

# Chapter 1

## Zigbee Software Modifications

Several modifications have been done to the original source code to personalize the code to ZMD's Transceiver. The source code depends on the way, data is interfaced. If data is passed directly from the upper layers (SSCS, for example), the behavior of the code has been in accordance with the protocol. However, it is always not possible to send data directly from the upper layers. Frequent changes in the traffic parameters make this a non-idealistic way of dealing with the situation. Hence, using the NS-2 data interface to use external data generators, helps in these situations. But, this revealed several problems which are discussed in the following sections. Similarly, problems like Radio frequency selection, number of scanning channels, triggering of unwanted routing mechanism and address resolution for star topologies, improper messing up of EDThresh\_ (used for Energy Detection mechanism) and CStresh\_(Used for Carrier Sensing) are some of the other problems that are dealt in this section. The section highlights these problems in detail and highlights the changes made to the source code to eliminate or move around these problems.

### 1.1 Energy Detection Threshold

The receiver ED measurement is intended for use by a network layer as part of a channel selection algorithm. It is an estimate of the received signal power within the bandwidth of an IEEE 802.15.4 channel. During the Clear Channel Assessment procedure for the CSMA-CA algorithm, using mode-1, use the ED measurement to determine the status of the channel. The Energy Threshold value, EDThresh\_ comes into play as a reference. Thus, any ED measurement which is above the EDThresh\_ value shall indicate the channel is busy, and value less than the EDThresh\_ shall indicate a free medium. Similarly the CStresh\_ indicates the carrier sensing ability of a node. It is the least power signal of the incoming signal that the receiver can detect. However, in the source code, the EDThresh\_ is never mentioned and the CCA mechanism in turn uses CStresh\_ as the threshold to assess a clear channel, which is prone to misunderstandings and confusion. It does not provide a meaningful and understandable distinction between the CStresh\_ and EDThresh\_.

The confusion cannot be understood until we realize the significance of EDThresh\_ and CStresh\_. According to the standard, EDThresh\_ is at-most 10dB above the Receiver sensitivity. But, the carrier sense threshold, CStresh\_, is chosen based on the *Personal Operating Space* of the devices. And in some cases, the receiver may not be sensitive enough, hence is very likely the RXThresh\_ to be above the CStresh\_. Following the original code might allow an error free operation as long as the receiver sensitivity is same as the carrier sensing capability of the device and also when assumed that the EDThresh\_ is same as RXThresh\_. Its true that these conditions are difficult to implement. Hence, separating the dependence of EDThresh\_ on CStresh\_ would give a better control on the architecture.

The following modification is implemented to incorporate EDThresh\_, separating it from CStresh\_. But an ideal implementation would be to modify the 'mac' layer files to incorporate the variable individually, which would allow EDThresh\_ to be specified as a TCL command in the scenarios. However, this is a much easier method of implementation. An important fact to be remembered here is that, the EDThresh\_ is assumed to be 10dB above the RXThresh\_. Modify the variable, 'aboveRxThresh' to set it to other value.

**Function:** *Phy802\_15\_4::CCAHandler*

Add the following code at the start of the function.

```
// (aboveRxThresh)dB above RxThresh_
int aboveRxThresh = 10;
// Convert RxThresh_ to dBm
double RxindBm = 10 * log10(RXThresh_ * 1000);
// EDThresh_ in dBm
double EDThreshIndBm = RxindBm + aboveRxThresh;
// EDThresh_ in Watts.
double EDThresh_ = pow(10.0, EDThreshIndBm/10.0) * 0.001;
```

Also modify,

```
t_status = (rxTotPower[ppib.phyCurrentChannel] >= CStresh_)?p_BUSY:p_IDLE;
to,
t_status = (rxTotPower[ppib.phyCurrentChannel] >= EDThresh_)?p_BUSY:p_IDLE;
and,
t_status = ((rxTotPower[ppib.phyCurrentChannel] >=CStresh_)
&&(rxTotNum[ppib.phyCurrentChannel] > 0))?p_BUSY:p_IDLE;
to,
t_status = ((rxTotPower[ppib.phyCurrentChannel] >=EDThresh_)
&&(rxTotNum[ppib.phyCurrentChannel] > 0))?p_BUSY:p_IDLE;
```

## 1.2 TxOptions

The MAC sublayer builds an MPDU to transmit from the supplied arguments. The TxOptions parameter indicates how the MAC sublayer data service transmits the supplied MSDU.

0x00  $\longrightarrow$  *DirectTransmission*  $\longrightarrow$  *Endnodes*  
 0x02  $\longrightarrow$  *GTStransmission*  $\longrightarrow$  *Coordinator/Endnodes*  
 0x04  $\longrightarrow$  *indirecttransmission*  $\longrightarrow$  *Coordinator*

If the TxOptions parameter is equal to 0x00, indicating a direct transmission, the MAC-Sublayer transmits the MPDU simply by competing for its turn to transmit using slotted CSMA-CA in the CAP for a beacon-enabled PAN or unslotted CSMA-CA for a nonbeacon-enabled PAN.

If the TxOptions parameter specifies that a GTS transmission(0x02) is required, the MAC sublayer will determine whether it has a valid GTS. If the device is a PAN coordinator, it will determine whether it has a receive GTS with the device with the given destination address. If a valid GTS could not be found, the MAC sublayer will issue the MCPS-DATA.confirm primitive with a status of INVALID\_GTS. If a valid GTS was found, the MAC sublayer will defer, if necessary, until the GTS. At this time, the MAC sublayer transmits the MPDU without using CSMA-CA, provided that the entire transmission and acknowledgment, if requested, can be completed before the end of the GTS.

Similarly a value of 0x04, indicates an Indirect transmission of the MPDU. Generally its a transmission type that is only possible by a coordinator. The coordinator intimates the nodes which have data to receive, in the pending address fields of the beacon. A node upon reception of the node, checks the pending address fields and upon finding its address mentioned, prepares a DataRequest message and transmits it to the coordinator. The coordinator on receiving the datarequest message, acknowledges with an acknowledgement message and if the node has data to be sent, sends the data, using CSMA-CA.

Also, the TxOptions parameter can be used to specify, security, which is not implemented in the code, yet and hence is not considered any further.

The implementation of the TxOptions parameter (referred as *txoption* in the source code), allows to choose the transmission type based on the node in consideration. However, as said earlier, it is not possible to control this parameter when a data generator, like a cbr, tcp or a poisson, is interfaced to NS-2 to generate traffic automatically. The data needs to be passed from the upper layer, which in this case is the SSCS. The code is written to assume a default value of 0x00 for data transmission. So the following modification in the function, "Mac802\_15\_4::recv" has been done to choose the right txoption value based on the node type. However, support for GTS transmission type is not yet implemented.

**Function:** *Mac802\_15\_4::recv*

Replace,

```
txop |= Mac802_15_4::txOption;
```

with,

```
if (capability.FFD&&(numberDeviceLink(&deviceLink1) > 0))
    txop |= 0x04;
else
    txop |= 0x00;
```

## 1.3 Radio Frequency Selection

The source code inherently supports the 2.4Ghz of operation. Since the frequency mode of operation for our current study is 868Mhz, the following modification, has been implemented to choose the right frequency band.

Function: `SSCS802_15_4::startPANCoord` The following lines need to be commented. Presence of these code lines prevent scanning the channels ranging from 0-10, which are the supported by the 868 and 914Mhz, frequency band. The code allows scanning of channels from 0-10, only in the absence of a free channel to establish a PAN, in the range of 11-27. However, working in the current scenario, such situations are remote, because the focus of this study is only a standalone star network topology.

```
// for (i=11;i<27;i++)
// if ((T_UnscannedChannels & (1 << i)) == 0)
// break;
// if (i >= 27)
```

## 1.4 Number of Scanned Channels

A device working in the 868Mhz or the 914Mhz frequency band, the number of channels supported are 11, together for the 868 and the 914 Mhz bands, a node should scan all of these channels to either associate to a PAN or to establish one. However, since we only simulate a standalone network, without any alien devices present in their vicinity, we limit the number of scanning channels to 3. It is to be noted that this change does not fix any bug in the code, but is implemented only to save simulation time. Also to be noted that, this would give a wrong idea about the association time of the devices. This change need to be taken into consideration before working with association/orphan-scanning times.

**File:** `p802_15_4sscs.cc`

```
// Scan the first three channels. 0, 1 and 2.
UINT_32 SSCS802_15_4::ScanChannels = 0x0007
```

## 1.5 Channels Supported

The number of channels supported is indicated by the parameter, `def_phyChannelsSupported` in the file, `p802_15_4const.h`.

**File:** `p802_15_4const.h`

```
#define def_phyChannelsSupported 0x07ff
```

## 1.6 Passive Scan Support

The code does not provide support for passive mode of scanning. The following modifications in their respective indicated locations would implement it.

**Function\_1:** `SSCS802_15_4::MLME_SCAN_confirm`

Modify,

```

//Including Passive scantype
if (ScanType == 0x01)
    dispatch(status, "MLME_SCAN_confirm");

to

//Including Passive scantype
if (ScanType == 0x01 || ScanType == 0x02)
    dispatch(status, "MLME_SCAN_confirm");

Function 2: Mac802_15_4::scanHandler
Modify,

if (taskP.mlme_scan_request_ScanType == 0x01)
    taskP.taskStep(TP_mlme_scan_request)++;

to,

if ((taskP.mlme_scan_request_ScanType == 0x01) || (taskP.mlme_scan_request_ScanType == 0x02))
    taskP.taskStep(TP_mlme_scan_request)++;

```

## 1.7 Routing

Routing is a method of determining routes to nodes to which information is due for transfer. The routing mechanism is guided with routing tables, which holds the best path to the destination node. These tables might have a direct path to the destination or a path to the nearest node, which can forward the information further, thus bridging the route to the destination node. A node which has data to be transmitted to a destination node, first looks up into its routing table, to determine if a path to the destination is available. If yes, it would go ahead with the transmission to the next nearest node in the path or may transmit a Route Request message to determine a route to the node.

However, a simple STAR topology with one way communication with the coordinator doesn't require passing through the routing mechanism. Yet, we use the AODV routing in the scenarios. Since there can be no forwarding of data in an IEEE 802.15.4 STAR topology, the next hop to the destination would be the destination itself. Hence there is no need for any determination if an entry of the path is missing in the routing table. Information can be directly sent without any lookup, being assured it is sent in the right path. Given this explanation, the routing table lookup and route finding mechanism is considered, unnecessary. Hence the following changes are made to prevent calling the resolve procedure for route finding, hence always assuming the route is present in the routing table. And finally, filling the next hop address for the node same as the destination. It is to be noted that these changes does not support other topologies which need route resolution.

**Function: AODV::recv**

Replace,

```

if ( (u_int32_t)ih->daddr() != IP_BROADCAST)
    rt_resolve(p); // Check of route is present
else
    forward((aodv_rt_entry*) 0, p, NO_DELAY);

```

```

with,

if ( (u_int32_t)ih->daddr() != IP_BROADCAST)
    forward((aodv_rt_entry*) 1, p, NO_DELAY); //Always assume route is present.
else
    forward((aodv_rt_entry*) 0, p, NO_DELAY);

Function: AODV::forward
Comment the following lines.

// if(ih->ttl_ == 0) { // Drop the packet if its Time To Live is expired.
// #ifdef DEBUG
//     fprintf(stderr, "%s: calling drop()\n", __PRETTY_FUNCTION__);
// #endif // DEBUG
//     drop(p, DROP_RTR_TTL);
//     return;
// }

Replace,

assert(rt->rt_flags == RTF_UP);
rt->rt_expire = CURRENT_TIME + ACTIVE_ROUTE_TIMEOUT;
ch->next_hop_ = rt->rtnexthop;

with,

// assert(rt->rt_flags == RTF_UP);
// rt->rt_expire = CURRENT_TIME + ACTIVE_ROUTE_TIMEOUT;
ch->next_hop_ = ih->daddr();

```

## 1.8 Address Resolution

The address resolution protocol is used to resolve the MAC addresses from the IP address. If the node has a mapping of the IP address to the MAC address, it would use it, but if its ARP table does not have an entry of the ARP of a destination node, it issues the ARP request with the IP address of destination. Nodes receiving it, respond with the MAC address of the node, if they have one. And, again as in the case of routing, given the minimal requirements of a STAR topology, an ARP message seeking the MAC address of the destination is unnecessary.

The function, `arplookup()` is called each time there is a packet to be transmitted. When there is no entry for the destination, the function is originally designed to return 0. And the calling function, `arpresolve()`, is supposed to send the arp request messages. But since we want to disable any request message transmission, we make changes to give an impression that an address mapping is always present in the arp table.

Reformat the function, `ARPTable::arplookup`, as in the following:

**Function:** *ARPTable::arplookup*

```

ARPEnt* ARPTable::arplookup(nsaddr_t dst)
{

```

```

ARPEntry *a;
// If a match is found, use it.
for(a = arphead_.lh_first; a; a = a->nextarp()) {
if(a->ipaddr_ == dst)
return a;
}
// The following change applies when no arp entry is found. We simply make a
// new entry and enable it by setting the up_ flag, assign the mac address of
// the destination same as the IP Address, and finally return the object.
a = new ARPEntry(&arphead_, dst);
a->up_ = 1;
a->macaddr_ = dst;
return a;
}

```

## 1.9 Acknowledgement Wait Duration

The MAC sublayers immediately enables the receiver after the transmission of the MPDU and waits for an acknowledgement from the recipient for at most, *macAckWaitDuration*. The value is dependent on the frequency of operation.

- 868Mhz: 120 Symbols
- 2.4Ghz: 55 Symbols.

An important point to be mentioned is that, for 2.4Ghz operation, the ideal value for *macAckWaitDuration* is 54. However, for NS-2 simulations, taking a value, of 55 is appropriate because of an inherent problem with the scheduler. In certain simulations with 54 as the Ack wait duration, for the 2.4Ghz operation, there are extensive retransmissions. Generally retransmission are carried out when the source node cannot receive an acknowledgement within the *macAckWaitDuration* time, if acknowledgements are enabled. And extensive debugging revealed that the transmission has been successful, and the destination node responds with an acknowledgement. However, the wait duration elapses by the time the acknowledgement reaches the source node. Coincidence as it may seem, the time of reception of the acknowledgement and retransmission of the next data frame is the same. But the order of these events is determined only by the scheduler. Since the two events occur at the same time, the precedence is determined by the earliest scheduled event and hence the retransmission of the data packet event is chosen. Changing the scheduler mechanism would be inappropriate in these situations, hence a longer wait duration for the *macAckWaitDuration* has been proposed.