ENSC 427: Communications Network

Project: ZigBee Transmission Analysis in Tree Topology
Spring 2009

Final Report

Professor: Dr. Ljiljana Trajkovic

Date: March 5, 2009

Team 6:
Web page – http://www.sfu.ca/~kyc1
Austen Chan – 200114471 – kyc1@sfu.ca
Brian Cheung – 200101891 – bcheung1@sfu.ca
Wing Kit Lee – 200122075 – wkl2@sfu.ca

Abstract

ZigBee, a communication protocol that uses radio-frequency, is used in portable electronics, such as PDA, cellular phones, and game consoles. ZigBee is a low-cost, low-power, wireless mesh networking standard that is targeted for use in applications requiring low data rates and low power consumption. In this project, ZigBee tree topology performance, such as end-to-end delay and throughput will be analyzed. We will also examine the ZigBee network performance on both fixed and mobile nodes.

Table of Content

1. Introduction

In recent years, great expansions of remote control devices have surrounded our daily lives. One of which is the ZigBee wireless technology, a wireless protocol for short-range communications using radio frequency. ZigBee was created to satisfy the market's need for a cost-effective, standards-based wireless network that supports low data rates, low power consumption, security, and reliability. The ZigBee Alliance is the main industry working group developing ZigBee, and they work closely with Institute of Electrical and Electronics Engineering (IEEE) to ensure an integrated, complete, and interoperable network for the market. ZigBee is intended to be simpler and less expensive than other WPANs, such as Bluetooth.

ZigBee is a low-cost, low-power, wireless network communication protocol (IEEE 802.15.4). ZigBee is very similar to Bluetooth™, however, its speed is slower than Bluetooth due to its low power consumption. Some of the applications of ZigBee include home entertainment and control systems, mobile services, and home awareness security systems. There are several topologies used for Zigbee: star, mesh, and tree. In this project, focus will only be on ZigBee's performance in tree topologies.

In a tree network, the coordinator, which defines the frequency channel of the network and allows other devices to join the network, are located at the very top of the tree. The coordinator may not be passed through in every transmission, as some transmission's path may be directed by routers. Under a tree network environment, the coverage area of ZigBee networks will be inspected. In addition, analysis will also be done on received data: when a device moves out of a ZigBee network and moves in to another ZigBee network, and the end-to-end delay and medium access delay.

2. Initial project topic: BlueTooth™ Transmission Analysis

In the beginning, our topic for this project was originally on the analysis of Bluetooth Transmission. We wanted to simulate and analyze the performance of audio and video data transmission, via Bluetooth, in real-time environment. However, a contribution model, suite-tooth, was found and downloaded to assist in starting of our overall design. Unfortunately, the contribution model was in OPNET version 9, and it was not possible to update the contribution model in order to be compatible for OPNET version 14.

First, the Bluetooth process models were opened and compiled into OPNET version 14. Since some kernel procedures changed, the compiler delivered error messages on some of the source codes in each process model. We followed the kernel procedures document from version 14 and fixed the errors. Moreover, essential header files were moved into the same folder and had OPNET to read header files from the right path. The above steps fixed the compiling errors, and we tried to run sample scenario which came with the contribution model. However, the simulation could not get started due to various recovery errors. Further steps were not possible because there was not enough knowledge about OPNET and Bluetooth in order to trace the bugs. Therefore, a change of project focus was directed to ZigBee. The reason that ZigBee protocol transmission was chosen to be analyzed was because it has a similar personal area network protocol as Bluetooth; also, OPNET version 14 has ZigBee contribution model which help saved time from building our own under a tight schedule.

3. Overall Design: Node and Process Model

Since we use the contribution model that comes with OPNET, we provide node and process model in this section for completeness. Figure 3.1 shows the overview node model of the node, and this node model consists of a coordinator, router and end device. It is constructed with 3 layers: application, network and medium access control (MAC) layer. There is one wireless transmitter and receiver for ZigBee wireless communication.

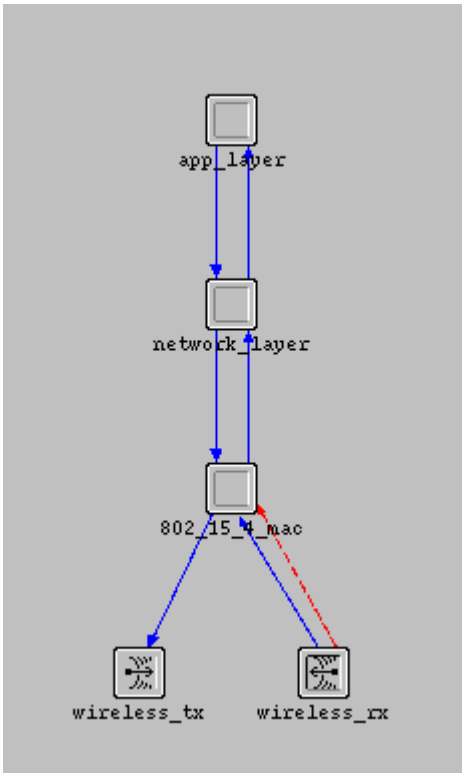

Figure 3.1 – Node Model Overview

Figure 3.2 shows the process model of the MAC, and it is the only process models that is provided by OPNET.

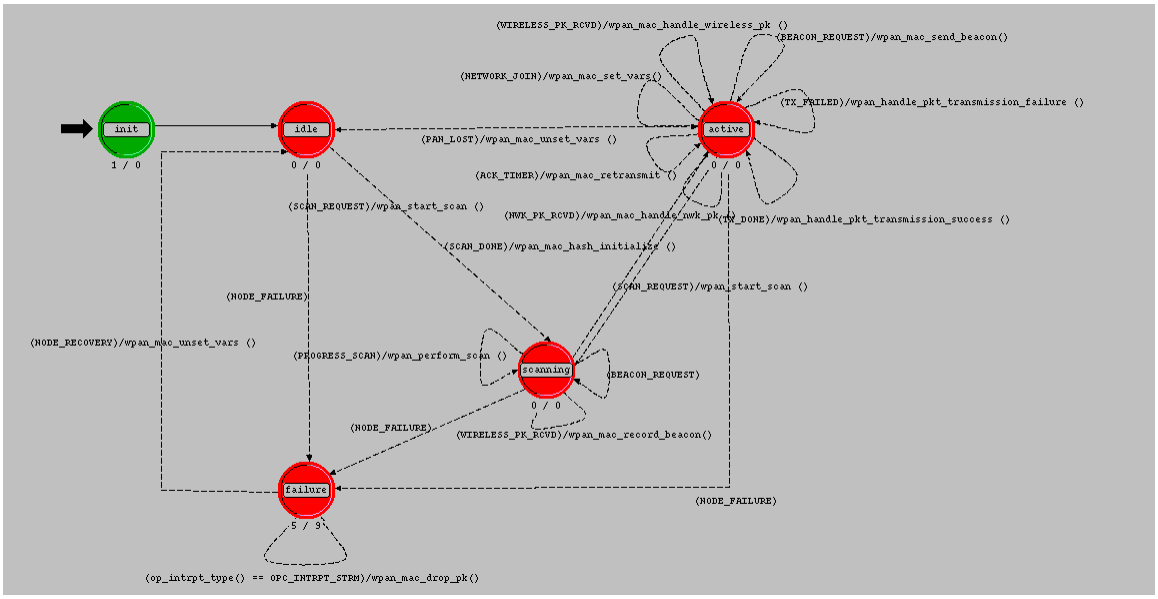

Figure 3.2 – MAC Process Model Overview

4. Project Simulation and Analysis

Five different scenarios were used to test and analyze ZigBee tree networks to thoroughly test its performance. The first scenario investigates the network coverage area, and the second scenario simulates the packet receive performance of a mobile device. The third and fourth scenarios deal with a mobile device moving from one ZigBee network to another, overall performance analysis of ZigBee networks in tree topology. Lastly, performance analysis was executed on the network after an extra device is added to a network to see how the overall network is affected. The following subsections lists out the step by step procedure involved in each of the five scenarios.

4.1 Scenario One: Network coverage

Before analyzing the network in tree topology, a few tasks must first be done, such as the inspection of the network coverage area. Practical ZigBee protocol has transmission power of 1mW, so the first step in this scenario is to set the coordinator transmission power to 1mW, as shown in figure 4.1. The general scenario layout is shown in figure 4.2, where the coordinator's ultimate destination is the mobile end device. The destination device's attributes are shown in figure 4.1 as well. The horizontal white line in figure 4.2 indicates the moving path of the mobile device, and the moving speed was set at 5m/s. Next, the packets received statistic of the mobile device was taken, and the result is shown in figure 4.3. The graph shows that the device does not receive any packet as it move out of the network coverage are. By using the simple

equation in Eq.4.1, the network coverage area is a circle with radius 5*32=160m at 1mW transmission power.

$$D = \text{Speed} \times \text{Time} \qquad \text{(Eq. 4.1)}$$

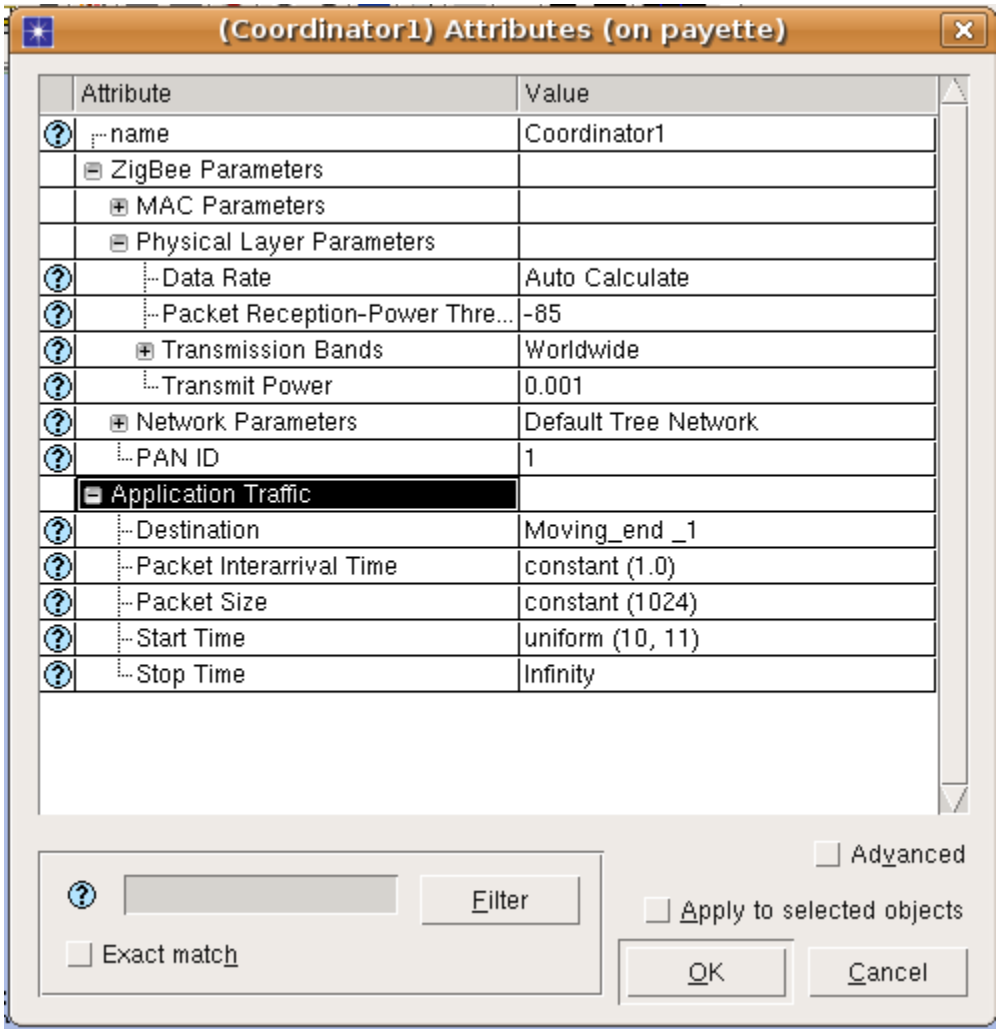

Figure 4.1 – Editing Attributes for the ZigBee Network Set-up

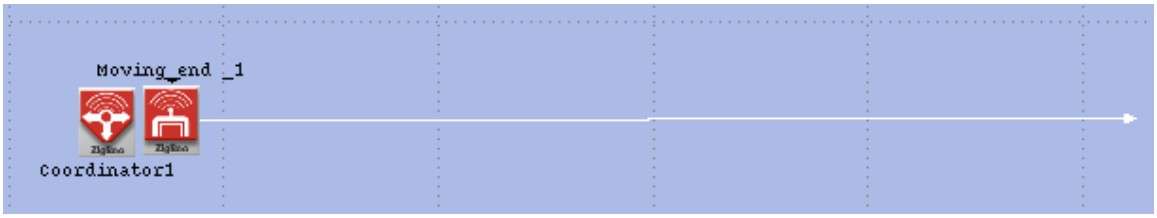

Figure 4.2 – Scenario Setup

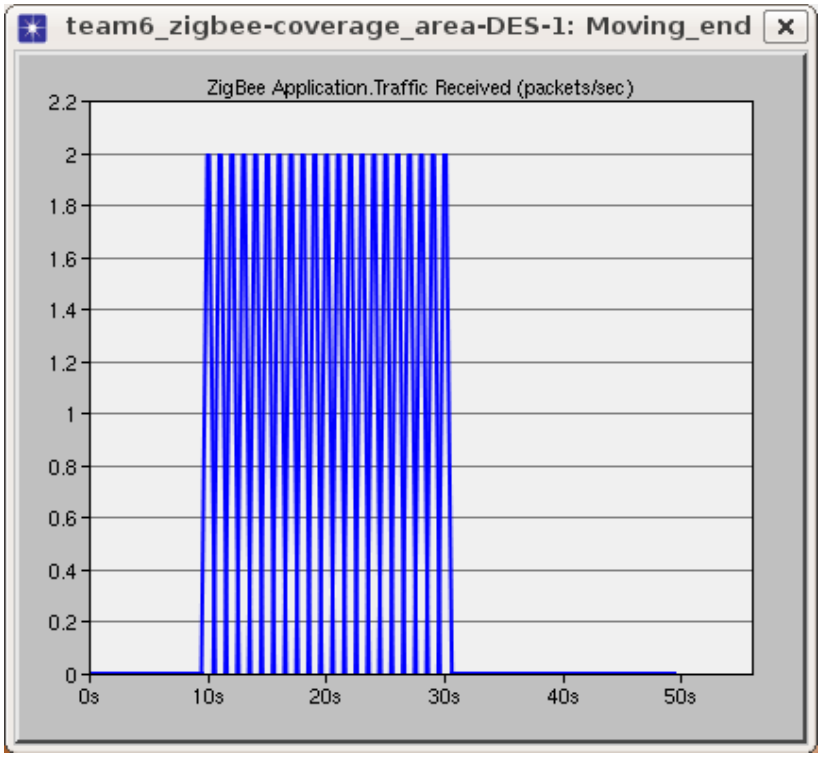

Figure 4.3 – Graph of Traffic Received for Mobile Device

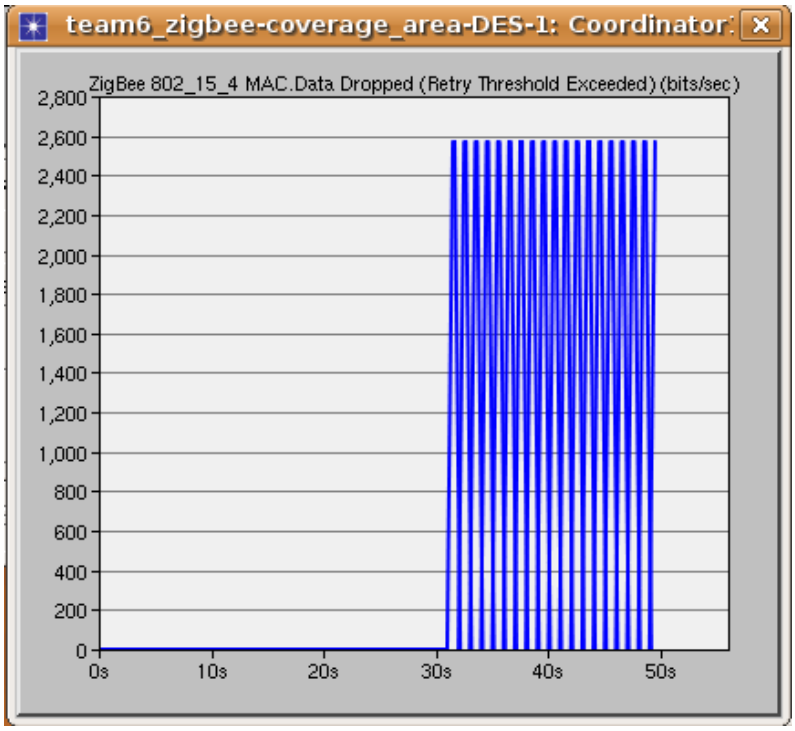

Figure 4.4 – Graph of Data Dropped for Mobile Device

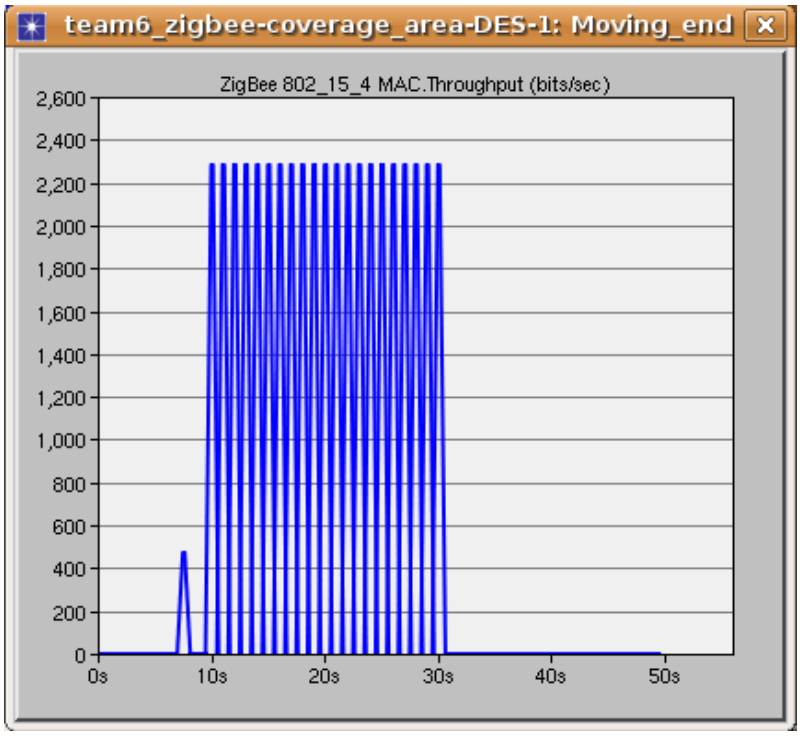

Figure 4.5 -Throughput Graph

Notice that the data dropped in figure 4.4 matches with the data received in figure 4.3. In addition, figure 4.5 illustrates the throughput for this scenario, and the first peak of throughput indicates the acknowledgement for setting up the communication medium.

4.2 Scenario Two: Packet receive analysis of a mobile device

In scenario two, which uses the same transmission power used in scenario one, a mobile device moves out of a ZigBee network, and moves back into the same network. In the general scenario setup diagram, shown in figure 4.6, the line indicates the moving path. The throughput of the second scenario is illustrated in figure 4.7, and the first peak is indicated the acknowledgement to the communication medium setup. The rest of the graph clearly shows that there is no throughput when the mobile device moves out of

range. However, it will pick up the signal again when it moves back into the coverage area. Figure 4.8 shows the data dropped in this scenario, and it matches the result we collected in the previous figure. Figure 4.9 demonstrates the traffics dropped and are not received by the mobile device. The dropped data fits into the empty slot in throughput traffic in figure 4.9.

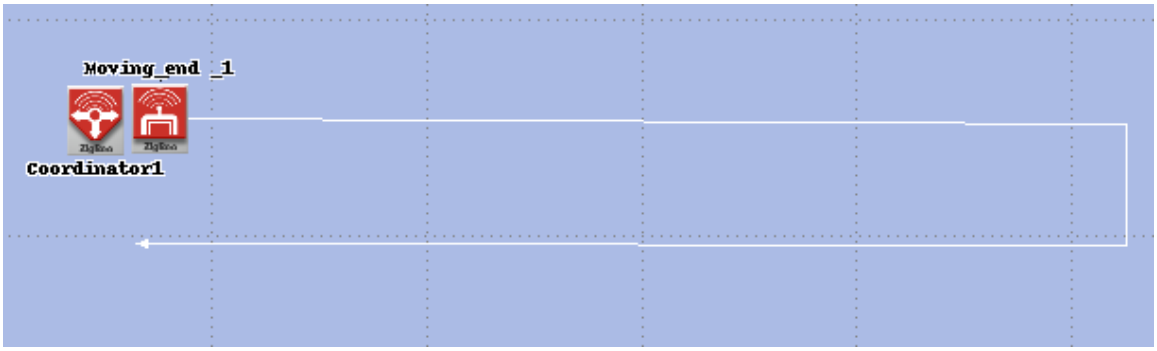

Figure 4.6 – General Scenario Setup

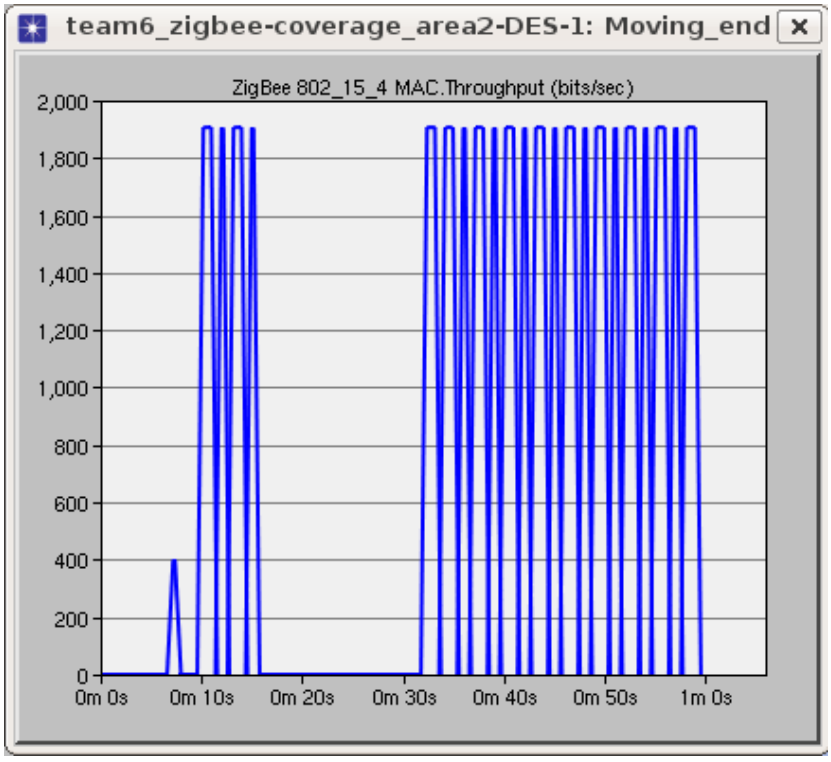

Figure 4.7 – Scenario Throughput for Moving In/Out of the Same Network

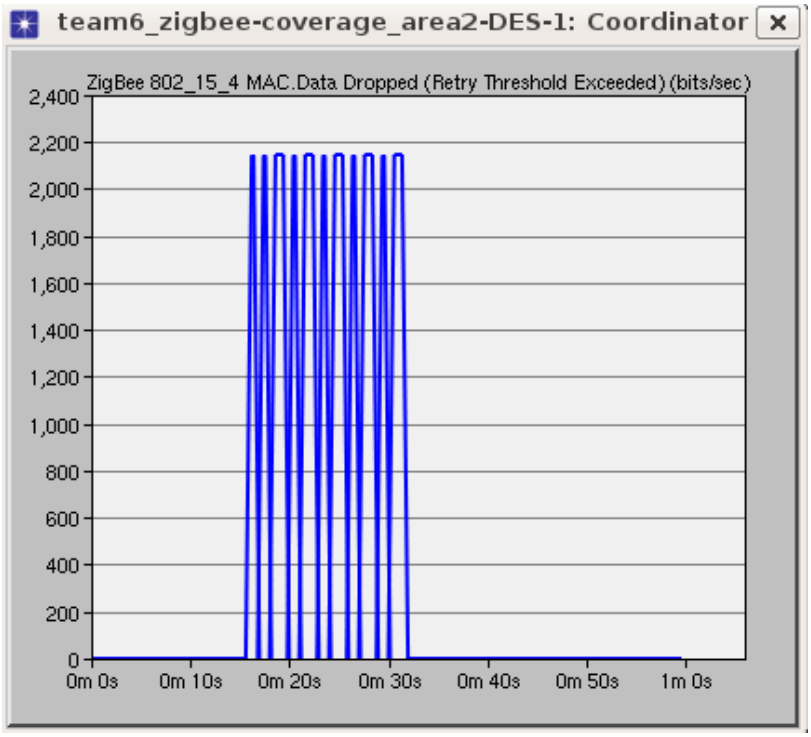

Figure 4.8 – Data Dropped in Scenario 2

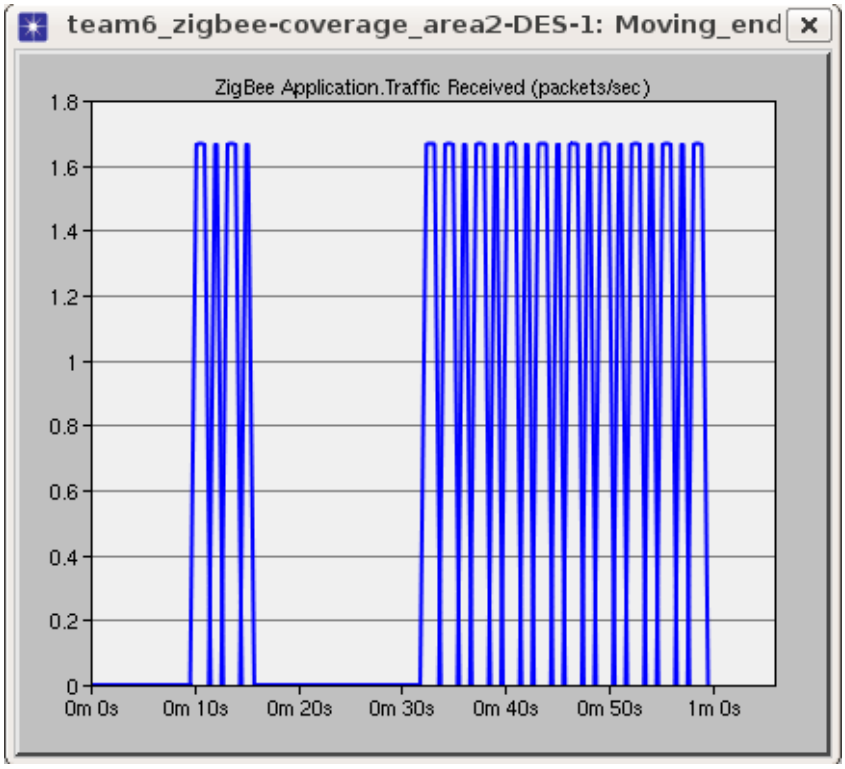

Figure 4.9 - Traffic Received by Mobile Device

4.3 Scenario Three: Mobile device moves from one network to another network

The third scenario investigates the transmission performance of ZigBee networks as mobile device moves from one network to another. Figure 4.10 shows the overview of the scenario setup. The top left network is assigned with a personal area network (PAN) of 1, and the bottom right network is assigned as PAN2. The mobile device moves from PAN1 to PAN2 as denoted by the white line. The transmission power of both networks is set to have no overlapping coverage to simplify the testing case. In addition, all nodes' destinations are set as the mobile device, so data will all be sent there. The resulting data received by the mobile device is shown in figure 4.11. We can observe that the mobile device will only pick up data from network 1, and it will not join network 2 automatically to receive data there. Handover setup is needed if a mobile device wants to join different networks at different time. Figure 4.12 shows the throughput rate to the mobile device as it moves from one network to another. There is no acknowledgement peak in this figure because no acknowledgement is present in this scenario, in fact, it is optional for ZigBee.

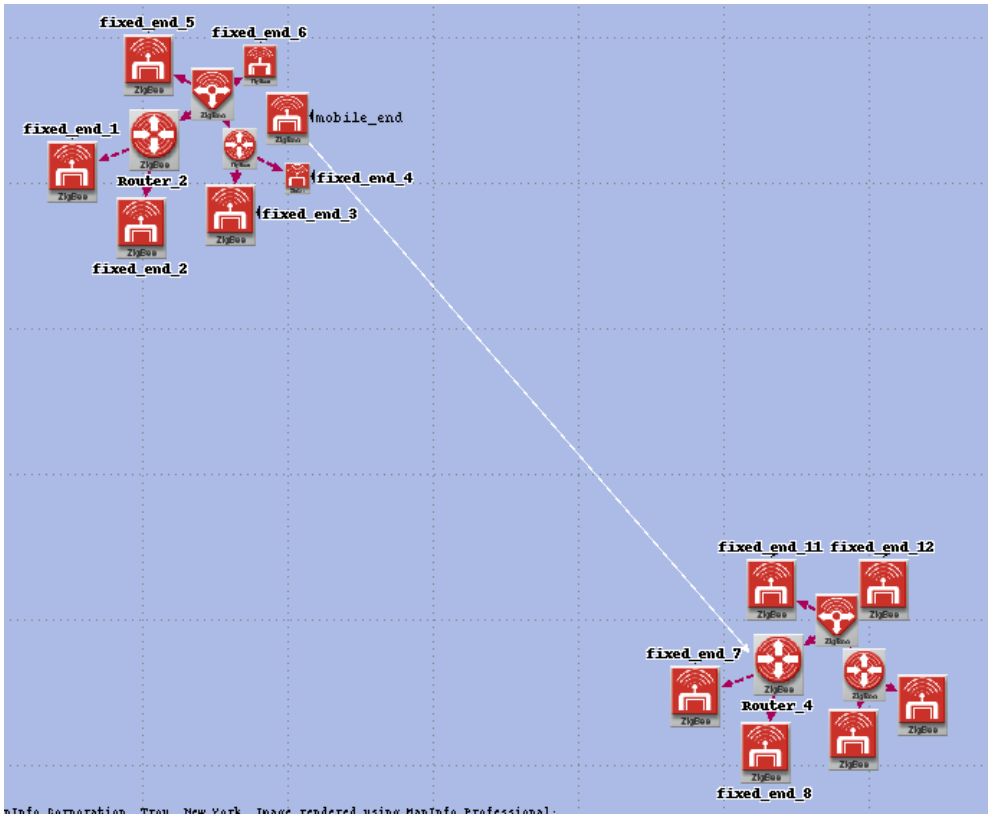

Figure 4.10 – Mobile Device Traveling From One Network to Another

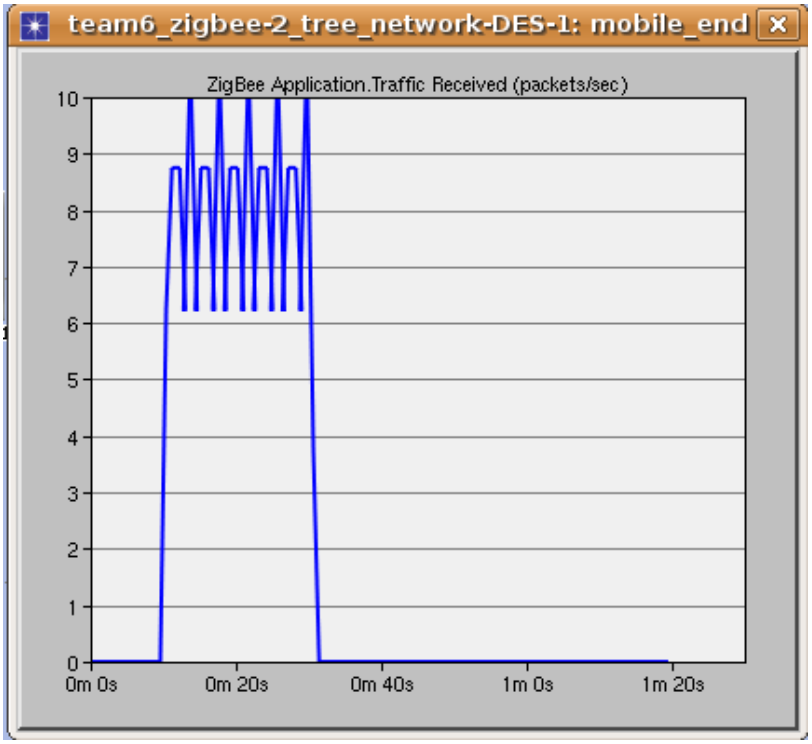

Figure 4.11 – Traffic Received by the Mobile Device

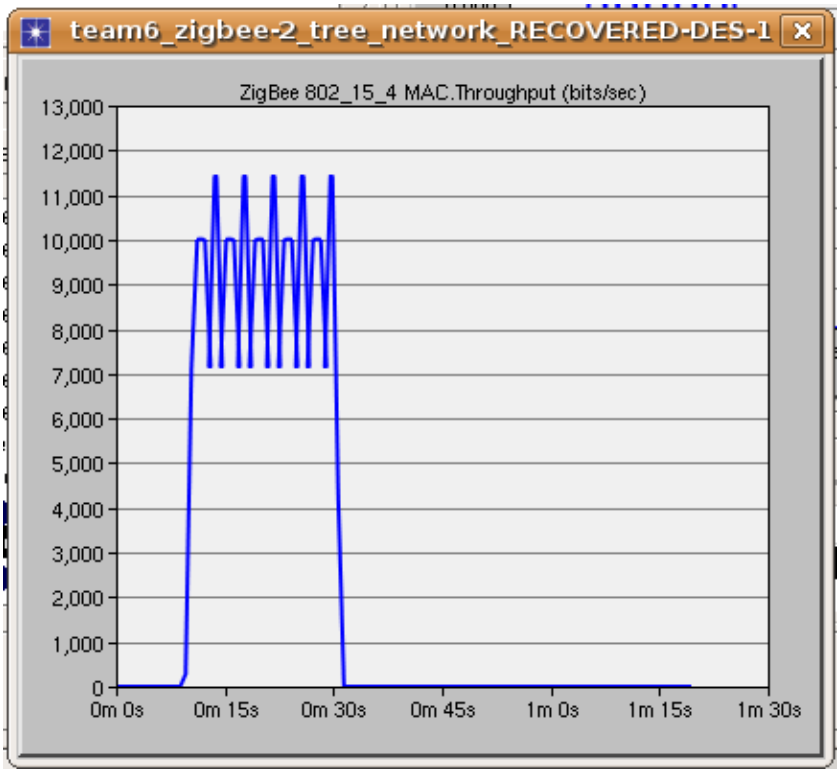

Figure 4.12 - Throughput to Mobile Device

4.4 Scenario Four: Performance analysis of ZigBee network in tree topology

The first step in analyzing the performance of ZigBee network in tree topology is to set up one coordinator, two routers, and four fixed end device as shown in figure 4.13. Each component's destinations of transmissions are set as random. The end-to-end delays of the three different transmissions are shown in Figure 4.14. The top line indicates the delay from end device 3 to end device 2, the middle line shows the delay from end device 1 to end device 2, and the bottom line is the delay from end device 6 to coordinator.

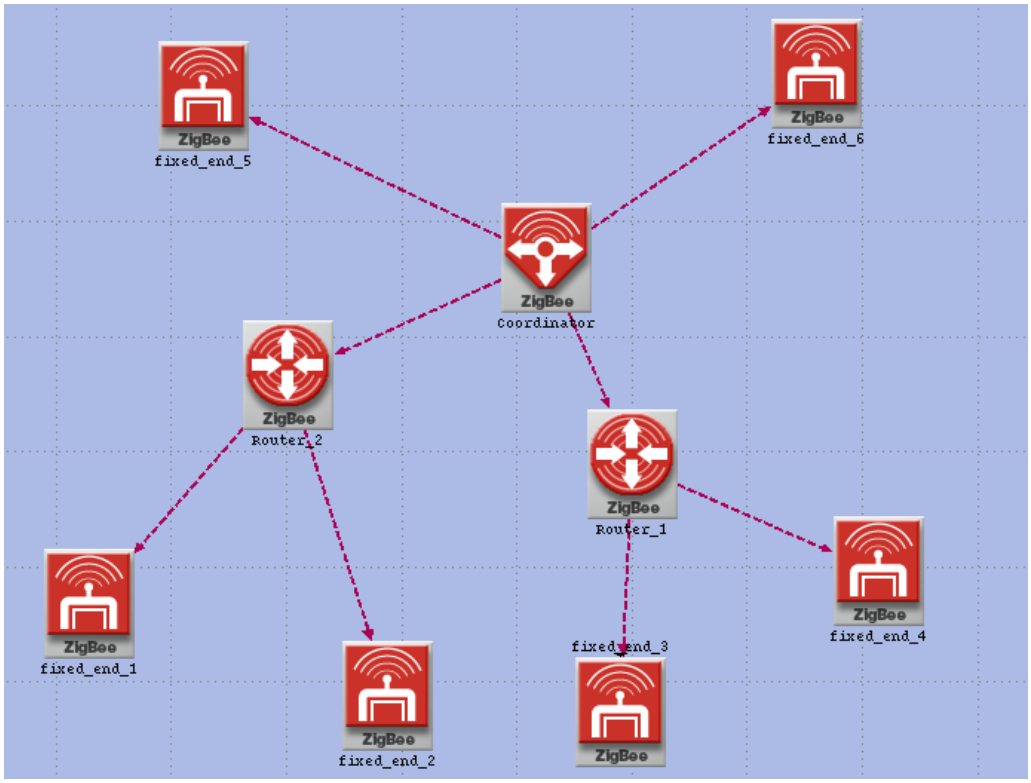

Figure 4.13– ZigBee Network in Tree Topology for Scenario 4

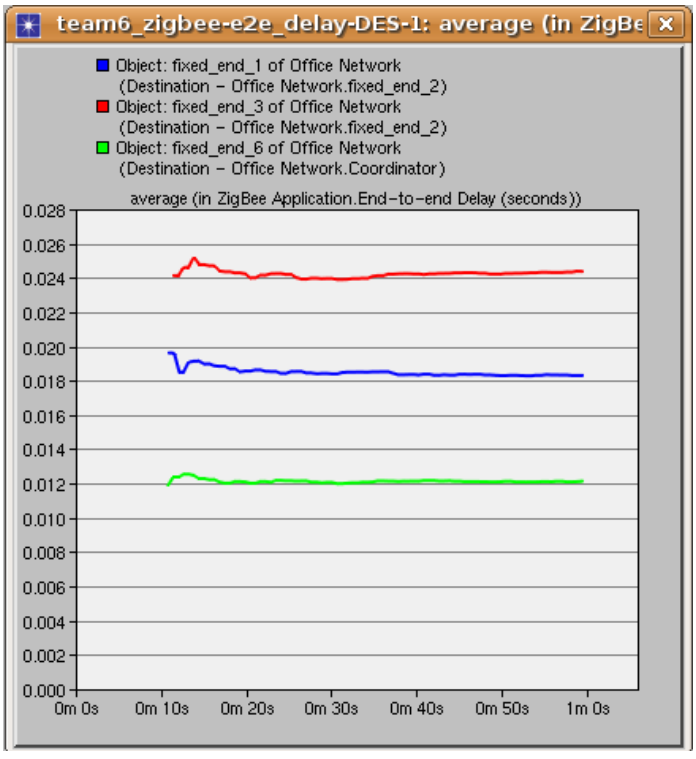

Figure 4.14 – End-to-End Delay of Different Devices

4.5 Scenario Five: Extra device is added to network in tree topology

Scenario five uses the same tree topology as the previous scenario, but with an extra end device which is not on the tree path added to it. Figure 4.15 shows the general setup of this scenario. All nodes, including the extra end device, are set to send data to end device 1. The end to end statistics are shown on figure 4.16 with different lines representing different nodes. Looking from top to bottom, the lines represent the delay of the additional node, end device 5, coordinator, end device 6, and router 2 respectively. It is obvious that the extra node has a higher end-to-end delay than other nodes.

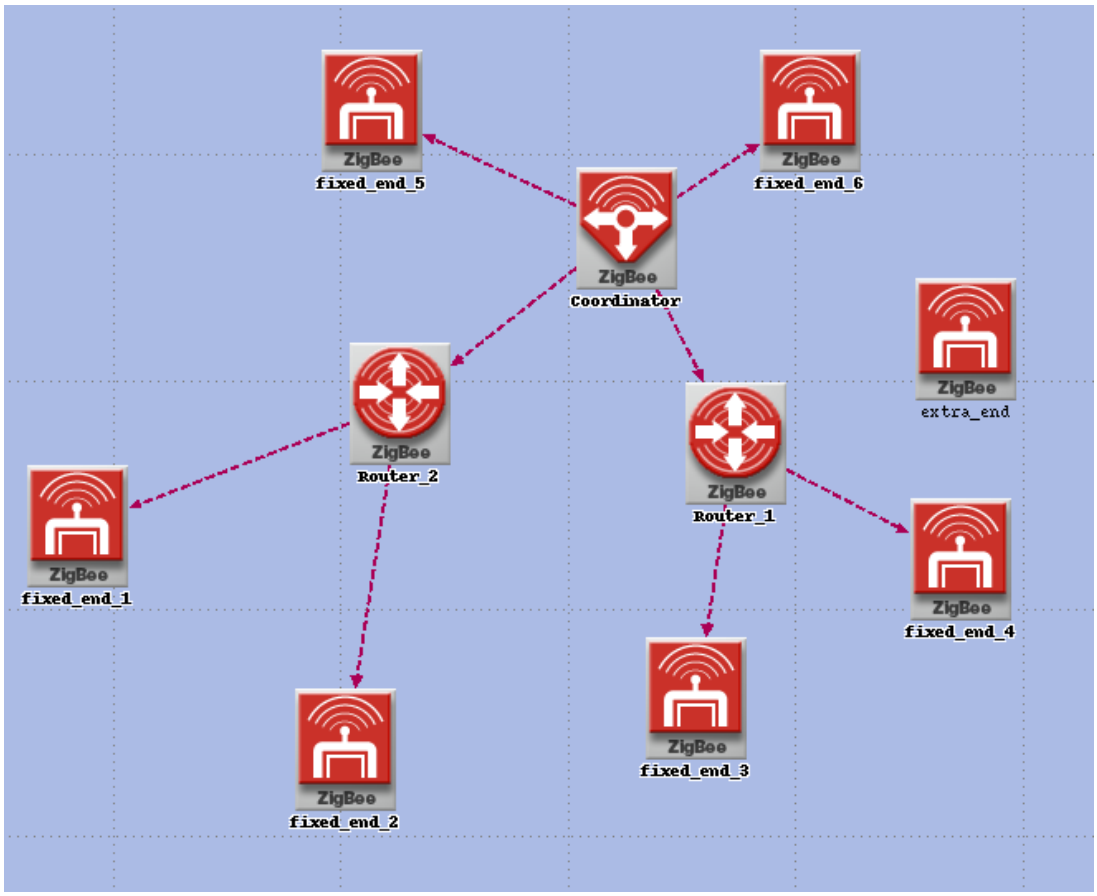

Figure 4.16 – General Scenario Setup

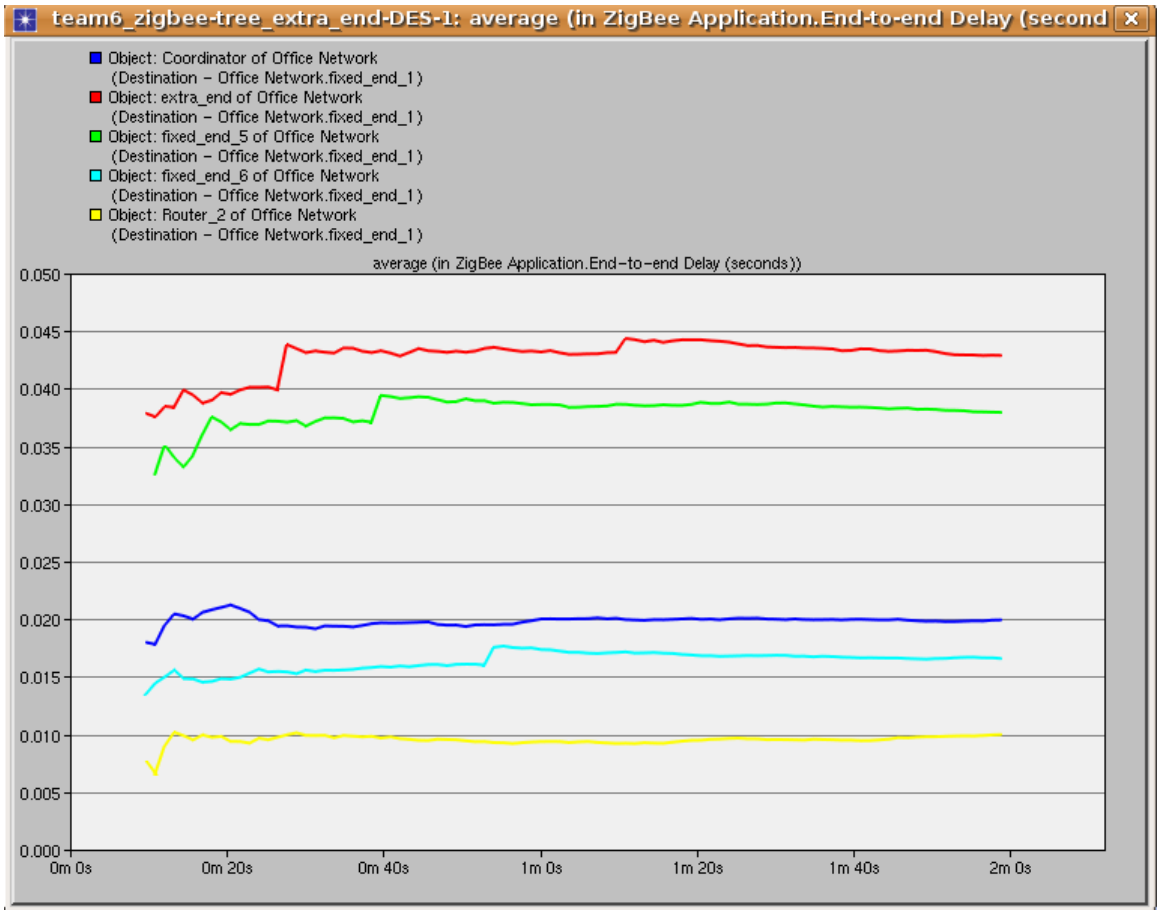

Figure 4.15 – End-to-end Delay of Different Nodes

5. Discussion and Conclusions

With the data collected from the 5 scenarios explained in the previous sections, a few observations and conclusions noted. At the typical transmission power of 1mW, the network coverage area of a ZigBee network is approximately 160 meters. In addition, expected results such as not receiving any data once the mobile device moves out of the coverage area, and handoffs are needed to transfer between different ZigBee networks were observed. In addition, it was noted that different routes resulted in different end-to-end delay because of the different status of the accessed mediums. For example, as the

number of hops increases in the path, the more likely it will result in a higher delay. Moreover, an end device that is not in the network has a higher end-to-end delay than those that are inside the network. Also, an end device cannot belong to two different networks simultaneously; it can only be under one network at one time. Lastly, it was observed that acknowledgement when setting up a connection is optional for ZigBee protocol, therefore it van be connection-oriented or connection-less oriented.

6. Future Work

ZigBee uses three different frequency bands for communication: 868, 915 and 2400 MHz, and 2400 MHz (2.4 GHz); they are commonly in use worldwide. As evident, Wi-Fi uses 2.4 GHz and it has been increasingly common. An interesting analysis of interference could be done with these known facts. We could simulate, in the future, ZigBee network under Wi-Fi environment with OPNET and analyze how they are affecting each other. The major data that can be collected are traffic packet drop and receive, throughput, end-to-end delay and traffic re-transmission. Moreover, due to limitations of time, we could not simulate and analyze the handover of a mobile device from one ZigBee network to another. However, we would like to perform this study in the future.

References

[1] Ahn S., Cho. J., & An S., “Slotted Beacon Scheduling Using ZigBee Cskip Mechanism,” Sensor Technologies and Applications, 2008. SENSORCOMM '08. Second International Conference, pp103-108. Aug 2008.

[2] Kim T., Kim D., Park N., Yoo S., & Lopesz T.S. “Shortcut tree routing in ZigBee network,” Wireless Pervasive Computing, 2007. ISWPC '07. 2nd International Symposium, Feb 2007.

[3] Li Weibo, Sirisena H., & Pawlkowski K., “An address base routing scheme for static applications of wireless sensor networks,” Telecommunication Networks and Application Conference, 2007. ATNAC 2007. Australasian, pp. 371-376, Dec 2007.

[4] Yeh L., Pan, M.S., & Tseng Y.C., “Two-way beacon scheduling in ZigBee tree-based wireless sensor networks,” 2008 IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing(SUTC '08), 130-7, June 2008.

[5]Yen L.H., & Tsai W.T., “Flexible address configuration for tree-based ZigBee/IEEE 802.15.4 wireless networks,” 2008 22nd International Conference on Advanced Information Networking and Application - Workshop, pp. 395-402, March 2008.

Appendix

Codes:

Referring to the process model, figure 3.2.

Function Block:

```
void
wpan_mac_init ()
	{

	Objid ack_objid;
	Objid ack_info_objid;
	double rx_power_threshold_dbm = 0.0;
	int data_rate;
	Packet * pk;

	/* The purpose of this function is to:		*/
	/*		- Register statistics			*/
	/*		- Initialize channels for scanning	*/
	/*		- Initialize state varibles		*/
	/*		- Create CSMA-CA child process		*/
	/*		- Read in MAC attributes		*/
	/*		- Initialize module-wide memory		*/

	FIN (wpan_mac_init ());

	/* Initialize packet index variables. */
	pk = op_pk_create_fmt ("zigbee_network_pdu");
	packet_type_index	= op_pk_nfd_name_to_index (pk, "packet_type");
	next_hop_address_index = op_pk_nfd_name_to_index (pk, "next_hop_address");
	stathandles_index	= op_pk_nfd_name_to_index (pk, "stathandles");
	pan_id_index		= op_pk_nfd_name_to_index (pk, "pan_id");
	payload_index	= op_pk_nfd_name_to_index (pk, "payload");
	op_pk_destroy (pk);

	pk = op_pk_create_fmt ("zigbee_beacon");
	WPANC_BEACON_PAN_ID_FIELD_INDEX		=
op_pk_nfd_name_to_index (pk, "pan_id");
	WPANC_BEACON_SRC_ADDR_FIELD_INDEX		=
op_pk_nfd_name_to_index (pk, "source_address");
	WPANC_BEACON_RTR_CAP_FIELD_INDEX		=
op_pk_nfd_name_to_index (pk, "router_capacity");
```

```
	WPANC_BEACON_END_DEVICE_CAP_FIELD_INDEX	=
op_pk_nfd_name_to_index (pk, "end_device_capacity");
	WPANC_BEACON_DEPTH_FIELD_INDEX	=
op_pk_nfd_name_to_index (pk, "depth");
	WPANC_BEACON_CAP_DURATION_FIELD_INDEX	=
op_pk_nfd_name_to_index (pk, "cap_duration");
	WPANC_BEACON_PARENT_BEACON_OFFSET_FIELD_INDEX =
op_pk_nfd_name_to_index (pk, "parent_beacon_offset");
	WPANC_BEACON_MAX_CHILDREN_FIELD_INDEX	=
op_pk_nfd_name_to_index (pk, "max_children");
	WPANC_BEACON_MAX_ROUTERS_FIELD_INDEX	=
op_pk_nfd_name_to_index (pk, "max_routers");
	WPANC_BEACON_MAX_DEPTH_FIELD_INDEX	=
op_pk_nfd_name_to_index (pk, "max_depth");
	WPANC_BEACON_BEACON_ORDER_FIELD_INDEX	=
op_pk_nfd_name_to_index (pk, "beacon_order");
	WPANC_BEACON_SUPERFRAME_ORDER_FIELD_INDEX	=
op_pk_nfd_name_to_index (pk, "superframe_order");
	op_pk_destroy (pk);

	pk = op_pk_create_fmt ("zigbee_beacon_request");
	/*  802.15.4 MAC Beacon Request Frame Fields */
	WPANC_BEACON_REQUEST_BAND_868_FIELD_INDEX  =
op_pk_nfd_name_to_index (pk, "band_868");
	WPANC_BEACON_REQUEST_BAND_915_FIELD_INDEX  =
op_pk_nfd_name_to_index (pk, "band_915");
	WPANC_BEACON_REQUEST_BAND_2450_FIELD_INDEX =
op_pk_nfd_name_to_index (pk, "band_2450");
	op_pk_destroy (pk);

	pk = op_pk_create_fmt ("zigbee_ack");
	WPANC_ACK_PAN_ID_FIELD_INDEX	= op_pk_nfd_name_to_index
(pk, "pan_id");
	WPANC_ACK_NEXT_HOP_ADDR_FIELD_INDEX =
op_pk_nfd_name_to_index (pk, "next_hop_address");
	WPANC_ACK_SEQ_NUM_FIELD_INDEX	=
op_pk_nfd_name_to_index (pk, "sequence_number");
	op_pk_destroy (pk);

	/* Initialize state variables */
	beacon_scan_list	= OPC_NIL;
	my_pan_id		= -1;
	my_network_address	= -1;
	my_parent_address	= -1;
	my_channel		= -1;
	stat_dim_index		= 0;
```

```
	outstanding_pk_lptr = op_prg_list_create ();
	self_id = op_id_self();
	nwk_id = op_id_from_name (op_topo_parent (op_id_self ()),
OPC_OBJTYPE_PROC, "network_layer");
	parent_id = op_topo_parent (self_id);

	/* Initialize categorized memory handle for the WPAN MAC */
	wpan_cmo_handle = op_prg_cmo_define ("WPAN MAC");

	/* Read in MAC attributes.    */
	op_ima_obj_attr_get (self_id, "Data Rate", &data_rate);
	op_ima_obj_attr_get (self_id, "Channel Sensing Duration",
&channel_sensing_duration);
	op_ima_obj_attr_get (self_id, "ACK Mechanism", &ack_objid);
	ack_info_objid = op_topo_child (ack_objid, OPC_OBJTYPE_GENERIC, 0);
	op_ima_obj_attr_get (ack_info_objid, "Status", &ack_enabled);
	op_ima_obj_attr_get (ack_info_objid, "ACK Wait Duration", &ack_duration);
	op_ima_obj_attr_get (ack_info_objid, "Number of Retransmissions",
&max_retrans);

	/* Convert the power threshold (receiver sensitivity) value from dBm to    */
	/* Watts.
                                                */
	op_ima_obj_attr_get (self_id, "Packet Reception-Power Threshold",
&rx_power_threshold_dbm);

	/* Convert the power threshold (receiver sensitivity) value from dBm to    */
	/* Watts.
                                                */
	modmem_ptr = (WPAN_Modmem_Info*) op_prg_cmo_alloc
(wpan_cmo_handle,sizeof (WPAN_Modmem_Info));
	modmem_ptr->rx_power_threshold = pow (10.0, rx_power_threshold_dbm /
10.0) / 1000.0;
	modmem_ptr->retry_csma_trans_pkptr = OPC_NIL;
	op_pro_modmem_install (modmem_ptr);

	/* Get the channel band for this device and populate                    */
	/* the channel info structures for the applicable channels    */
	wpan_mac_initialize_channels (parent_id, data_rate);

	wpan_mac_register_stats ();

	/* Create the process handling CSMA-CA    */
	csma_ca_prohandle = op_pro_create ("csma_ca", OPC_NIL);
	csma_ca_process_busy = OPC_FALSE;
```

```
	/* Affliated with a PAN?	*/
	joined = OPC_FALSE;

	FOUT;
	}


void
wpan_mac_register_stats ()
	{
	int index = 0;

	/** This function register both local and global statistics associated with this
MAC */

	FIN (wpan_mac_register_stats ());

	/* Register global statistics */
	gbl_thput_bps_stathandle =		op_stat_reg ("ZigBee 802_15_4
MAC.Throughput (bits/sec)",
		OPC_STAT_INDEX_NONE,	OPC_STAT_GLOBAL);
	gbl_num_retrans_stathandle =		op_stat_reg ("ZigBee 802_15_4
MAC.Retransmission Attempts (packets)",
	OPC_STAT_INDEX_NONE,	OPC_STAT_GLOBAL);
	gbl_data_dropped_stathandle =		op_stat_reg ("ZigBee 802_15_4
MAC.Data Dropped (Retry Threshold Exceeded) (bits/sec)",
	OPC_STAT_INDEX_NONE,	OPC_STAT_GLOBAL);
	gbl_data_rcvd_stathandle =		op_stat_reg ("ZigBee 802_15_4
MAC.Data Traffic Rcvd (bits/sec)",
	OPC_STAT_INDEX_NONE,	OPC_STAT_GLOBAL);
	gbl_mgmt_data_rcvd_stathandle =	op_stat_reg ("ZigBee 802_15_4
MAC.Management Traffic Rcvd (bits/sec)",
	OPC_STAT_INDEX_NONE,	OPC_STAT_GLOBAL);
	gbl_control_data_rcvd_stathandle =	op_stat_reg ("ZigBee 802_15_4
MAC.Control Traffic Rcvd (bits/sec)",
	OPC_STAT_INDEX_NONE,	OPC_STAT_GLOBAL);
	gbl_e2e_stathandle =			op_stat_reg ("ZigBee 802_15_4
MAC.Delay (sec)",
		OPC_STAT_INDEX_NONE,	OPC_STAT_GLOBAL);
	gbl_load_stathandle =				op_stat_reg ("ZigBee
802_15_4 MAC.Load (bits/sec)",
			OPC_STAT_INDEX_NONE,	OPC_STAT_GLOBAL);
	gbl_control_data_sent_stathandle =	op_stat_reg ("ZigBee 802_15_4
MAC.Control Traffic Sent (bits/sec)",
	OPC_STAT_INDEX_NONE,	OPC_STAT_GLOBAL);
```

```
	gbl_mgmt_data_sent_stathandle =	op_stat_reg ("ZigBee 802_15_4
MAC.Management Traffic Sent (bits/sec)",
	OPC_STAT_INDEX_NONE,	OPC_STAT_GLOBAL);

	op_stat_dim_size_get("ZigBee 802_15_4 MAC.Load per PAN (bits/sec)",
OPC_STAT_GLOBAL, &stat_dim_size);
	for (index = 0; index < stat_dim_size; index++)
		{
		gbl_load_dim_stathandle [index] = *((Stathandle*) op_prg_cmo_alloc
(wpan_cmo_handle, sizeof (Stathandle)));
		pan_stat_handle_array [index]  = -1;
		}


	/* Register local statistics */
	thput_bps_stathandle =				op_stat_reg ("ZigBee
802_15_4 MAC.Throughput (bits/sec)",
			OPC_STAT_INDEX_NONE,	OPC_STAT_LOCAL);
	num_retrans_stathandle =		op_stat_reg ("ZigBee 802_15_4
MAC.Retransmission Attempts (packets)",
	OPC_STAT_INDEX_NONE,	OPC_STAT_LOCAL);
	data_dropped_stathandle =		op_stat_reg ("ZigBee 802_15_4
MAC.Data Dropped (Retry Threshold Exceeded) (bits/sec)",
	OPC_STAT_INDEX_NONE,	OPC_STAT_LOCAL);
	data_rcvd_stathandle =				op_stat_reg ("ZigBee
802_15_4 MAC.Data Traffic Rcvd (bits/sec)",
		OPC_STAT_INDEX_NONE,	OPC_STAT_LOCAL);
	mgmt_data_rcvd_stathandle =		op_stat_reg ("ZigBee 802_15_4
MAC.Management Traffic Rcvd (bits/sec)",
	OPC_STAT_INDEX_NONE,	OPC_STAT_LOCAL);
	control_data_rcvd_stathandle =	op_stat_reg ("ZigBee 802_15_4
MAC.Control Traffic Rcvd (bits/sec)",
	OPC_STAT_INDEX_NONE,	OPC_STAT_LOCAL);
	load_stathandle =				op_stat_reg ("ZigBee
802_15_4 MAC.Load (bits/sec)",
			OPC_STAT_INDEX_NONE,	OPC_STAT_LOCAL);
	e2e_stathandle =				op_stat_reg ("ZigBee
802_15_4 MAC.Delay (sec)",
				OPC_STAT_INDEX_NONE,
	OPC_STAT_LOCAL);
	mgmt_data_sent_stathandle =		op_stat_reg ("ZigBee 802_15_4
MAC.Management Traffic Sent (bits/sec)",
	OPC_STAT_INDEX_NONE,	OPC_STAT_LOCAL);
	control_data_sent_stathandle =		op_stat_reg ("ZigBee 802_15_4
MAC.Control Traffic Sent (bits/sec)",
	OPC_STAT_INDEX_NONE,	OPC_STAT_LOCAL);
```

```
	q_size_stathandle =			op_stat_reg ("ZigBee 802_15_4
MAC.Queue Size (packets)",
	OPC_STAT_INDEX_NONE,	OPC_STAT_LOCAL);
	qdelay_stathandle =			op_stat_reg ("ZigBee 802_15_4
MAC.Queuing Delay (sec)",
	OPC_STAT_INDEX_NONE,	OPC_STAT_LOCAL);

	FOUT;
	}

void
wpan_mac_drop_pk ()
	{
	Packet *pk, * encap_pk;
	Stathandle_Bundle * stat_bun_ptr;

	/** Destroy the packet and write the Application Data Dropped statisitics. **/

	FIN (wpan_mac_drop_pk ());

	pk = op_pk_get (op_intrpt_strm ());

	if (wpan_network_pk_is_app_data (pk))
		{
		op_pk_fd_get_pkt (pk, payload_index, &encap_pk);
		op_pk_fd_get_ptr (encap_pk, stathandles_index, (void *) &stat_bun_ptr);

		/* Write to application layer packets dropped stathandle. */
		op_stat_write (*(stat_bun_ptr->sh1), 1.0);

		/* Free the statbundle. */
		op_prg_cmo_dealloc  (stat_bun_ptr);
		}

	op_pk_destroy (pk);

	FOUT;
	}



void
wpan_mac_record_beacon ()
	{
	Packet *pk;
	Neighbor_Entry * entry_ptr;
```

```
	char format_name[100];
	int temp = 0, temp2 = 0, i;
	double temp_dbl = 0.0;
	char message_str [255];
	double pk_size = 0.0;

	/** This function store the information in the incoming beacon to be later sent to
the Network layer **/

	FIN (wpan_mac_record_beacon ());

	pk = op_pk_get (op_intrpt_strm ());
	op_pk_format (pk, format_name);

	pk_size = (double) op_pk_total_size_get (pk);
	wpan_mac_mgmt_rcvd_stats_update (pk_size);

	/* Check packet type. */
	if (strcmp (format_name, "zigbee_beacon") == 0)
		{
		op_pk_fd_get_int32 (pk, WPANC_BEACON_PAN_ID_FIELD_INDEX,
&temp);
		op_pk_fd_get_int32 (pk,
WPANC_BEACON_SRC_ADDR_FIELD_INDEX, &temp2);

		for (i = 0; i < op_prg_list_size (beacon_scan_list); i++)
			{
			entry_ptr = (Neighbor_Entry *) op_prg_list_access
(beacon_scan_list, i);
			if (entry_ptr->pan_id == temp && entry_ptr->source_address ==
temp2)
				{
				op_pk_destroy (pk);
				FOUT;
				}
			}

		/* Store the beacon information in a list */
		entry_ptr = (Neighbor_Entry *) op_prg_cmo_alloc
(wpan_cmo_handle,sizeof (Neighbor_Entry));

		op_pk_fd_get_int32 (pk, WPANC_BEACON_PAN_ID_FIELD_INDEX,
&temp);
		entry_ptr->pan_id = temp;
```

```
		op_pk_fd_get_int32 (pk,
WPANC_BEACON_SRC_ADDR_FIELD_INDEX, &temp);
		entry_ptr->source_address = temp;

		op_pk_fd_get_int32 (pk,
WPANC_BEACON_RTR_CAP_FIELD_INDEX, &temp);
		entry_ptr->router_capacity = temp;

		op_pk_fd_get_int32 (pk,
WPANC_BEACON_END_DEVICE_CAP_FIELD_INDEX, &temp);
		entry_ptr->end_device_capacity = temp;

		op_pk_fd_get_int32 (pk, WPANC_BEACON_DEPTH_FIELD_INDEX,
&temp);
		entry_ptr->depth = temp;

		op_pk_fd_get_dbl (pk,
WPANC_BEACON_PARENT_BEACON_OFFSET_FIELD_INDEX, &temp_dbl);
		entry_ptr->parent_beacon_offset = temp_dbl;

		op_pk_fd_get_int32 (pk,
WPANC_BEACON_MAX_CHILDREN_FIELD_INDEX, &temp);
		entry_ptr->max_children = temp;

		op_pk_fd_get_int32 (pk,
WPANC_BEACON_MAX_ROUTERS_FIELD_INDEX, &temp);
		entry_ptr->max_routers = temp;

		op_pk_fd_get_int32 (pk,
WPANC_BEACON_MAX_DEPTH_FIELD_INDEX, &temp);
		entry_ptr->max_depth = temp;

		op_pk_fd_get_int32 (pk,
WPANC_BEACON_BEACON_ORDER_FIELD_INDEX, &temp);
		entry_ptr->beacon_order = temp;

		op_pk_fd_get_int32 (pk,
WPANC_BEACON_SUPERFRAME_ORDER_FIELD_INDEX, &temp);
		entry_ptr->superframe_order = temp;

		entry_ptr->potential_parent    = OPC_TRUE;
		entry_ptr->channel                        = channel_number;
		entry_ptr->time_received    = op_sim_time();

		op_prg_list_insert (beacon_scan_list, entry_ptr, OPC_LISTPOS_TAIL);
```

```
		if (op_prg_odb_ltrace_active ("wpan_mac") || op_prg_odb_ltrace_active
("wpan_mac_join"))
			{
			sprintf (message_str, "Recorded a beacon.\n");
			op_prg_odb_print_major (message_str, OPC_NIL, OPC_NIL,
OPC_NIL);
			}
		}

	op_pk_destroy (pk);

	FOUT;
	}

void
wpan_mac_handle_wireless_pk ()
	{

	Packet		*pk;
	char		format_name [100];
	int		pk_pan_id;
	int		pk_next_hop;
	int		pk_source_addr;
	char		message_str [255];
	int		pk_seq_number;
	double		pk_size;
	WPAN_Transmission_Info* trans_info_ptr;

	/** This function handles all the packets recieved from the lower layers in the
joined state. **/

	FIN (wpan_mac_handle_wireless_pk ());

	pk = op_pk_get (op_intrpt_strm ());
	pk_size = (double) op_pk_total_size_get (pk);

	/* If packet is node's own transmission then drop the packet. */
	if (op_pk_stamp_mod_get (pk) == self_id)
		{
		op_pk_destroy (pk);
		FOUT;
		}

	/* Handle the packet based on its type.		*/
	op_pk_format (pk, format_name);
```

```
	if (strcmp (format_name, "zigbee_beacon") == 0)
		{
		op_pk_fd_get_int32 (pk, WPANC_BEACON_PAN_ID_FIELD_INDEX,
&pk_pan_id);
		op_pk_fd_get_int32 (pk,
WPANC_BEACON_SRC_ADDR_FIELD_INDEX, &pk_source_addr);

		wpan_mac_mgmt_rcvd_stats_update (pk_size);

		op_pk_destroy (pk);
		}
	else if (strcmp (format_name, "zigbee_beacon_request") == 0)
		{

		wpan_mac_mgmt_rcvd_stats_update (pk_size);

		/* If it's a beacon request, send it up if we're joined. */
		if (my_pan_id != -1)
			{
			op_pk_send (pk, NWK_STRM);

			if (op_prg_odb_ltrace_active ("wpan_mac") ||
op_prg_odb_ltrace_active ("wpan_mac_join") || op_prg_odb_ltrace_active
("wpan_mac_br"))
				{
				sprintf (message_str, "Recieved beacon request, sending it
to the upper layer to generate a beacon.\n");
				op_prg_odb_print_major (message_str, OPC_NIL,
OPC_NIL, OPC_NIL);
				}
			}
		else
			op_pk_destroy (pk);
		}
	else if (strcmp (format_name, "zigbee_ack") == 0)
		{
		/* Retrieve relevant information from the packet.	*/
		op_pk_fd_get (pk, WPANC_ACK_PAN_ID_FIELD_INDEX,
&pk_pan_id);
		op_pk_fd_get (pk,
WPANC_ACK_NEXT_HOP_ADDR_FIELD_INDEX, &pk_next_hop);
		op_pk_fd_get (pk, WPANC_ACK_SEQ_NUM_FIELD_INDEX,
&pk_seq_number);

		/* Update recieved stats */
		wpan_mac_control_rcvd_stats_update (pk_size);
```

```
		/* Process this ACK message only if it is destined for this node. */
		if ((pk_pan_id == my_pan_id && pk_next_hop == 
my_network_address))
			{
			/* Retrieve the hash table entry and free its memory. */
			trans_info_ptr = (WPAN_Transmission_Info*) 
prg_bin_hash_table_item_remove (ack_outstanding_htable_ptr, &pk_seq_number);

			if (trans_info_ptr)
				{
				wpan_trans_info_free (trans_info_ptr);

				if (op_prg_odb_ltrace_active ("wpan_mac") || 
op_prg_odb_ltrace_active ("wpan_mac_ack_retrans"))
					{
					sprintf (message_str, "Recieved an ACK for 
sequence number %d at %d.\n", pk_seq_number, my_network_address);
					op_prg_odb_print_major (message_str, OPC_NIL, 
OPC_NIL, OPC_NIL);
					}
				}
			else
				{
				if (op_prg_odb_ltrace_active ("wpan_mac") || 
op_prg_odb_ltrace_active ("wpan_mac_ack_retrans"))
					{
					sprintf (message_str, "Duplicate ACK for sequence 
number %d recieved at %d.\n", pk_seq_number, my_network_address);
					op_prg_odb_print_major (message_str, OPC_NIL, 
OPC_NIL, OPC_NIL);
					}
				}
			}

		/* Destroy the original ACK packet. */
		op_pk_destroy (pk);
		}
	else
		{
		/* This is a data packet. */
		op_pk_fd_get_int32 (pk, pan_id_index, &pk_pan_id);
		op_pk_fd_get_int32 (pk, next_hop_address_index, &pk_next_hop);

		/* Update recieved stats */
		wpan_mac_rcvd_stats_update (pk_size);
```

```
		/* Process the packet only if it is destined for this node or is a broadcast
packet. */
		if ((pk_pan_id == my_pan_id && pk_next_hop == my_network_address)
||
			(my_pan_id == -1 && pk_next_hop == my_network_address)
		 ||
			(pk_pan_id == my_pan_id && pk_next_hop ==
BROADCAST_CODE_ALL) ||
			(pk_pan_id == my_pan_id && pk_next_hop ==
BROADCAST_CODE_RC))
			{
			if (op_prg_odb_ltrace_active ("wpan_mac"))
				{
				sprintf (message_str, "Received a packet from the wireless
medium, sending it upto the network layer.\n");
				op_prg_odb_print_major (message_str, OPC_NIL,
OPC_NIL, OPC_NIL);
				}

			/* Reduce the packet size of the physical layer overhead */
			pk_size = op_pk_total_size_get (pk) -
WPAN_MAC_DATA_OVERHEAD;
			op_pk_total_size_set (pk, (OpT_Packet_Size) pk_size);

			/* Update the throughput and delay statistics. */
			wpan_mac_thput_and_e2e_stats_update (pk);

			/* Send an ACK if the sender has requested fot it.     */
			if (op_pk_fd_is_set (pk, WPANC_MAC_ACK_FIELD_INDEX)
== OPC_TRUE)
				wpan_mac_send_ack (pk);


			/* Strip the packet of the fields that have local significance and
may affect */
			/* the MAC behavior once the packet is resent to this MAC by the
upper layer  */
			wpan_mac_fields_strip (pk);

			/* Send the packet to network layer */
			op_pk_send (pk, NWK_STRM);
			}
		else
			{
			op_pk_destroy (pk);
```

```
                        }
                }

        FOUT;
        }


void
wpan_mac_send_beacon ()
        {

        Packet *pk;
        Beacon_Format * beacon_contents;
        char message_str [255];
        int pk_size;

        /** This function will broadcast a beacon based on the information recieved from
the network layer. **/

        FIN (wpan_mac_send_beacon ());

        /* Get the information sent by the network layer and create an beacon packet
appropriately. */
        beacon_contents = (Beacon_Format *) op_intrpt_state_ptr_get ();

        pk = op_pk_create_fmt ("zigbee_beacon");

        op_pk_fd_set_int32    (pk, WPANC_BEACON_PAN_ID_FIELD_INDEX,
                               beacon_contents->pan_id,
        OPC_FIELD_SIZE_UNCHANGED);
        op_pk_fd_set_int32    (pk, WPANC_BEACON_SRC_ADDR_FIELD_INDEX,
                        beacon_contents->source_address,
        OPC_FIELD_SIZE_UNCHANGED);
        op_pk_fd_set_int32    (pk, WPANC_BEACON_RTR_CAP_FIELD_INDEX,
                               beacon_contents->router_capacity,
        OPC_FIELD_SIZE_UNCHANGED);
        op_pk_fd_set_int32    (pk,
WPANC_BEACON_END_DEVICE_CAP_FIELD_INDEX,
        beacon_contents->end_device_capacity,
        OPC_FIELD_SIZE_UNCHANGED);
        op_pk_fd_set_int32    (pk,
WPANC_BEACON_MAX_CHILDREN_FIELD_INDEX,
        beacon_contents->max_children,
        OPC_FIELD_SIZE_UNCHANGED);
        op_pk_fd_set_int32    (pk,
WPANC_BEACON_MAX_ROUTERS_FIELD_INDEX,
```

```
	beacon_contents->max_routers,
	OPC_FIELD_SIZE_UNCHANGED);
	op_pk_fd_set_int32	(pk, WPANC_BEACON_MAX_DEPTH_FIELD_INDEX,
				beacon_contents->max_depth,
	OPC_FIELD_SIZE_UNCHANGED);
	op_pk_fd_set_int32	(pk,
WPANC_BEACON_BEACON_ORDER_FIELD_INDEX,
	beacon_contents->beacon_order,
	OPC_FIELD_SIZE_UNCHANGED);
	op_pk_fd_set_int32	(pk,
WPANC_BEACON_SUPERFRAME_ORDER_FIELD_INDEX,  beacon_contents-
>superframe_order,		OPC_FIELD_SIZE_UNCHANGED);
	op_pk_fd_set_dbl	(pk,
WPANC_BEACON_PARENT_BEACON_OFFSET_FIELD_INDEX,
	beacon_contents->parent_beacon_offset,
	OPC_FIELD_SIZE_UNCHANGED);
	op_pk_fd_set_int32	(pk, WPANC_BEACON_DEPTH_FIELD_INDEX,
				beacon_contents->depth,
	OPC_FIELD_SIZE_UNCHANGED);

	/* Free dynamic memory. */
	op_prg_cmo_dealloc (beacon_contents);

	if (op_prg_odb_ltrace_active ("wpan_mac") || op_prg_odb_ltrace_active
("wpan_mac_br"))
		{
		sprintf (message_str, "Sending a beacon out on channel %d.\n",
my_channel);
		op_prg_odb_print_major (message_str, OPC_NIL, OPC_NIL, OPC_NIL);
		}

	/* Beacon use the longer overhead fields, as per data packets. */
	pk_size = op_pk_total_size_get (pk) + WPAN_MAC_DATA_OVERHEAD;
	op_pk_total_size_set (pk, pk_size);

	/* Update the control traffic stats */
	wpan_mac_mgmt_sent_stats_update ((double) op_pk_total_size_get (pk));

	/* For coexistence with WLAN model. */
	op_pk_encap_flag_set (pk, OMSC_JAMMER_ENCAP_FLAG_INDEX);
	op_pk_send (pk, WIRELESS_STRM);

	FOUT;
	}
```

```
void
wpan_mac_handle_nwk_pk ()
	{
	int		pk_next_hop;
	Packet* pkptr;
	double  list_size;
	int*	channel_ptr;

	/** Handle packets coming from the network layer once the node has successfully
completed scanning. **/

	FIN (wpan_mac_handle_nwk_pk ());

	pkptr = op_pk_get (op_intrpt_strm ());
	op_pk_fd_get_int32 (pkptr, next_hop_address_index, &pk_next_hop);

	/* Update the pkt size, store relevant stathandles and stamp the pkt with current
time */
	wpan_prepare_mac_pkt_for_tx (pkptr);

	if (joined == OPC_FALSE)
		{
		channel_ptr = (int*) op_ev_state (op_ev_current ());
		if (channel_ptr != OPC_NIL)
			wpan_channel_info_set (*channel_ptr, OPC_TRUE);
		else
			op_sim_end ("Unable to retrieve the channel information from the
interrupt from Network Layer", "", "", "");
		}

	/* Invoke the CSMA-CA process if it is not busy to transmit the incoming packet
*/
	if (csma_ca_process_busy == OPC_FALSE)
		{

		/* If ACK enabled and the packet is not a braodcast packet then store a
copy of the packet in the hash table. */
		if (ack_enabled != 0 && (pk_next_hop != BROADCAST_CODE_ALL)
&& (pk_next_hop != BROADCAST_CODE_RC))
			wpan_tx_info_hash_table_update (pkptr);

		/* Invoke the process and set the busy status. */
		op_pro_invoke (csma_ca_prohandle, pkptr);
		csma_ca_process_busy = OPC_TRUE;
		}
	else
```

```
		{
		/* Buffer the packet in the queue and update the queue statistics. */
		op_prg_list_insert (outstanding_pk_lptr, pkptr, OPC_LISTPOS_TAIL);
		list_size = op_prg_list_size (outstanding_pk_lptr);
		op_stat_write (q_size_stathandle, list_size);
		}

	FOUT;
	}

void
wpan_tx_info_hash_table_update (Packet* pkptr)
	{
	WPAN_Transmission_Info* trans_ptr;

	/* This fuunction updates the hash table for any transmissions that require an
ACK back. */

	FIN (wpan_tx_info_hash_table_update (Packet* pkptr));

	sequence_number++;

	/* Set a field requesting an acknowledgement back, sequence number and this
node's network address. */
	op_pk_fd_set (pkptr, WPANC_MAC_ACK_FIELD_INDEX,
OPC_FIELD_TYPE_INTEGER, 1, 0);
	op_pk_fd_set (pkptr, WPANC_MAC_PHOP_FIELD_INDEX,
OPC_FIELD_TYPE_INTEGER, my_network_address, 0);
	op_pk_fd_set (pkptr, WPANC_MAC_SEQ_NUM_FIELD_INDEX,
OPC_FIELD_TYPE_INTEGER, sequence_number, 0);

	/* Initiate the transmission information structure. */
	trans_ptr = (WPAN_Transmission_Info*) op_prg_cmo_alloc
(wpan_cmo_handle,sizeof (WPAN_Transmission_Info));
	trans_ptr->pkptr = op_pk_copy (pkptr);
	trans_ptr->retrans_count = 0;
	trans_ptr->key_ptr = op_prg_cmo_alloc (wpan_cmo_handle, sizeof (int));

	*(trans_ptr->key_ptr) = sequence_number;
	prg_bin_hash_table_item_insert (ack_outstanding_htable_ptr, trans_ptr->key_ptr,
trans_ptr, PRGC_NIL);

	FOUT;
	}

void
```

```
wpan_prepare_mac_pkt_for_tx (Packet* pkptr)
	{
	WPAN_MAC_Stathandles_Info* stat_info_ptr;
	OpT_Packet_Size		pk_size;

	/** This function updates the pkt size, store relevant stathandles and stamp the pkt
with current time. **/

	FIN (wpan_prepare_mac_pkt_for_tx ());

	/* Update the load statistics before updating its size. */
	pk_size = op_pk_total_size_get (pkptr);
	wpan_mac_load_stats_update ((double) pk_size);

	pk_size = pk_size + WPAN_MAC_DATA_OVERHEAD;
	op_pk_total_size_set (pkptr, pk_size);

	/* Store the stathandles that the next hop of this packet will update. */
	stat_info_ptr = (WPAN_MAC_Stathandles_Info*) op_prg_cmo_alloc
(wpan_cmo_handle, sizeof (WPAN_MAC_Stathandles_Info));
	stat_info_ptr->senders_e2e_stathandle = &e2e_stathandle;
	if (op_pk_fd_set_ptr(pkptr, WPANC_MAC_STATHANDLES_FIELD_INDEX,
stat_info_ptr, 0, op_prg_mem_copy_create, op_prg_cmo_dealloc, sizeof
(WPAN_MAC_Stathandles_Info)) == OPC_COMPCODE_FAILURE)
		op_sim_end ("Unable to set the stathandle structure field.\n", "", "", "");

	/* Stamp the packet for calculating E2E delay at the destination hop. */
	op_pk_stamp (pkptr);

	FOUT;
	}


void
wpan_handle_pkt_transmission_success ()
	{
	int	key;
	double tx_duration;
	double pk_size;
	double list_size;
	double qdelay = 0.0;
	char	message_str [255];
	Packet* pkptr;
	WPAN_Transmission_Info* trans_info_ptr;
```

```
	/** This function handles the successful transmission by CSMA-CA process:
	**/
	/**		- Processes the next packet in the queue.
			**/
	/**			- Starts an ACK timer if ACK is enabled.
				**/

	FIN (wpan_handle_pkt_transmission_success ());

	csma_ca_process_busy = OPC_FALSE;

	/* Interrrupt code it the sequence number which is the key for the hash table */
	key = op_intrpt_code ();

	if (op_prg_odb_ltrace_active ("wpan_mac"))
		{
		sprintf (message_str, "Recieved a notification from CSMA-CA process on
successful tx of a packet with sequence number %d.\n", key);
		op_prg_odb_print_major (message_str, OPC_NIL, OPC_NIL, OPC_NIL);
		}

	/* Start the ack timer now if ACKs are enabled for this node. */
	if (ack_enabled != 0)
		{
		trans_info_ptr = (WPAN_Transmission_Info*)
prg_bin_hash_table_item_get (ack_outstanding_htable_ptr, (void*) &key);

		/* This pointer may be null if we recieved an ACK for the original
transmission  */
		/* when the CSMA-CA process was performing a retransmission for it.
			*/
		if (trans_info_ptr)
			{
			pk_size = (double) op_pk_total_size_get (trans_info_ptr->pkptr);
			tx_duration = pk_size/drate_sv;

			if (op_pk_fd_is_set (trans_info_ptr->pkptr,
WPANC_MAC_ACK_FIELD_INDEX) == OPC_TRUE)
				trans_info_ptr->ack_timer_evhandle =
op_intrpt_schedule_self (op_sim_time () + tx_duration + ack_duration, key);
			}
		}


	/* Check to see if there are any other packets that need to be transmitted. */
	list_size = op_prg_list_size (outstanding_pk_lptr);
```

```
	if (list_size > 0.0)
		{
		pkptr = op_prg_list_remove (outstanding_pk_lptr,
OPC_LISTPOS_HEAD);

		/* Update Queue Size and Queuing Delay statistics */
		op_stat_write (q_size_stathandle, list_size-1);
		qdelay =  op_sim_time () - op_pk_stamp_time_get (pkptr);
		op_stat_write (qdelay_stathandle, qdelay);

		/* Is this not a retransmission update the hash table, otherwise	*/
		/* table already has a entry in it from the original transmission.  */
		if (ack_enabled != 0)
			{
			if (op_pk_fd_is_set (pkptr,
WPANC_MAC_ACK_FIELD_INDEX) == OPC_FALSE)
				{
				wpan_tx_info_hash_table_update (pkptr);
				modmem_ptr->is_retransmission = OPC_FALSE;
				}
			else
				{
				modmem_ptr->is_retransmission = OPC_TRUE;
				}
			}

		/* Invoke the process and set the busy status. */
		op_pro_invoke (csma_ca_prohandle, pkptr);
		csma_ca_process_busy = OPC_TRUE;
		}

	FOUT;
	}

void
wpan_handle_pkt_transmission_failure ()
	{
	/* This function performs another transmission attempt to send the last failed
packet. */
	FIN (wpan_handle_pkt_transmission_failure ());

	/* Invoke the process again and set the busy status. */
	op_pro_invoke (csma_ca_prohandle, modmem_ptr->retry_csma_trans_pkptr);
	csma_ca_process_busy = OPC_TRUE;

	FOUT;
```

```
	}


void
wpan_mac_send_ack (Packet* pkptr)
	{
	Packet* ack_pkptr = OPC_NIL;
	int	pk_pan_id = 0;
	int	pk_next_hop = 0;
	int	pk_seq_number = 0;
	char	message_str1 [255];
	char	message_str2 [255];
	char	message_str3 [255];
	OpT_Packet_Size pk_size;

	/** This function sends an ACK back to the sender of the original transmission
(argument). **/

	FIN (wpan_mac_send_ack (Packet* pkptr));

	ack_pkptr = op_pk_create_fmt ("zigbee_ack");

	/* Using information on the original pkptr set the PAN ID and the network
address on the ACK. */
	op_pk_fd_get_int32 (pkptr, pan_id_index, &pk_pan_id);
	op_pk_fd_get (pkptr, WPANC_MAC_PHOP_FIELD_INDEX, &pk_next_hop);
	op_pk_fd_get (pkptr, WPANC_MAC_SEQ_NUM_FIELD_INDEX,
&pk_seq_number);

	op_pk_fd_set (ack_pkptr, WPANC_ACK_PAN_ID_FIELD_INDEX,
OPC_FIELD_TYPE_INTEGER, pk_pan_id, 0);
	op_pk_fd_set (ack_pkptr, WPANC_ACK_NEXT_HOP_ADDR_FIELD_INDEX,
OPC_FIELD_TYPE_INTEGER, pk_next_hop, 0);
	op_pk_fd_set (ack_pkptr, WPANC_ACK_SEQ_NUM_FIELD_INDEX,
OPC_FIELD_TYPE_INTEGER, pk_seq_number, 0);

	pk_size = op_pk_total_size_get (ack_pkptr) +
WPAN_MAC_ACK_OVERHEAD;
	op_pk_total_size_set (ack_pkptr, pk_size);

	/* For coexistence with WLAN model. */
	op_pk_encap_flag_set (ack_pkptr, OMSC_JAMMER_ENCAP_FLAG_INDEX);
	op_pk_send (ack_pkptr, WIRELESS_STRM);

	wpan_mac_control_sent_stats_update ((double) pk_size);
```

```
	if (op_prg_odb_ltrace_active ("wpan_mac") || op_prg_odb_ltrace_active
("wpan_mac_ack_retrans"))
		{
		sprintf (message_str1, "Sending an ACK from MAC module (ID %d) with
network address %d for data with sequence number %d.\n", self_id,
my_network_address, pk_seq_number);
		sprintf (message_str2, "\tACK Sent from: %d\n", my_network_address);
		sprintf (message_str3, "\tACK Sent to: %d\n", pk_next_hop);
		op_prg_odb_print_major (message_str1, message_str2, message_str3,
OPC_NIL);
		}

	FOUT;
	}

void
wpan_mac_fields_strip (Packet* pkptr)
	{

	/* This function strips the packet of the MAC specific fields before sending it to
the network layer.	*/
	FIN (wpan_mac_fields_strip (Packet pkptr));

	if (op_pk_fd_is_set (pkptr, WPANC_MAC_ACK_FIELD_INDEX) ==
OPC_TRUE)
		op_pk_fd_strip (pkptr, WPANC_MAC_ACK_FIELD_INDEX);

	if (op_pk_fd_is_set (pkptr, WPANC_MAC_PHOP_FIELD_INDEX) ==
OPC_TRUE)
		op_pk_fd_strip (pkptr, WPANC_MAC_PHOP_FIELD_INDEX);

	if (op_pk_fd_is_set (pkptr, WPANC_MAC_SEQ_NUM_FIELD_INDEX) ==
OPC_TRUE)
		op_pk_fd_strip (pkptr, WPANC_MAC_SEQ_NUM_FIELD_INDEX);

	if (op_pk_fd_is_set (pkptr, WPANC_MAC_STATHANDLES_FIELD_INDEX)
== OPC_TRUE)
		op_pk_fd_strip (pkptr,
WPANC_MAC_STATHANDLES_FIELD_INDEX);

	FOUT;
	}


void
wpan_mac_retransmit ()
```

```
	{
	char	message_str [255];
	int*	pk_next_hop;
	Packet* retrans_pkptr;
	int	key;
	WPAN_Transmission_Info* trans_info_ptr;
	double list_size;

	/* This function retransmits the unacknowledged packet. */
	FIN (wpan_mac_retransmit ());

	/* Get the interrupt code and associated transmission info from the hash table. */
	key = op_intrpt_code ();
	trans_info_ptr = (WPAN_Transmission_Info*) prg_bin_hash_table_item_get
(ack_outstanding_htable_ptr, (void*) &key);

	if (trans_info_ptr->retrans_count >= max_retrans)
		{
		/* Inform the network layer about the failure to trasmit the packet
					*/
		/* For the data packet and the join additional information is sent to the
network layer.		*/
		if (wpan_network_pk_is_app_data (trans_info_ptr->pkptr))
			{
			pk_next_hop = (int*) op_prg_cmo_alloc
(wpan_cmo_handle,sizeof (int));
			op_pk_fd_get_int32 (trans_info_ptr->pkptr,
next_hop_address_index, pk_next_hop);
			op_ev_state_install (pk_next_hop, OPC_NIL);
			op_intrpt_schedule_remote (op_sim_time (),
WPANC_NWK_FAILURE_CODE, nwk_id);
			op_ev_state_install (OPC_NIL, OPC_NIL);
			}
		else if (wpan_network_pk_is_join_response (trans_info_ptr->pkptr))
			{
			pk_next_hop = (int*) op_prg_cmo_alloc
(wpan_cmo_handle,sizeof (int));
			op_pk_fd_get_int32 (trans_info_ptr->pkptr,
next_hop_address_index, pk_next_hop);
			op_ev_state_install (pk_next_hop, OPC_NIL);
			op_intrpt_schedule_remote (op_sim_time (),
WPANC_NWK_JOIN_RESP_FAILURE_CODE, nwk_id);
			op_ev_state_install (OPC_NIL, OPC_NIL);
			}

		/* Update the MAC dropped stats.	*/
```

```
		op_stat_write (gbl_data_dropped_stathandle, op_pk_total_size_get
(trans_info_ptr->pkptr));
		op_stat_write (gbl_data_dropped_stathandle, 0.0);
		op_stat_write (data_dropped_stathandle, op_pk_total_size_get
(trans_info_ptr->pkptr));
		op_stat_write (data_dropped_stathandle, 0.0);

		/* Remove the entry from the hash table. */
		trans_info_ptr = (WPAN_Transmission_Info*)
prg_bin_hash_table_item_remove (ack_outstanding_htable_ptr, (void*) &key);
		wpan_trans_info_free (trans_info_ptr);

		if (op_prg_odb_ltrace_active ("wpan_mac") || op_prg_odb_ltrace_active
("wpan_mac_ack_retrans"))
			{
			sprintf (message_str, "Sending network failure notification to the
network layer from MAC module %d.\n", self_id);
			op_prg_odb_print_major (message_str, OPC_NIL, OPC_NIL,
OPC_NIL);
			}
		}
	else
		{
		trans_info_ptr->retrans_count++;

		op_stat_write (gbl_num_retrans_stathandle, 1.0);
		op_stat_write (gbl_num_retrans_stathandle, 0.0);
		op_stat_write (num_retrans_stathandle, 1.0);
		op_stat_write (num_retrans_stathandle, 0.0);

		retrans_pkptr = op_pk_copy (trans_info_ptr->pkptr);

		/* Add this packet to the top of the outstanding packets if the CSMA
process is busy.	*/
		/* Otherwise invoke the process to transmit it.
								*/
		if (csma_ca_process_busy == OPC_TRUE)
			{
			op_prg_list_insert (outstanding_pk_lptr, retrans_pkptr,
OPC_LISTPOS_HEAD);

			list_size = op_prg_list_size (outstanding_pk_lptr);
			op_stat_write (q_size_stathandle, list_size);

			if (op_prg_odb_ltrace_active ("wpan_mac") ||
op_prg_odb_ltrace_active ("wpan_mac_ack_retrans"))
```

```
                                {
                                sprintf (message_str, "Scheduling retransmission (number
%d) for packet with sequence number %d.\n", trans_info_ptr->retrans_count, key);
                                op_prg_odb_print_major (message_str, OPC_NIL,
OPC_NIL, OPC_NIL);
                                }
                        }
                else
                        {
                        if (op_prg_odb_ltrace_active ("wpan_mac") ||
op_prg_odb_ltrace_active ("wpan_mac_ack_retrans"))
                                {
                                sprintf (message_str, "Sending retransmission (number %d)
for packet with sequence number %d.\n", trans_info_ptr->retrans_count, key);
                                op_prg_odb_print_major (message_str, OPC_NIL,
OPC_NIL, OPC_NIL);
                                }

                        /* Invoke the process and set the busy status. */
                        modmem_ptr->is_retransmission = OPC_TRUE;
                        csma_ca_process_busy = OPC_TRUE;
                        op_pro_invoke (csma_ca_prohandle, retrans_pkptr);
                        }

                }

        FOUT;
        }

void
wpan_mac_set_vars ()
        {
        Address_Info *addr_info_ptr;
        char message_str [255];
        int     index;

        /** Set address, channel, etc. based on command from NWK. **/
        FIN (wpan_mac_set_vars ());

        addr_info_ptr = (Address_Info *) op_intrpt_state_ptr_get ();

        my_pan_id                       = addr_info_ptr->pan_id;
        my_network_address  = addr_info_ptr->network_address;
        my_parent_address     = addr_info_ptr->parent_address;
        my_channel                      = addr_info_ptr->channel;
        my_beacon_order     = addr_info_ptr->beacon_order;
```

```
	my_superframe_order = addr_info_ptr->superframe_order;
	op_prg_cmo_dealloc (addr_info_ptr);

	if (op_prg_odb_ltrace_active ("wpan_mac") || op_prg_odb_ltrace_active
("wpan_mac_join"))
		{
		sprintf (message_str, "Joining the network with ID %d on channel %d.\n",
my_pan_id, my_channel);
		op_prg_odb_print_minor (message_str, OPC_NIL, OPC_NIL,
OPC_NIL);
		}

	/* Update the PAN ID to index mapping array */
	index = wpan_mac_stathandle_index_get (my_pan_id);

	if (index == -1)
		{
		pan_stat_handle_array [stat_dim_index] = my_pan_id;
		stat_dim_index++;
		}

	/* Set the channel attributes for the transmitter and the receiver. */
	wpan_channel_info_set (my_channel);
	joined = OPC_TRUE;

	FOUT;
	}


void
wpan_mac_unset_vars ()
	{
	char message_str [255];
	int  num_pkts_outstanding = 0;
	Packet* pkptr;
	int  index = 0;

	/** Unset address, channel, etc. based on command from NWK. **/
	FIN (wpan_mac_unset_vars ());

	if (op_prg_odb_ltrace_active ("wpan_mac") || op_prg_odb_ltrace_active
("wpan_mac_join"))
		{
		sprintf (message_str, "Leaving the network with ID %d on channel
%d.\n", my_pan_id, my_channel);
```

```
		op_prg_odb_print_minor (message_str, OPC_NIL, OPC_NIL,
OPC_NIL);
		}

	/* Destroy any outstanding packets in the buffer. */
	num_pkts_outstanding = op_prg_list_size (outstanding_pk_lptr);
	while (index < num_pkts_outstanding)
		{
		pkptr = op_prg_list_remove (outstanding_pk_lptr,
OPC_LISTPOS_TAIL);

		/* Update the MAC dropped stats.	*/
		op_stat_write (gbl_data_dropped_stathandle, op_pk_total_size_get
(pkptr));
		op_stat_write (gbl_data_dropped_stathandle, 0.0);
		op_stat_write (data_dropped_stathandle, op_pk_total_size_get (pkptr));
		op_stat_write (data_dropped_stathandle, 0.0);

		op_pk_destroy (pkptr);
		index ++;
		}

	op_stat_write (q_size_stathandle, 0.0);

	/* Clear the hash table of the transmissions that are awaiting ACKs. */
	if (ack_outstanding_htable_ptr)
		{
		prg_bin_hash_table_destroy (ack_outstanding_htable_ptr,
wpan_trans_info_free);
		ack_outstanding_htable_ptr = OPC_NIL;
		}

	my_pan_id			= -1;
	my_network_address	= -1;
	my_parent_address	= -1;
	my_channel			= -1;
	my_beacon_order		= -1;
	my_superframe_order = -1;

	/* Invoke the CSMA-CA process to abort any ongoing transmission. */
	op_pro_invoke (csma_ca_prohandle, OPC_NIL);
	csma_ca_process_busy = OPC_FALSE;

	joined = OPC_FALSE;

	FOUT;
```

```
	}


void
wpan_start_scan ()
	{
	int *temp_net_addr;
	char message_str [255];

	/* This function handles the start scan command from the network layer. */
	FIN (wpan_start_scan ());

	temp_net_addr = (int *) op_intrpt_state_ptr_get ();
	my_network_address = *temp_net_addr;

	/* Reset the channel_index and beacon list from any prior scan. */
	channel_index = 0;

	if (beacon_scan_list != OPC_NIL)
		op_prg_list_free (beacon_scan_list);
	else
		beacon_scan_list = op_prg_list_create ();

	if (op_prg_odb_ltrace_active ("wpan_mac") || op_prg_odb_ltrace_active
("wpan_mac_join"))
		{
		sprintf (message_str, "Starting the scan.\n");
		op_prg_odb_print_major (message_str, OPC_NIL, OPC_NIL, OPC_NIL);
		}

	wpan_perform_scan ();

	FOUT;
	}


void
wpan_perform_scan ()
	{

	char message_str [255];
	double scan_duration;

	/** This function updates the channel information based on the channel that needs
to be scanned. **/
```

```
	/** It also calls a function to send a beacon request out and informs the higher
layer when all **/
	/** the active channels are scanned.
                                                                **/

	FIN (wpan_perform_scan ());

	while ((channel_index == 0 && !band_id_1) ||
		  (channel_index > 0 && channel_index <= 10 && !band_id_2) ||
		  (channel_index > 10 && channel_index <= 26 && !band_id_3))
		{
		channel_index++;
		}

	/* All channels have been scanned send the list of recorded beacons to network
layer.  */
	if (channel_index > 26)
		{
		op_ev_state_install (beacon_scan_list, OPC_NIL);
		if (op_prg_odb_ltrace_active ("wpan_mac") || op_prg_odb_ltrace_active
("wpan_mac_join"))
			{
			sprintf (message_str, "Sending the beacon list to the higher layer
with %d beacons.\n", op_prg_list_size (beacon_scan_list));
			op_prg_odb_print_major (message_str, OPC_NIL, OPC_NIL,
OPC_NIL);
			}
		op_intrpt_schedule_remote (op_sim_time (), 2, nwk_id);
		op_ev_state_install (OPC_NIL, OPC_NIL);

		/* Move out of scanning state using the beacon_scan_list size.	*/
		op_intrpt_schedule_self (op_sim_time (),
WPANC_SCAN_DONE_CODE);

		FOUT;
		}

	if (op_prg_odb_ltrace_active ("wpan_mac"))
		{
		sprintf (message_str, "Starting the scan for channel number %d in the
active band.\n", channel_index);
		op_prg_odb_print_major (message_str, OPC_NIL, OPC_NIL, OPC_NIL);
		}

	/* Set the channel info on the physical layer based on the channel that needs to be
scanned.	*/
```

```
	wpan_channel_info_set (channel_index);

	/* Send the beacon request out and wait for the channel sensing duration to record
beacons.		*/
	wpan_mac_beacon_request_send ();

	/* Increment the channel index for the next channel to be scanned	*/
	/* and schedule a self interrupt for the same.
	*/
	channel_index++;

	if (channel_sensing_duration == -1)
		{
		scan_duration = .05
				/* maximum wait for beacon tx	*/
					+ ((0.0 +  WPAN_MAC_ACK_OVERHEAD)  /
drate_sv)	/* tx delay for beacon request */
					+	((88.0 +
WPAN_MAC_DATA_OVERHEAD) / drate_sv)	/* tx delay for beacon		*/
					+ (2 * 1000.0 / PROP_VELOCITY);
				/* prop delay for both	*/
		}
	else
		{
		scan_duration = channel_sensing_duration;
		}

	op_intrpt_schedule_self (op_sim_time () + scan_duration,
WPANC_PROGRESS_SCAN_CODE);

	FOUT;
	}


void
wpan_mac_beacon_request_send ()
	{
	Packet* pk;
	char message_str [255];
	int pk_size;

	/** This function just creates a beacon request and sends it out. **/

	FIN (wpan_mac_beacon_request_send ());

	pk = op_pk_create_fmt ("zigbee_beacon_request");
```

```
	op_pk_fd_set_int32 (pk,
WPANC_BEACON_REQUEST_BAND_868_FIELD_INDEX,  band_id_1,
	OPC_FIELD_SIZE_UNCHANGED);
	op_pk_fd_set_int32 (pk,
WPANC_BEACON_REQUEST_BAND_915_FIELD_INDEX,  band_id_2,
	OPC_FIELD_SIZE_UNCHANGED);
	op_pk_fd_set_int32 (pk,
WPANC_BEACON_REQUEST_BAND_2450_FIELD_INDEX, band_id_3,
	OPC_FIELD_SIZE_UNCHANGED);

	if (op_prg_odb_ltrace_active ("wpan_mac") || op_prg_odb_ltrace_active
("wpan_mac_join") || op_prg_odb_ltrace_active ("wpan_mac_br"))
		{
		sprintf (message_str, "Sending a beacon request message: %d.\n",
channel_index);
		op_prg_odb_print_major (message_str, OPC_NIL, OPC_NIL, OPC_NIL);
		}

	/* Beacon requests use the shorter overhead fields, as per ACKs. */
	pk_size = op_pk_total_size_get (pk) + WPAN_MAC_ACK_OVERHEAD;
	op_pk_total_size_set (pk, pk_size);

	wpan_mac_mgmt_sent_stats_update ((double) op_pk_total_size_get (pk));

	/* For coexistence with WLAN model. */
	op_pk_encap_flag_set (pk, OMSC_JAMMER_ENCAP_FLAG_INDEX);
	op_pk_send (pk, WIRELESS_STRM);

	FOUT;
	}

void
wpan_mac_initialize_channels (Objid my_node_id, int data_rate)
	{
	Objid	trans_bands_id = OPC_OBJID_INVALID;
	Objid	trans_info_objid = OPC_OBJID_INVALID;
	Objid	channel_objid = OPC_OBJID_INVALID;
	WPAN_Channel_Info* channel_info_ptr = OPC_NIL;
	int	index = 0;
	char	message_str [255];
	double	channel_center_frequency = 0.0;

	/* Get the channel band for this device and populate the channel information
structures for the applicable channels. */
```

```
	FIN (wpan_mac_initialize_channels (my_node_id));

	/* Store the channel object ids as state variables. */
	rcvr_objid = op_id_from_name (my_node_id, OPC_OBJTYPE_RARX,
"wireless_rx");
	op_ima_obj_attr_get (rcvr_objid, "channel", &channel_objid);
	rx_channel_info_objid = op_topo_child (channel_objid,
OPC_OBJTYPE_RARXCH, 0);

	tx_objid = op_id_from_name (my_node_id, OPC_OBJTYPE_RATX,
"wireless_tx");
	op_ima_obj_attr_get (tx_objid, "channel", &channel_objid);
	tx_channel_info_objid = op_topo_child (channel_objid,
OPC_OBJTYPE_RATXCH, 0);

	/* Set RX power threshold as a reciever state */
	op_ima_obj_state_set (rcvr_objid, &(modmem_ptr->rx_power_threshold));

	/* Get channel bands that are enabled. */
	if (op_ima_obj_attr_exists (my_node_id, "Transmission Bands") == OPC_TRUE)
		{
		op_ima_obj_attr_get (my_node_id, "Transmission Bands",
&trans_bands_id);

		trans_info_objid = op_topo_child (trans_bands_id,
OPC_OBJTYPE_GENERIC, 0);
		op_ima_obj_attr_get_toggle (trans_info_objid, "2450 MHz Band",
&band_id_3);
		op_ima_obj_attr_get_toggle (trans_info_objid, "915 MHz Band",
&band_id_2);
		op_ima_obj_attr_get_toggle (trans_info_objid, "868 MHz Band",
&band_id_1);
		}

	channel_info_lptr = op_prg_list_create ();

	/* For each enabled band allocate the memory associated with its channel and
populate the channels. */
	/*band_id_1*/
	channel_info_ptr = (WPAN_Channel_Info*) op_prg_cmo_alloc
(wpan_cmo_handle, sizeof (WPAN_Channel_Info));
	channel_info_ptr->min_frequency = WPANC_EUROPE_BAND_FREQ -
WPANC_BANDWIDTH/2.0;
	if (data_rate == -1)
		channel_info_ptr->drate = WPANC_EUROPE_BAND_DRATE;
	else
```

```
		channel_info_ptr->drate = data_rate;

	channel_info_ptr->tx_band = WPANC_EUROPE_BAND;

	op_prg_list_insert (channel_info_lptr, channel_info_ptr, OPC_LISTPOS_TAIL);

	/*band_id_2*/
	for (index = 1; index < 11; index++)
		{
		channel_info_ptr = (WPAN_Channel_Info*) op_prg_cmo_alloc
(wpan_cmo_handle, sizeof (WPAN_Channel_Info));

		channel_center_frequency =
WPANC_NORTHAMERICA_BAND_FREQ + 2*(index-1);
		channel_info_ptr->min_frequency = channel_center_frequency -
WPANC_BANDWIDTH/2.0 ;

		if (data_rate == -1)
			channel_info_ptr->drate =
WPANC_NORTHAMERICA_BAND_DRATE;
		else
			channel_info_ptr->drate = data_rate;

		channel_info_ptr->tx_band = WPANC_NORTHAMERICA_BAND;

		op_prg_list_insert (channel_info_lptr, channel_info_ptr,
OPC_LISTPOS_TAIL);
		}


	/*band_id_3*/
	for (index = 11; index < 27; index++)
		{
		channel_info_ptr = (WPAN_Channel_Info*) op_prg_cmo_alloc
(wpan_cmo_handle, sizeof (WPAN_Channel_Info));

		channel_center_frequency = WPANC_WORLDWIDE_BAND_FREQ +
5*(index-11);
		channel_info_ptr->min_frequency = channel_center_frequency -
WPANC_BANDWIDTH/2.0 ;

		if (data_rate == -1)
			channel_info_ptr->drate =
WPANC_WORLDWIDE_BAND_DRATE;
		else
			channel_info_ptr->drate = data_rate;
```

```
		channel_info_ptr->tx_band = WPANC_WORLDWIDE_BAND;

		op_prg_list_insert (channel_info_lptr, channel_info_ptr,
OPC_LISTPOS_TAIL);
		}

	if (op_prg_odb_ltrace_active ("wpan_mac"))
		{
		sprintf (message_str, "Number of channels on which if the information is
stored %d.\n", op_prg_list_size (channel_info_lptr));
		op_prg_odb_print_minor (message_str, OPC_NIL, OPC_NIL,
OPC_NIL);
		}


	FOUT;
	}

void
wpan_channel_info_set (int index)
	{

	/* Set the channel information on both transmitter and reciever. */

	WPAN_Channel_Info* channel_info_ptr = OPC_NIL;

	FIN (wpan_channel_info_set (int index, Boolean index_absolute));

	/* If none of the bands are enabled then this function is a no-op */
	if (!(band_id_1 || band_id_2 || band_id_3))
		FOUT;

	/* Set the channel attributes using the channel information initialized. */
	channel_info_ptr = (WPAN_Channel_Info*) op_prg_list_access
(channel_info_lptr, index);

	op_ima_obj_attr_set (rx_channel_info_objid, "data rate", channel_info_ptr-
>drate);
	op_ima_obj_attr_set (rx_channel_info_objid, "min frequency", channel_info_ptr-
>min_frequency);

	op_ima_obj_attr_set (tx_channel_info_objid, "data rate", channel_info_ptr-
>drate);
	op_ima_obj_attr_set (tx_channel_info_objid, "min frequency", channel_info_ptr-
>min_frequency);
```

```
	/* Also update the module wide memory with appropriate symbol rate. */
	/* Here is how symbol rate is determined:
		*/
	/*	- For Worldwide band it is 62500 (per spec)
		*/
	/*	- For North American band it is 40000 (per spec)
	*/
	/*  - For Eureopean band it is 20000 (per spec)
		*/
	/*	- For all custom data rates it is 0.25*data_rate
	*/
	if (channel_info_ptr->tx_band == WPANC_NORTHAMERICA_BAND ||
channel_info_ptr->tx_band == WPANC_EUROPE_BAND)
		modmem_ptr->symbol_rate = channel_info_ptr->drate;
	else
		modmem_ptr->symbol_rate = 0.25* (channel_info_ptr->drate);

	drate_sv = channel_info_ptr->drate;

	/* Set the modulation scheme appropriately  */
	if (channel_info_ptr->tx_band == WPANC_WORLDWIDE_BAND)
		{
		op_ima_obj_attr_set (tx_objid, "modulation", "qpsk");
		op_ima_obj_attr_set (rcvr_objid, "modulation", "qpsk");
		}
	else
		{
		/* Set modulation back to bpsk when we're not in 2.4 GHz */
		op_ima_obj_attr_set (tx_objid, "modulation", "bpsk");
		op_ima_obj_attr_set (rcvr_objid, "modulation", "bpsk");
		}

	/* Also update the state variable that tracks the current channel number this node
is on. */
	channel_number = index;

	FOUT;
	}


void
wpan_trans_info_free (void* htable_entry_ptr)
	{
	WPAN_Transmission_Info* trans_info_ptr;
```

```
	/* Frees up the memory associated with a given hash table entry. */
	FIN (wpan_trans_info_free ());

	trans_info_ptr = (WPAN_Transmission_Info*) htable_entry_ptr;

	op_pk_destroy (trans_info_ptr->pkptr);

	op_prg_cmo_dealloc (trans_info_ptr->key_ptr);

	/* If the ACK event handle is not expired and the current event		*/
	/* is not ACK expiry event cancel the ACK event handle.
	*/
	if (op_ev_valid (trans_info_ptr->ack_timer_evhandle))
		{
		if (op_ev_equal  (op_ev_current (), trans_info_ptr->ack_timer_evhandle)
== OPC_FALSE)
			op_ev_cancel (trans_info_ptr->ack_timer_evhandle);
		}
	op_prg_cmo_dealloc (trans_info_ptr);

	FOUT;
	}

void
wpan_mac_thput_and_e2e_stats_update (Packet* pkptr)
	{
	double					pk_size = 0.0;
	double					delay = 0.0;
	WPAN_MAC_Stathandles_Info*	sender_stat_info_ptr = OPC_NIL;

	/* This function updates the throughput and MAC hop delay statistics.	*/
	/* For the local statistics it uses the stathandles that are being		*/
	/* set by the sender MAC.
					*/

	FIN (wpan_mac_thput_and_e2e_stats_update (Packet* pkptr));

	pk_size = (double) op_pk_total_size_get (pkptr);
	op_stat_write (gbl_thput_bps_stathandle, pk_size);
	op_stat_write (gbl_thput_bps_stathandle, 0.0);

	op_stat_write (thput_bps_stathandle, pk_size);
	op_stat_write (thput_bps_stathandle, 0.0);

	delay = op_sim_time () - op_pk_stamp_time_get (pkptr);
	op_stat_write (gbl_e2e_stathandle, delay);
```

```
	op_pk_fd_get_ptr (pkptr, WPANC_MAC_STATHANDLES_FIELD_INDEX,
(void *) &sender_stat_info_ptr);

	if (sender_stat_info_ptr)
		{
		op_stat_write (*(sender_stat_info_ptr->senders_e2e_stathandle), delay);
		op_prg_cmo_dealloc (sender_stat_info_ptr);
		}

	FOUT;
	}

void
wpan_mac_load_stats_update (double pk_size)
	{
	int	index;
	char annot_string [8];

	/* This function updates both the global and local load statistics. */

	FIN (wpan_mac_load_stats_update (double pk_size));

	op_stat_write (gbl_load_stathandle, pk_size);
	op_stat_write (gbl_load_stathandle, 0.0);

	/* Update the global statistics per PAN */
	if (my_pan_id != -1)
		{
		index = wpan_mac_stathandle_index_get (my_pan_id);

		gbl_load_dim_stathandle [index] = op_stat_reg ("ZigBee 802_15_4
MAC.Load per PAN (bits/sec)", index, OPC_STAT_GLOBAL);
		sprintf (annot_string, "PAN %d", my_pan_id);
		op_stat_annotate (gbl_load_dim_stathandle [index], annot_string);

		op_stat_write (gbl_load_dim_stathandle [index], pk_size);
		op_stat_write (gbl_load_dim_stathandle [index], 0.0);
		}

	op_stat_write (load_stathandle, pk_size);
	op_stat_write (load_stathandle, 0.0);

	FOUT;
	}
```

```
void
wpan_mac_rcvd_stats_update (double pk_size)
	{
	/* This function updates both the global and local data recieved statistics. */

	FIN (wpan_mac_rcvd_stats_update (double pk_size));

	op_stat_write (gbl_data_rcvd_stathandle, pk_size);
	op_stat_write (gbl_data_rcvd_stathandle, 0.0);

	op_stat_write (data_rcvd_stathandle, pk_size);
	op_stat_write (data_rcvd_stathandle, 0.0);

	FOUT;
	}

void
wpan_mac_mgmt_rcvd_stats_update (double pk_size)
	{

	/* This function updates both the global and local mgmt traffic recieved statistics.
*/

	FIN (wpan_mac_mgmt_rcvd_stats_update (double pk_size));

	op_stat_write (gbl_mgmt_data_rcvd_stathandle, pk_size);
	op_stat_write (gbl_mgmt_data_rcvd_stathandle, 0.0);

	op_stat_write (mgmt_data_rcvd_stathandle, pk_size);
	op_stat_write (mgmt_data_rcvd_stathandle, 0.0);

	FOUT;
	}


void
wpan_mac_mgmt_sent_stats_update (double pk_size)
	{

	/* This function updates both the global and local mgmt traffic sent statistics. */


	FIN (wpan_mac_mgmt_rcvd_stats_update (double pk_size));

	op_stat_write (gbl_mgmt_data_sent_stathandle, pk_size);
```

```
	op_stat_write (gbl_mgmt_data_sent_stathandle, 0.0);

	op_stat_write (mgmt_data_sent_stathandle, pk_size);
	op_stat_write (mgmt_data_sent_stathandle, 0.0);

	FOUT;
	}

void
wpan_mac_control_rcvd_stats_update (double pk_size)
	{

	/* This function updates both the global and local control traffic recieved
statistics. */

	FIN (wpan_mac_control_rcvd_stats_update (double pk_size));

	op_stat_write (gbl_control_data_rcvd_stathandle, pk_size);
	op_stat_write (gbl_control_data_rcvd_stathandle, 0.0);

	op_stat_write (control_data_rcvd_stathandle, pk_size);
	op_stat_write (control_data_rcvd_stathandle, 0.0);

	FOUT;
	}


void
wpan_mac_control_sent_stats_update (double pk_size)
	{

	/* This function updates both the global and local control traffic sent statistics. */


	FIN (wpan_mac_control_sent_stats_update (double pk_size));

	op_stat_write (gbl_control_data_sent_stathandle, pk_size);
	op_stat_write (gbl_control_data_sent_stathandle, 0.0);

	op_stat_write (control_data_sent_stathandle, pk_size);
	op_stat_write (control_data_sent_stathandle, 0.0);

	FOUT;
	}

Boolean wpan_network_pk_is_app_data (Packet *pk)
```

```
	{
	int pk_type;
	char format_name[100];

	/* Performs a check on the network PDU to determine whether it is an
Application traffic. */
	FIN (wpan_network_pk_is_app_data (void));

	/* Check if the packet is a network pdu. */
	op_pk_format (pk, format_name);
	if (strcmp (format_name, "zigbee_network_pdu") != 0)
		FRET (OPC_FALSE);

	/* Check if the packet is application data. */
	op_pk_fd_get_int32 (pk, packet_type_index, &pk_type);
	if (pk_type != 0)
		FRET (OPC_FALSE);

	FRET (OPC_TRUE);
	}


Boolean
wpan_network_pk_is_join_response (Packet *pk)
	{
	int pk_type;
	char format_name[100];

	/* Performs a check on the network PDU to determine whether it is an
Application traffic. */
	FIN (wpan_network_pk_is_join_response (void));

	/* Check if the packet is a network pdu. */
	op_pk_format (pk, format_name);
	if (strcmp (format_name, "zigbee_network_pdu") != 0)
		FRET (OPC_FALSE);

	/* Check if the packet is application data. */
	op_pk_fd_get_int32 (pk, packet_type_index, &pk_type);
	if (pk_type != 1)
		FRET (OPC_FALSE);

	FRET (OPC_TRUE);
	}

void
```

```
wpan_mac_hash_initialize ()
	{
	/* If ACK is enabled create a hash table to store the packets using key as their
sequence numbers. */

	FIN (wpan_mac_hash_initialize ());

	/* This is the old hash table that is not freed up beacuse the node never
successfully joined a PAN */
	if (ack_outstanding_htable_ptr)
		prg_bin_hash_table_destroy (ack_outstanding_htable_ptr,
wpan_trans_info_free);

	if (ack_enabled != 0)
		{
		/* This hash can store upto 256 entries, meaning this is the maximum
number		*/
		/* of packets that can be outstanding waiting for an ACK.
				*/
		ack_outstanding_htable_ptr =  prg_bin_hash_table_create (8, sizeof (int));
		}

	FOUT;
	}

int
wpan_mac_stathandle_index_get (int curr_pan_id)
	{
	int index;
	int pan_id;

	/* This function determines the index of the global stat handle for a given PAN ID
*/

	FIN (wpan_mac_stathandle_index_get (int curr_pan_id));

	for (index = 0; index < stat_dim_size; index ++)
		{
		pan_id = pan_stat_handle_array [index];

		if (pan_id == curr_pan_id)
			FRET (index);
		}

	FRET (-1);
	}
```

Header Block

```
/****************************************/
/* Copyright (c) 1987-2007 */
/* by OPNET Technologies, Inc. */
/* (A Delaware Corporation) */
/* 7255 Woodmont Av., Suite 250  */
/*     Bethesda, MD 20814, U.S.A.       */
/* All Rights Reserved. */
/****************************************/


#include <802_15_4.h>
#include <csma_ca.h>

#include <prg_bin_hash.h>
#include <math.h>
#include <oms_dist_support.h>
#include <jammers.h>

#define          PROP_VELOCITY    3.0E+08

#define WPANC_SCAN_DONE_CODE              -1
#define WPANC_PROGRESS_SCAN_CODE          -2

#define NWK_STRM                          0
#define WIRELESS_STRM                     1

#define WPAN_MAC_DATA_OVERHEAD 144
#define WPAN_MAC_ACK_OVERHEAD 88

#define SCAN_REQUEST              (op_intrpt_type() == OPC_INTRPT_REMOTE &&
op_intrpt_code() == 0)
#define BEACON_REQUEST                  (op_intrpt_type() ==
OPC_INTRPT_REMOTE && op_intrpt_code() == 1)
#define NWK_PK_RCVD               (op_intrpt_type() == OPC_INTRPT_STRM &&
op_intrpt_strm() == 0)
#define WIRELESS_PK_RCVD          (op_intrpt_type() == OPC_INTRPT_STRM &&
op_intrpt_strm() == 1)
#define PROGRESS_SCAN             (op_intrpt_type() == OPC_INTRPT_SELF &&
op_intrpt_code() == WPANC_PROGRESS_SCAN_CODE)
#define NETWORK_JOIN              (op_intrpt_type() == OPC_INTRPT_REMOTE &&
op_intrpt_code() == 2)
#define NODE_FAILURE              (op_intrpt_type() == OPC_INTRPT_FAIL)
#define NODE_RECOVERY             (op_intrpt_type() == OPC_INTRPT_RECOVER)
```

```
#define SCAN_DONE                       (op_intrpt_type() == OPC_INTRPT_SELF
&& op_intrpt_code() == WPANC_SCAN_DONE_CODE)
#define ACK_TIMER               (op_intrpt_type() == OPC_INTRPT_SELF &&
op_intrpt_code() != WPANC_PROGRESS_SCAN_CODE && op_intrpt_code () !=
WPANC_SCAN_DONE_CODE)
#define PAN_LOST                (op_intrpt_type() == OPC_INTRPT_REMOTE &&
op_intrpt_code() == WPANC_PARENT_LOST)
#define TX_DONE                 (op_intrpt_type () == OPC_INTRPT_PROCESS
&& op_intrpt_code () != WPANC_TX_FAILURE)
#define TX_FAILED               (op_intrpt_type () == OPC_INTRPT_PROCESS
&& op_intrpt_code () == WPANC_TX_FAILURE)


/* Function prototypes for initialization and network join process. */
void wpan_mac_init ();
void wpan_mac_record_beacon ();
void wpan_mac_send_results ();
void wpan_mac_send_beacon ();
void wpan_mac_set_vars ();
void wpan_mac_unset_vars ();
void wpan_mac_beacon_request_send ();
void wpan_mac_initialize_channels ();
void wpan_channel_info_set ();
void wpan_start_scan ();
void wpan_perform_scan ();


/* Function prototypes for transmission and reception of data. */
void wpan_mac_handle_wireless_pk ();
void wpan_mac_handle_nwk_pk ();
void wpan_prepare_mac_pkt_for_tx ();
void wpan_mac_send_ack (Packet* pkptr);
void wpan_mac_retransmit ();
void wpan_tx_info_hash_table_update (Packet* pkptr);
void wpan_handle_pkt_transmission_success ();
void wpan_handle_pkt_transmission_failure ();

/* Function prototypes for statistics update. */
void wpan_mac_register_stats ();
void wpan_mac_rcvd_stats_update (double pk_size);
void wpan_mac_thput_and_e2e_stats_update (Packet* pkptr);
void wpan_mac_load_stats_update (double pk_size);
void wpan_mac_mgmt_sent_stats_update (double pk_size);
void wpan_mac_mgmt_rcvd_stats_update (double pk_size);
void wpan_mac_control_sent_stats_update (double pk_size);
void wpan_mac_control_rcvd_stats_update (double pk_size);
```

```
/* Function prototypes for miscellanesous tasks. */
void wpan_mac_drop_pk ();
Boolean wpan_network_pk_is_app_data (Packet *);
Boolean wpan_network_pk_is_join_response (Packet *pk);
void wpan_mac_fields_strip (Packet* pkptr);
void wpan_trans_info_free (void* htable_entry_ptr);
void wpan_mac_hash_initialize ();
int wpan_mac_stathandle_index_get (int curr_pan_id);

/* Globals to maintain the mapping of stathandle indices to corresponding PAN IDs */
int stat_dim_index = 0;
int pan_stat_handle_array [32];
```

Initial State

```
wpan_mac_init();
```

Failure State Enter Execution

```
if (op_intrpt_type() == OPC_INTRPT_FAIL)
        {
        op_intrpt_clear_self ();
        op_strm_flush (OPC_STRM_ALL);
        }
```

Failure State Exit Execution

```
int i;

for (i = 0; !op_strm_empty (0); i++)
        op_pk_destroy (op_pk_get (0));


for (i = 0; !op_strm_empty (1); i++)
        op_pk_destroy (op_pk_get (1));
```