSN.Properties.Resources.green;

     // Update node's error data
     if (error_sign == 1)
         box_node11.Text = "-" + error_data;
     else
         box_node11.Text = "+" + error_data;
}
else if(data_counter11==48)
{
    pictureBox11.Image = WSN.Properties.Resources.yellow;
    box_node11.Text = "";
} else {
    data_counter11++;
}

//Update node 11's GUI info
if(node_add == 12)
```

```
{
     data_counter12 = 0;

     // Update node’s sensor data
     if (sensor_data == 1)
         pictureBox12.Image = WSN.Properties.Resources.red;
     else
         pictureBox12.Image = WSN.Properties.Resources.green;

     // Update node’s error data
     if (error_sign == 1)
         box_node12.Text = "-" + error_data;
     else
         box_node12.Text = "+" + error_data;
}
else if(data_counter12==48)
{
    pictureBox12.Image = WSN.Properties.Resources.yellow;
    box_node12.Text = "";
} else {
    data_counter12++;
}

//Update node 13's GUI info
if(node_add == 13)
{
     data_counter13 = 0;

     // Update node’s sensor data
     if (sensor_data == 1)
         pictureBox13.Image = WSN.Properties.Resources.red;
     else
         pictureBox13.Image = WSN.Properties.Resources.green;

     // Update node’s error data
     if (error_sign == 1)
         box_node13.Text = "-" + error_data;
     else
         box_node13.Text = "+" + error_data;
}
else if(data_counter13==48)
{
    pictureBox13.Image = WSN.Properties.Resources.yellow;
    box_node13.Text = "";
} else {
    data_counter13++;
}

//Update node 14's GUI info
if(node_add == 14)
{
     data_counter14 = 0;

     // Update node’s sensor data
     if (sensor_data == 1)
         pictureBox14.Image = WSN.Properties.Resources.red;
     else
```

```
            pictureBox14.Image = WSN.Properties.Resources.green;

        // Update node's error data
        if (error_sign == 1)
            box_node14.Text = "-" + error_data;
        else
            box_node14.Text = "+" + error_data;
}
else if(data_counter14==48)
{
    pictureBox14.Image = WSN.Properties.Resources.yellow;
    box_node14.Text = "";
} else {
    data_counter14++;
}

//Update node 15's GUI info
if (node_add == 15)
{
    data_counter15 = 0;

     // Update node's sensor data
    if (sensor_data == 1)
        pictureBox15.Image = WSN.Properties.Resources.red;
    else
        pictureBox15.Image = WSN.Properties.Resources.green;

     // Update node's error data
    if (error_sign == 1)
        box_node15.Text = "-" + error_data;
    else
        box_node15.Text = "+" + error_data;
}

else if (data_counter15 == 48)
{
    pictureBox15.Image = WSN.Properties.Resources.yellow;
    box_node15.Text = "";
}
else
{
    data_counter15++;
}

//Update node 16's GUI info
if(node_add == 16)
{
     data_counter16 = 0;

     // Update node's sensor data
     if (sensor_data == 1)
         pictureBox16.Image = WSN.Properties.Resources.red;
     else
         pictureBox16.Image = WSN.Properties.Resources.green;

     // Update node's error data
     if (error_sign == 1)
```

```
                box_node16.Text = "-" + error_data;
            else
                box_node16.Text = "+" + error_data;
        }
        else if(data_counter16==16)
        {
            pictureBox16.Image = WSN.Properties.Resources.yellow;
            box_node16.Text = "";
        } else {
            data_counter16++;
        }
    }

//=====================================================================
// Function: Function which does the exporting to spread sheet
//=====================================================================
    private void button3_Click(object sender, EventArgs e)
    {
        var csv = new StringBuilder();
        string filePath = "Output.csv";     // Output file name

        for (int i = 0; i < average_loop; i++)
        {
            var newLine = string.Format("Node{0}", i+1);    // Node label
            csv.Append(newLine);
            for (int j = 0; j < error_loop; j++)
            {
                if (node_error[i, j] == 0xFF)
                    newLine = string.Format(",");           // Empty cell
                else
                {

                    if ((node_error[i, j] & 0x80) == 0x80)
                        newLine = string.Format(",-{0}", \
                        (node_error[i, j] & 0x7F));
                    else
                        newLine = string.Format(",+{0}", \
                        (node_error[i, j]));
                }
                csv.Append(newLine);
            }
            newLine = string.Format("{0}", Environment.NewLine);
            csv.Append(newLine);
        }
        var result_line = string.Format("{0},Max,Min,Mean{0}",
        Environment.NewLine);// Labels Max, Min, Mean
        csv.Append(result_line);
        for (int i = 0; i < average_loop; i++)
        {
            // Get the min,max and the mean for the offsets collected
            result_line = string.Format("Node{0},=MAX({0}:{0}),
            =MIN({0}:{0}),=AVERAGE({0}:{0}){1}", i + 1,
            Environment.NewLine);
            csv.Append(result_line);
        }
        result_line = string.Format("{0}", Environment.NewLine);
        csv.Append(result_line);
```

```
            error_loop = 0;
            File.WriteAllText(filePath, csv.ToString()); // Write to the file
            infobox.Text = "File saved";
        }
    }
}
```