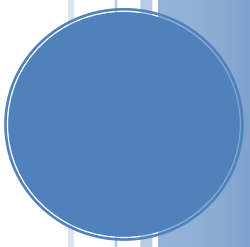
INSTITUTO
SUPERIOR
TÉCNICO

IEEE 802.15.5 Low Rate WPAN Mesh Partial Implementation

Wireless Mobile Networks – 2009/2010 Spring Semester

Duarte Dias	Nº 58068	duarte.dias@ist.utl.pt
Pedro Silva	Nº 58035	pedro.silva@ist.utl.pt
Oleksandr Yefimochkin	Nº 58958	alexander.yefimochkin@ist.utl.pt

6/30/2010



ACKNOWLEDGMENTS

We would like to thank Tiago Queiroz for being available to help us, anytime, during the months it took to develop this project. Specially, for the interface he granted us, in order to visualise the topology and for the reviewing of the report.

We would also want to thank Prof. Mario Nunes, for giving us access to the 416 room of INOV.

CONTENTS

1 INTRODUCTION

In this report we will present a partial implementation of the routing protocol laid out in the **IEEE 802.15.5 recommended practice** (1), for Jennic's 5139 modules.

Wireless sensor networks (WSN) aim to achieve reliable low-power and low-cost devices; that can be deployed in an ad-hoc fashion throughout the area/building of interest. These objectives pose design constraints during the design of these devices, which leads to challenges when dealing with the conception and implementation of routing protocols.

Jennic (2) is a fabless semiconductor company that produces several microcontrollers and modules commercially available on the market. To facilitate the development of applications for its modules, Jennic currently provides four main Stacks, where a Stack consists of a library that implements a network layer or a set of network layers, and sometimes some hardware management functionality.

The four main stacks are 802.15.4, ZigBee PRO, Jennie, and 6LoWPAN as shown in Figure 1.

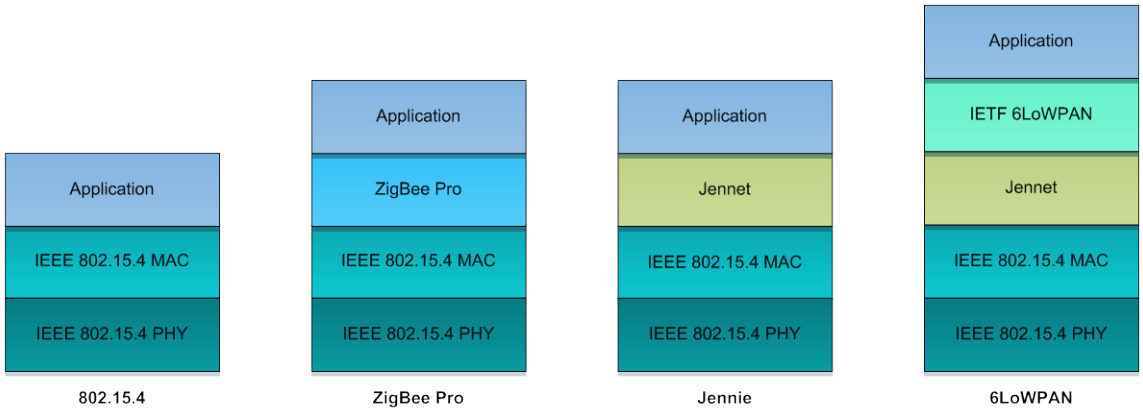


Figure 1 – Stacks provided by Jennic.

The main drawback of the Jennic's Stacks is that neither of them provides flexibility and functionality simultaneously, this arises from two facts; first the application is only able to access the interfaces provided by layer directly below it, secondly the source code associated with the stacks is not provided. The Zigbee Pro, Jennie and 6LoWPAN Stacks offer the programmer a great deal of functionality (ad-hoc network formation, routing, address assignment and reliable channels), but with no means to make any adjustments to the behaviour of the network and transport layer, however minimal (ex: give the application access to RSSI measurements). On the other hand the 802.15.4 Stack provides many degrees of freedom to the programmer but little functionality.

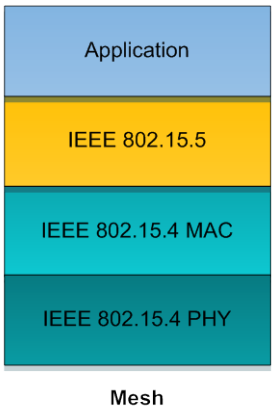


Figure 2 - Mesh Stack.

With the purpose to overcome the aforementioned problems, an implementation of the IEEE 802.15.5 LR-WPAN Mesh is presented throughout this paper. Mesh Stack was the name chosen for this implementation that is depicted in Figure 2. It is flexible because access to the code is granted and therefore its behaviour can be altered, and also resourceful, since it deals with ad-hoc network formation, link maintenance and routing. These two aspects can meet with the demands of the application layer for freedom and functionality.

In chapter 2 it is given a brief presentation of the IEEE 802.15.5. In chapter 3 several primitives are presented that allow the Application and Mesh layer to exchange messages between them and the lower layers.Various examples are presented regarding the proper away to issue such messages. In chapter 4 the various types of frames are discussed, with the respective format and meaning of the fields with greater relevance. Finally, in chapter 5 the API is presented.

2 General Description of IEEE 802.15.5

2.1 Architecture

The IEEE 802.15.5 recommended practice provides the architectural framework enabling WPAN devices to promote interoperable, stable, and scalable wireless mesh topologies (1). This recommended practice is composed of two parts: low-rate WPAN mesh and high-rate WPAN mesh networks. The low-rate mesh is built on IEEE 802.15.4 MAC, while high rate mesh utilizes IEEE 802.15.3/3b MAC. Common features of both meshes include network initialization, addressing, and multihop unicasting. In addition, low-rate mesh supports multicasting, reliable broadcasting, portability support, trace route and energy saving function, and high rate mesh supports multihop time-guaranteed service. The reference model of LR-WPAN Mesh is presented in Figure 3, where it's made clear that the Mesh layer is built upon the services provided by the MAC and PHY layers specified in the IEEE 802.15.4 standard, and can be said to belong the Network layer of the OSI model.

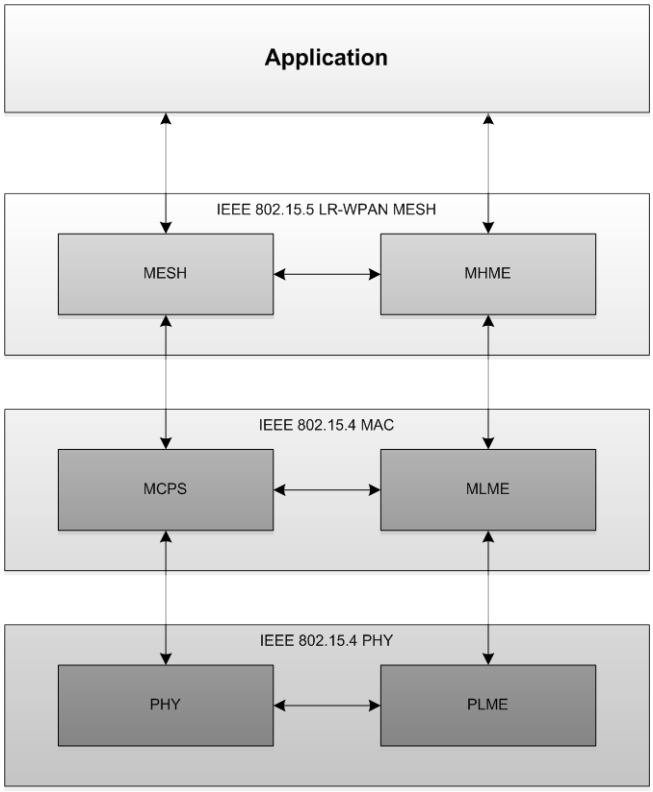


Figure 3 - The reference model of IEEE 802.15.5.

In the Mesh network topology, all devices can be identical (provided that at least one has the capability to act as the PAN Coordinator) and are deployed in an ad-hoc arrangement (with no particular network structure). Some (if not all) nodes can communicate directly. If the nodes aren't directly within range of each other, a package can be passed from one node to another until it reaches its final destination. Alternative routes may be available to some destinations,

allowing message delivery to be maintained in the case of an RF link failure (contrary to what happens when we have a tree structure). The Mesh topology is illustrated in Figure 4.

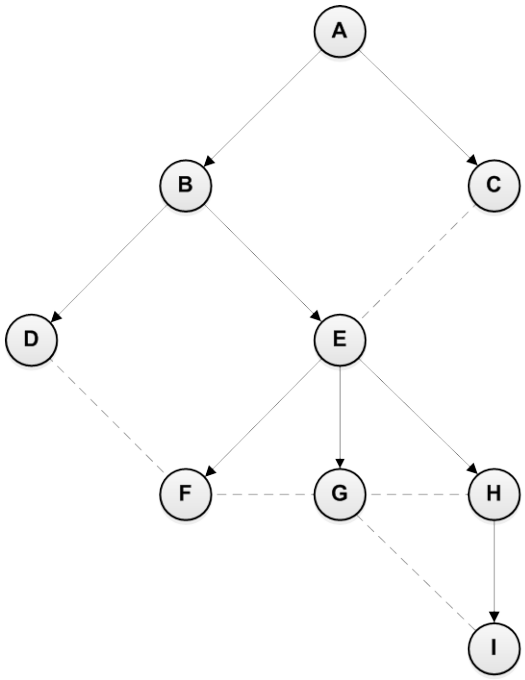


Figure 4 – Mesh topology.

The Mesh layer main advantages form the application point-of-view are:

1. Increased range of the network.
2. Enhanced reliability.

Increased range of the network

IEEE 802.15.4 offers a given node only the possibility of communicating with direct one-hop neighbours, restricting possible network topologies to star-networks (many-to-one) and peer-to-peer networks(many-to-many) where all nodes are within one-hop of the PAN coordinator. Restrictions that to be surmounted require a great deal of work and complexity at the application layer.

LR-WPAN Mesh improves matters by taking into itself the task of network formation, routing of packets, address assignment and maintenance. Providing the application with services that allow the easy establishment of a Mesh network, and the transmission of packets to nodes separated by much more than one-hop, what in practice greatly improves the range when compared to simple star-topologies provided by the IEEE 802.15.4 standard alone.

Enhanced reliability

By storing information in each node regarding the local topology of the network, LR-WPAN Mesh is capable of taking better decisions about the path taken by a packet within the network. For example, as illustrated in Figure 5, in LR-WPAN Mesh a packet whose destiny is a direct one-hop neighbour is sent directly to it, not having to route the packect through its parent.This decision increases the reliability of the network by reducing the number of hops that the packet has to travel; it should be noted that this behaviour has never been observed in any of the Jennic's stacks. Figure 6 shows the advantage of having knowledge of the local topology in order to find possible alternative routes.

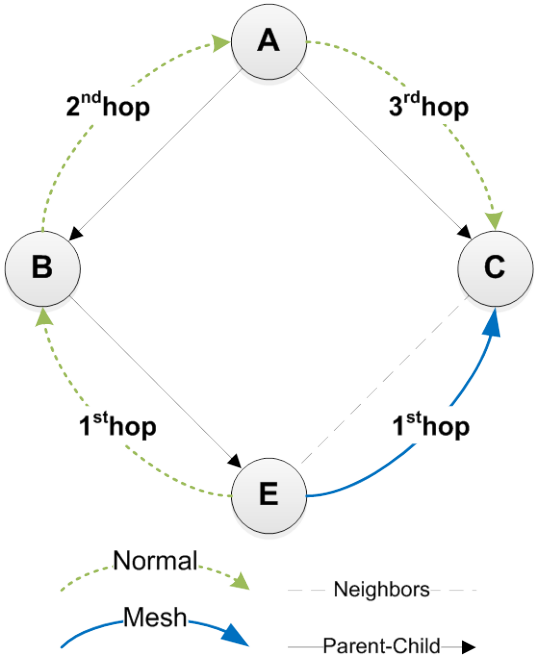


Figure 5 – Example of packet from node E to C.

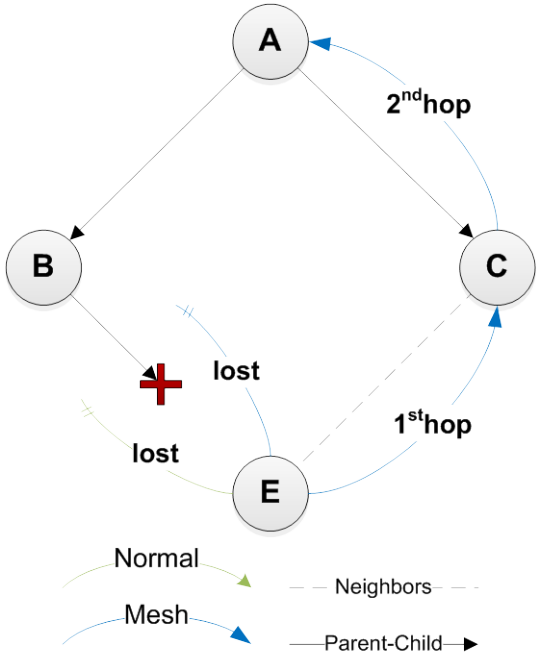


Figure 6 – Example of a packet from E to A, with a temporarily broken link.

2.2 Network Establishment

2.2.1 Starting the network

A full function device (FFD), i.e. mesh devices, that wishes to form a mesh network (i.e. become a mesh coordinator (MC)) will have the application layer above the MHME issuing a MHME-START-NETWORK.request on the sublayer, as shown in Figure 7. The MHME will deal with this request in two phases. First it will gather information on the surrounding networks issuing to the MLME a MLME-SCAN.request and storing the data present in the MLME-SCAN.confirm. Then, with the gathered information it will choose its PAN ID and start the network issuing to the MLME a MLME-START.request with PANCoordinator parameter set to TRUE. The MAC sublayer enters now in operating mode and issues a MLME-START.confirm to the MHME sublayer which reports to the higher layer with a MHME-START-NETWORK.confirm.

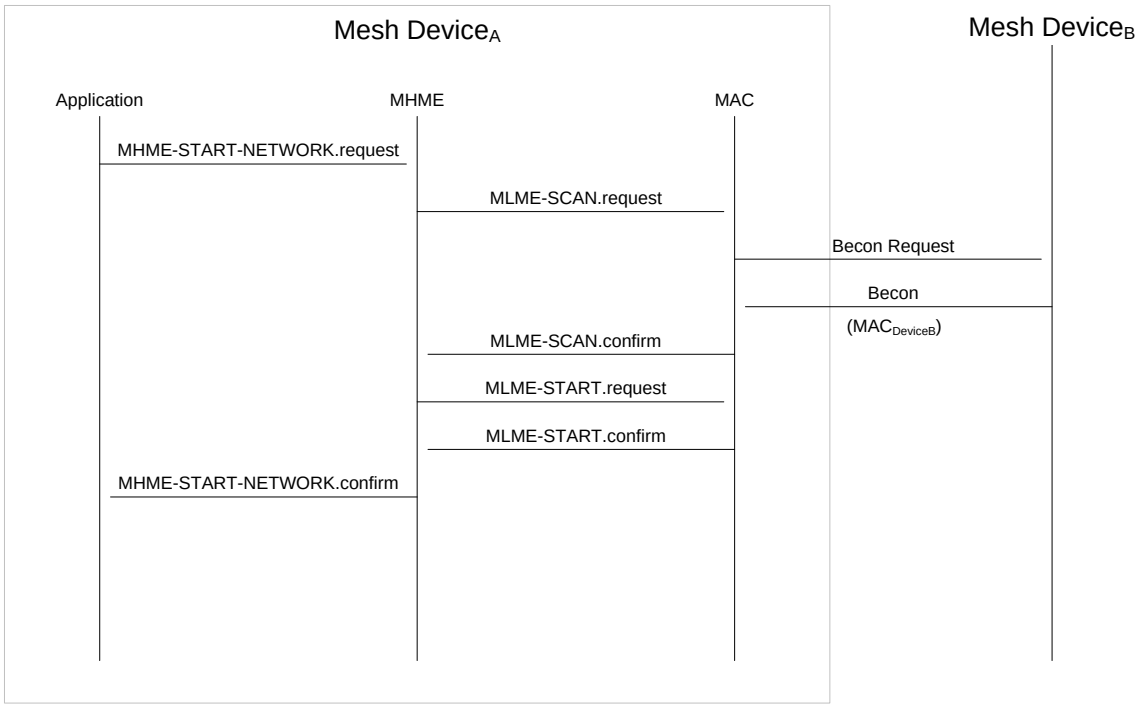


Figure 7 – Starting Network.

2.2.2 Joining the network

In order for a device to join the network, the application layer above the MHME needs to call two services from that layer: scan for other networks implemented by the MHME-DISCOVER primitive, and selecting a channel/network parent device to join implemented by the MHME-JOIN primitive, as shown in Figure 8. First the higher layer will issue a MHME-DISCOVER.request which will do the first phase of the MHME-START-NETWORK.request. The MHME will then notify the higher layer issuing a MHME-DISCOVER.confirm. The higher layer will then decide the network it the device will be joining and the modes it will be operating, and then it will issue a MHME-JOIN.request with those specification. Based on the given information the MHME sublayer will determine the best device to join and issue to the MLME a MLME-ASSOCIATE.request. The MLME sublayer will respond with a MLME-ASSOCIATE.confirm. Upon success the device needs to update its data base entries related to its parent.

If a device enters the network as a mesh device it may want to initiate the mesh functions after it has joined. This is done by issuing a MHME-START-DEVICE.request which will do the second phase of the MHME-START-NETWORK.request. The MME will then notify the higher layer issuing a MHME-START-DEVICE.confirm.

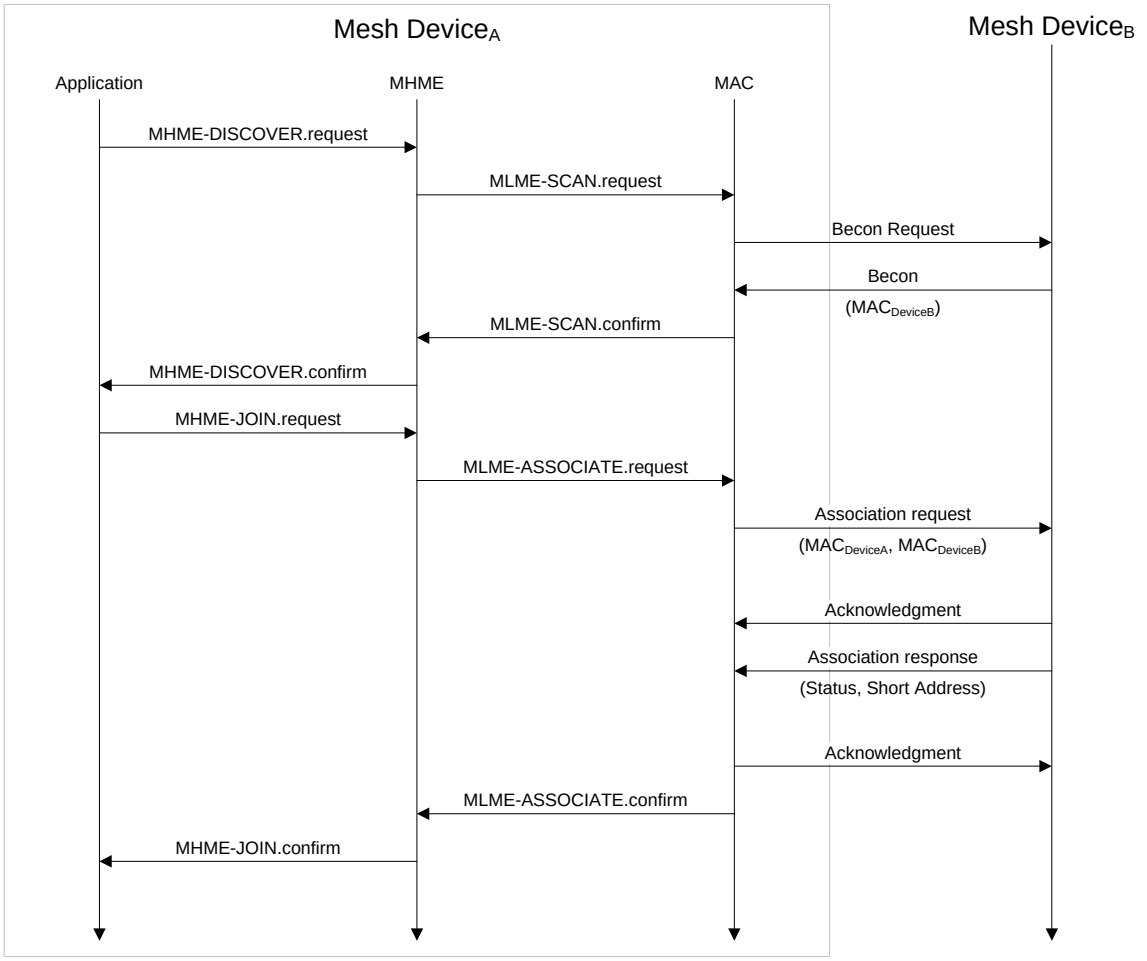


Figure 8 – Joining process.

2.3 Network Topology

2.3.1 Address Assignment

This stage is internal to the mesh layer. The process is initialized upon the arrival of the MLME-ASSOCIATE.confirm of the Join process. A timer is set to *meshChildNbReportTime* and if it expires before any other devices try to associate to the current device, the MHME will classify the device as a leaf of the network (i.e. it has no children) and send a child report command message to the parent of the current device indicating the number of children (in this case set to one) and the number of addresses needed (at least equal to the number of children reported). Devices that receive this kind of message from their children update their child number and number of requested addresses on each branch/child. After the arrival of all the children report command frames, the MHME layer of that device sends a child report command message to its parent, like in the leaf situation. The requested addresses can be higher than the number of children but never fewer.

When the network coordinator (NC) receives the child report command message from all its children it will start distributing the addresses. This is done by the MHME layer sending address assignment command messages to all the NC's children containing the start and ending addresses reserved for each device. Those devices will then send address assignment command messages to their children, and so on, until this process reaches the leafs. Note that, at each device, the addresses are divided by the number of max neighbours and each child has a block of addresses imediatly allocated. This way address managment will not force the network to go through the initialization process everytime a new child appears, unless the parent needs another block of addresses.

The exchange of this command messages will be performed by the use of the MAC data services on the MCPS sublayer. The MHME sublayer will issue an MCPS-DATA.request to the MAC sublayer and it will respond with a MCPS-DATA.confirm.

The result of this procedure can be translated into a tree, beginning with the NC. Figure 8 is an example of such a tree.

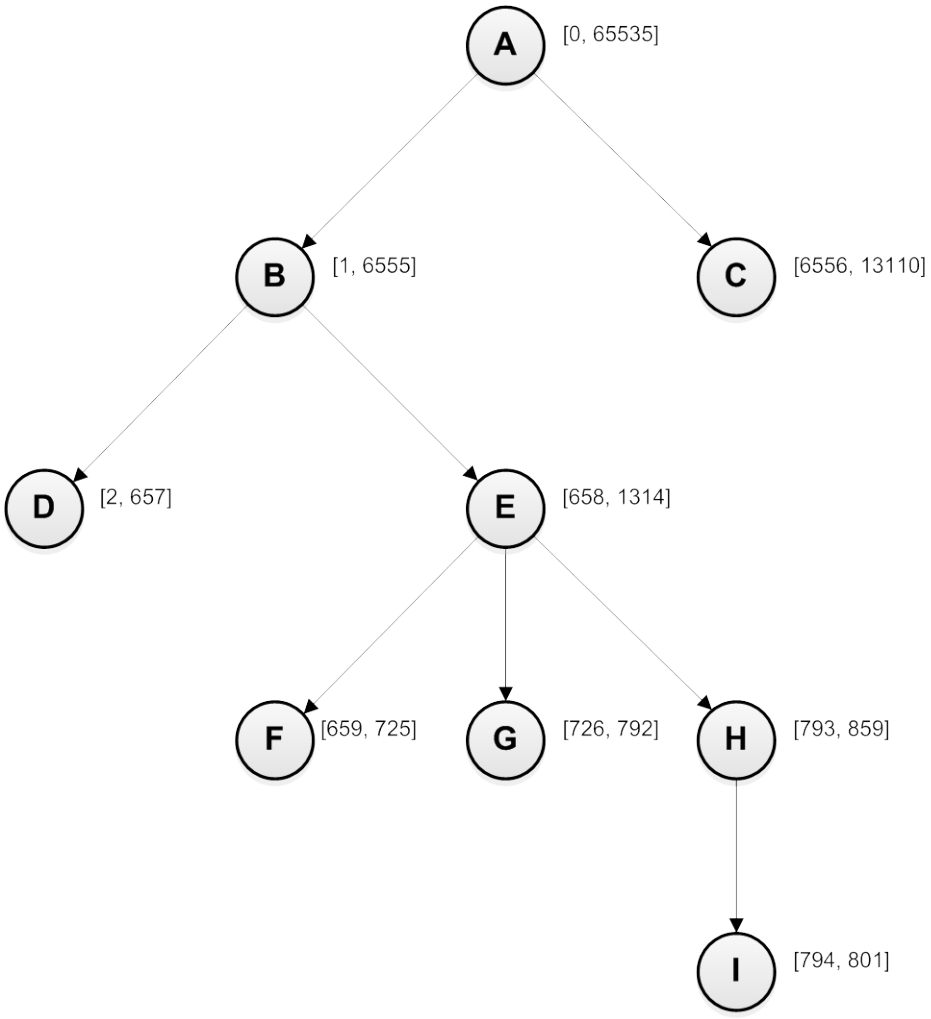


Figure 8 – Calculation of the number of devices along each branch.

2.3.2 Mesh topology discovery and formation

As soon as the device has been assigned an address block it broadcasts to its neighbours its local vision of the network. This vision is composed by itself and a list of one-hop neighbours of the device. The information is sent via a hello command frame which contains the local information of the network. This command frame is then retransmitted TTL (time to live) times, where TTL is one of its field and is set by the MHME sublayer by meshTTLOfHello. When one hello command arrives at a device, it updates its link state table (LST) with the provided information (one-hope neighbours are not be updated if the TTL value is set to one or less) and decrease the TTL by one. If the TTL is not zero the device will broadcast the same command frame to its neighbours. If the TTL is zero the command frame is ignored.

At the beginning no device knows the network topology, so the hello command frame is sent only with the device information (i.e. with no one-hope neighbours). The information referring to the one-hope neighbours that will appear in the hello command frame comes from the LST (it will search all the entries and collect all of the neighbours that have the number of hops field equal to one).

The link state table consists in a meshTTLOfHello-hop neighbour list and an auxiliary connectivity matrix (the last element exists only for aiding the calculation of the number of hops field of the neighbour list). An example of the neighbour list is given on Table 1, and a connectivity matrix on Table 2.

In the example of the connectivity matrix the 1's represent the one-hop neighbours, i.e. those where the hello command frame arrives to this device (reference as ME in the matrix) with the TTL equal meshTTLOfHello, which means that had not passed by any other device yet, and the 0's represent the devices that are not directed connected to the current device but where present in the one-hope fields of some hello command frames. For b-directional networks the matrix is symmetric so only half of the matrix has all the information contained by the matrix.

The purpose of constructing a connectivity matrix is to calculate the value of the number of hops for each entry in a neighbour list. First the field number of hops of each device is set to infinity. Then, all devices directly connected to the current device (marked as "self" in Table 2, right side) are one hop neighbours (nb2, nbn-1, … in the example). Next, all devices directly connected to one-hop neighbours (and having a hops of infinity) are two-hop neighbours (nb1, nb3, … in above example). This procedure continues until hop numbers of all neighbours are populated. These hop numbers are then filled into the number of hops field for each entry in a neighbour list for data forwarding.

Table 1 - Neighbor List.

Beginning Address	Ending Address	Tree Level	Link Quality	Relationship	Number of Hops	...
4	9	3	202	Child	1	...
10	15	3	174	Child	1	...
25	30	4	110	Sibling	2	..
...	...	...	...	...	...	...

Table 2 - Connectivity Matrix.

	Self	Nb_1	Nb_2	Nb_3	...	Nb_{n-2}	Nb_{n-1}	Nb_n
Self	-	0 or 1	0 or 1	0 or 1	...	0 or 1	0 or 1	0 or 1
Nb_1	-	-	0 or 1	0 or 1	...	0 or 1	0 or 1	0 or 1
Nb_2	-	-	-	0 or 1	...	0 or 1	0 or 1	0 or 1
Nb_3	-	-	-	-	...	0 or 1	0 or 1	0 or 1
...	-	-	-	-	-	...	...	...
Nb_{n-2}	-	-	-	-	-	-	0 or 1	0 or 1
Nb_{n-1}	-	-	-	-	-	-	-	0 or 1
Nb_n	-	-	-	-	-	-	-	-

2.3.3 Mesh path selection

When communicating with each other, devices need to know how to route the packets received. There are several algorithms that can be used to route packets, based on link state and vector distance(3). Although, devices have strict hardware limitation, especially on memory and processing power, which means that the routing algorithms should not need a lot of processing and memory to decide where to route the packets. Mobility and medium conditions can also influence the discovery and transmission of packets, something that can be challenging when finding routes (4).

When forwarding data frames the mesh sublayer follow the algorithm shown in Figure 9. First it checks the neighbour list for the destination address. If it is presented then the data frame is sent to the one hop neighbour. According to the algorithm, if at the current node (source) the destination address falls in the address block of one of his neighbours, then the data frame is forwarded to that neighbour. Though, if that does not happen, then an anchor is set on the node with the smallest tree level and number of hops, seen from the source. Now, starting from the anchor, the algorithm starts to find the next neighbour node with a number of hops equal to n-1, where n is the number of hops to the anchor. This is done recursively, until n-1 equals 1, meaning that when this condition is reached a one hop neighbour of the source is found, allowing the device to route the data frame to its destination. In case of a tie, the algorithm can select a random node. Although, to avoid confusion on the network, as data frame out of order, a certain device should stick for a while with the selected neighbour (4) (1).

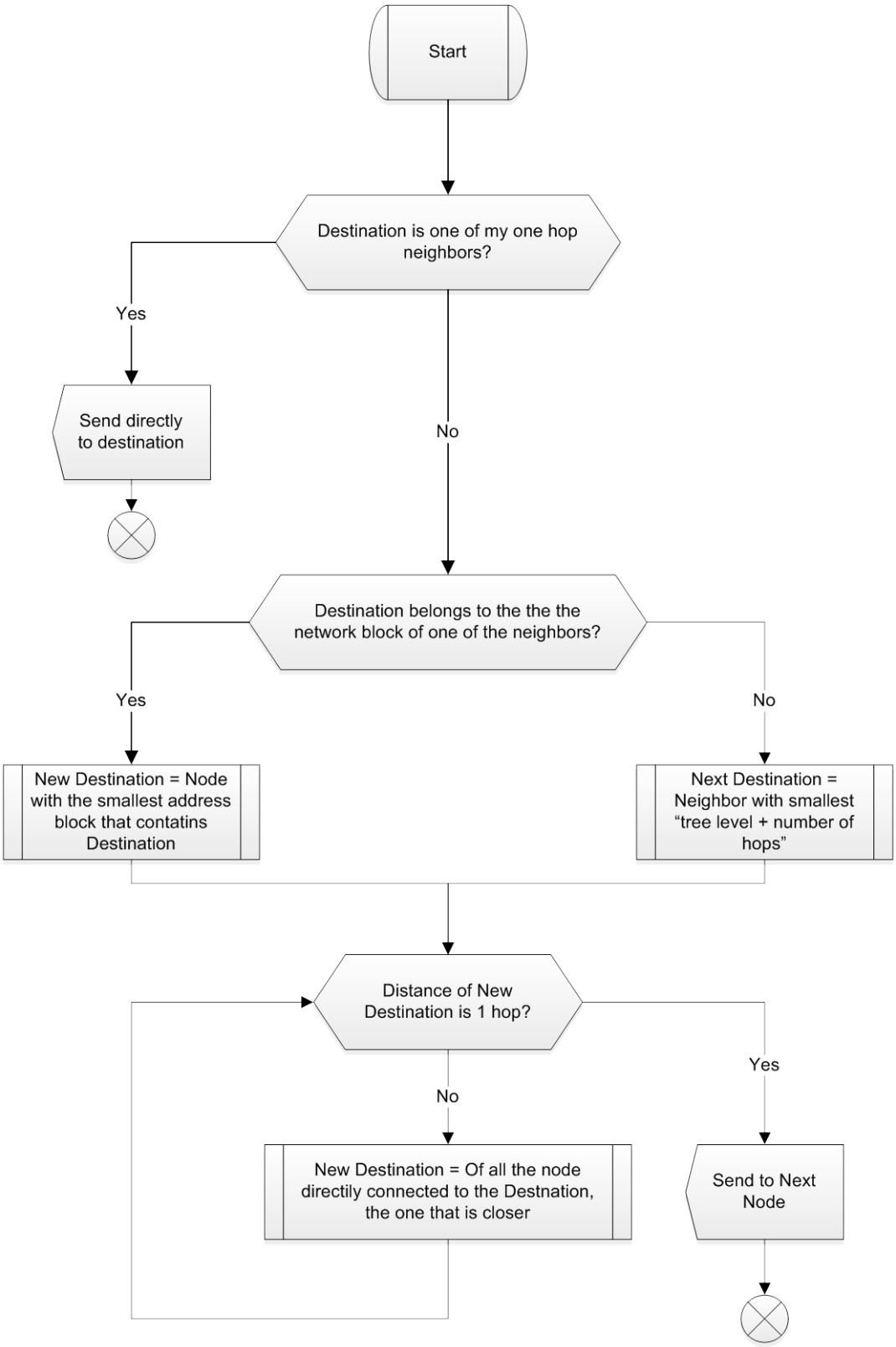


Figure 9 – Flow diagram of the routing algorithm.

2.4 Leaving the Network

A device may leave a mesh network upon receiving a MHME-LEAVE.request primitive from its application layer or upon receiving a leave command frame from its parent device. The first case is called active leaving while the second is called passive leaving.

2.4.1 Active leaving

In this case, the application layer of a mesh device decides to remove the device from a mesh network using MHME-LEAVE.request primitive (with the DeviceAddress parameter set to NULL). If the RemoveChildren parameter is equal to TRUE, the MHME first attempts to remove the device's children before leaving the network. If the RemoveChildren parameter is equal to FALSE, the MHME removes itself from the network. It then broadcasts a hello command frame, with the leaving network bit of the hello control field set to one, to its neighbours indicating it is leaving the network.

The MHME then clears all of its references to the current mesh network, including connectivity matrix and neighbour list. It also sets all MeshIB fields to their default values and issue an MLME-RESET.request primitive to the MAC sublayer to reset the MAC sublayer PIB.

Finally, the mesh sublayer issues a MHME-LEAVE.confirm to its application layer to indicate the completion of the leaving process. When the hello command frame with the leaving network bit set to one reaches neighbors of the leaving device, the neighbors check their relationship with the leaving device. If the hello command frame is received from a child device, the parent device updates its connectivity matrix, if necessary. It also changes the status of this child device in its neighbor list to "left" and starts a rejoin timer, meshRejoinTimer. The entry of the child device is kept in the neighbor list in case it rejoins the network after leaving. If the child device rejoins the network before the timer expires, it will be assigned the same mesh sublayer short address. When the timer expires and the child device has not rejoined the network, its entry is deleted from the neighbor list and all references to this device are removed. The parent device then sends its own hello command frame to update its neighbors with the change of link state.

If the hello command frame is received from a sibling device (neither a child device nor a parent device), the device follows all the processes the parent device does except that it deletes the leaving device from its neighbor list immediately and remove all the references to the leaving device. It does not need to start a timer and wait for the leaving device to rejoin the network.

2.4.2 Passive leaving

In the case of passive leaving, a mesh device receives leave command frame to request it to leave the mesh network. To achieve this, the application layer of a device's parent issues a MHME-LEAVE.request primitive to its mesh sublayer with the DeviceAddress parameter set to the child device's mesh address. The MHME of the parent device first determines whether the specified device is a child device. If the requested device does not exist, the MHME issues the MHME-LEAVE.confirm primitive with a status of UNKNOWN_CHILD_DEVICE. If the child device does exist, the parent device attempts to remove it from the network by issuing a leave

command frame to the child device. If the RemoveChildren parameter of the leave command is set to one then the device will be requested to remove its children as well. The mesh sublayer of the parent device issues a MHME-LEAVE.confirm to its application layer indicating the child device has been requested to leave. However, the real leaving process will be conducted when the hello command frame from the child device is received. Upon receiving the leave command, the child device removes itself from the network following the same process of active leaving. Basically it sends a hello command frame to all of its one-hop neighbors to indicate its leaving, clear all references to the mesh network at both the mesh sublayer and the MAC layer. Finally, the mesh sublayer of the child device issues a MHME-LEAVE.indication to its application layer to indicate the completion of the leaving process.

3 PRIMITIVES

3.1 Types

The service primitives are classified has:

- Request
- Confirm
- Indication
- Response

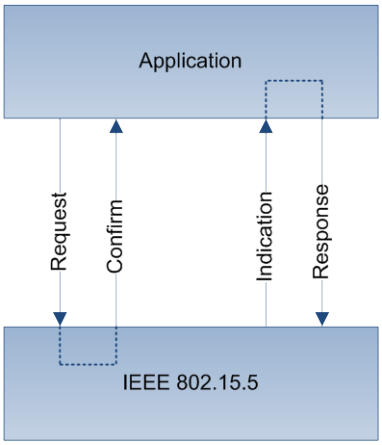


Figure 10 – Types of primitives.

The Application issues a request to MESH or MHME and them receives a Confirm to Request issued. Indications and Responses function in a similar away but with the roles reversed.

3.2 MESH Primitives

MESH primitives	Request	Confirm	Indication
MESH-DATA	Implemented	Implemented	Implemented
MESH-PURGE	Not Implemented	Not Implemented	-

Figure 11 – List of primitives offered by the IEEE 802.15.5 LR-WPAN MESH sublayer.

3.2.1 MESH-DATA.request

The application layer issues an MESH-DATA.request when it wishes to send a package.

An example of a possible usage of this primitive, using our implementation, is given in Listing 1.

Listing 1 - Example of a MESH-DATA.request.

```
//Set the type of the request
sMeshReq.u8Type=NET_MESH_REQ_DATA;

//Set the payload
u16MsgLengh=0;
sMeshReq.uParam.sMeshReqData.pu8Mhsdu[u16MsgLengh++]='1';
sMeshReq.uParam.sMeshReqData.pu8Mhsdu[u16MsgLengh++]='2';
sMeshReq.uParam.sMeshReqData.pu8Mhsdu[u16MsgLengh++]='3';
sMeshReq.uParam.sMeshReqData.pu8Mhsdu[u16MsgLengh++]='4';
sMeshReq.uParam.sMeshReqData.pu8Mhsdu[u16MsgLengh++]='5';
sMeshReq.uParam.sMeshReqData.pu8Mhsdu[u16MsgLengh++]='\n';
sMeshReq.uParam.sMeshReqData.pu8Mhsdu[u16MsgLengh++]='\r';
sMeshReq.uParam.sMeshReqData.pu8Mhsdu[u16MsgLengh++]='\0';

//Short Address for both source and destination
sMeshReq.uParam.sMeshReqData.u8SrcAddrMode=0x02;
sMeshReq.uParam.sMeshReqData.u8DstAddrMode=0x02;

//Send data to coordinator
sMeshReq.uParam.sMeshReqData.uDstAddr.u16Short=0x00;

//Set the length
sMeshReq.uParam.sMeshReqData.u8MhsduLength=u16MsgLengh;

//Request Ack (note that it is not an end-to-end ack, only a hop-by-hop ack)
sMeshReq.uParam.sMeshReqData.u8AckTransmission=TRUE;

//Multicast and broadcast are not supported
sMeshReq.uParam.sMeshReqData.u8McstTransmission=FALSE;
sMeshReq.uParam.sMeshReqData.u8BcstTransmission=FALSE;
sMeshReq.uParam.sMeshReqData.u8ReliableBcst=FALSE;
```

```
//Set the handle, the cofirm to this request will have an handle with the same
value
sMeshReq.uParam.sMeshReqData.u8MhsduHandle=sAppData.u8AppHandle++;

//Issue the request to the Mesh sublayer
vNetApiMeshRequest(&sMeshReq,&sMeshSyncCfm);
```

3.2.2 MESH-DATA.confirm

This primitive is generated by the mesh sublayer in response to a MESH-DATA.request, indicating if the operation was successful.

3.2.3 MESH-DATA.indication

This primitive is generated in response to a MAC-DATA.indication arrival indicating that a packet has arrived for the higher layers. This primitive is generated within the mesh sublayer.

3.3 MHME Primitives

MHME primitives	Request	Confirm	Indication	Response
MHME-DISCOVER	Implemented	Implemented	-	-
MHME-START-NETWORK	Implemented	Implemented	-	-
MHME-START-DEVICE	Implemented	Implemented	-	-
MHME-JOIN	Implemented	Implemented	Implemented	-
MHME-LEAVE	Implemented	Implemented	Implemented	-
MHME-RESET	Implemented	Implemented	-	-
MHME-GET	Implemented	Implemented	-	-
MHME-SET	Implemented	Implemented	-	-
MHME-START-SYNC (Optional)	Not Implemented	Not Implemented	-	-
MHME-TRACE-ROUTE (Optional)	Not Implemented	Not Implemented	Not Implemented	-
MHME-MULTICAST-JOIN	Not Implemented	Not Implemented	-	-
MHME-MULTICAST-LEAVE	Not Implemented	Not Implemented	-	-

Figure 12 - List of primitives offered by the IEEE 802.15.5 LR-WPAN MHME sublayer.

These services are used for mesh network management. We implemented eight types of primitives: network discovering related primitives (MHME-DISCOVER), network creation related primitives (MHME-START-NETWORK), initialization/resetting related primitives (MHME-START-DEVICE and MHME-RESET), network associating related primitives (MHME-JOIN and MHME-LEAVE), and attribute management related primitives (MHME-GET and MHME-SET).

3.3.1 MHME-DISCOVER.request

This primitive allows the application layer to request the mesh sublayer to discover mesh networks currently operating within the neighbourhood.

An example of a possible usage of this primitive, using our implementation, is given in Listing 2.

Listing 2 – Example of a MHME-DISCOVER.request.

```
//Set the type of the request
sMhmeReqRsp.u8Type=NET_MHME_REQ_DISCOVER;

//The 27 bits (b0, b1,... b26) indicate which channels are to be scanned (1 =
scan, 0 = do not scan) for each of the 27 channels supported by the
ChannelPage parameter as defined in IEEE Std 802.15.4-2006.
//In this case we will the 26 scan channel
sMhmeReqRsp.uParam.sReqDiscover.u32ScanChannels=0x02000000;

//A value used to calculate the length of time to spend scanning each channel.
sMhmeReqRsp.uParam.sReqDiscover.u8ScanDuration=6;

//The channel page on which to perform the scan.
sMhmeReqRsp.uParam.sReqDiscover.u8ChannelPage=1;

//The field indicates which criterion is used to select the best neighbor to
be reported to the application layer.
sMhmeReqRsp.uParam.sReqDiscover.eReportCriteria=LINK_QUALITY;

//Issue the request to the Mhme sublayer
vNetApiMhmeRequest(&sMhmeReqRsp, &sMhmeSyncCfm);
```

3.3.2 MHME-DISCOVER.confirm

This primitive provides the application layer with the results of a MHME-DISCOVER.request. It is generated by the MHME and issued to its application layer.

3.3.3 MHME-START-NETWORK.request

This primitive allows the application layer to request that the device start a new mesh network with itself as the mesh coordinator (MC). It is generated by the application layer of a mesh coordinator-capable device and issued to its MHME.

Listing 34 – Example of a MHME-START-NETWORK.request.

```
sMhmeReqRsp.u8Type = NET_MHME_REQ_START_NETWORK;
sMhmeReqRsp.uParam.sReqStartNetwork.u8BeaconOrder=0x0f;
sMhmeReqRsp.uParam.sReqStartNetwork.u8SuperFrameOrder=0x0f;
vNetApiMhmeRequest(&sMhmeReqRsp,&sMhmeSyncCfm);
```

3.3.4 MHME-START-NETWORK.confirm

This primitive allows the report of the result of a MHME-START-NETWORK.request to the application layer. It is generated by the MHME and issued to its application layer.

3.3.5 MHME-START-DEVICE.request

This primitive allows the application layer to request for the device to initialize the mesh functions. It is generated by the application layer of a mesh-capable device and issued to its MHME.

An example of a possible usage of this primitive, using our implementation, is given in Listing 5.

Listing 5– Example of a MHME-START-DEVICE.request.

```
sMhmeReqRsp.u8Type=NET_MHME_REQ_START_DEVICE;
// The beacon order of the network that the higher layers wish to form.
For this version of the recommended practice, the value is always set
to 0x0f indicating no periodic beacons are transmitted.
sMhmeReqRsp.uParam.sReqStartDevice.u8BeaconOrder=0x0f;
// The superframe order of the network that the higher layers wish to
form. For this version of the recommended practice, the value is always
set to 0x0f indicating no periodic beacons are transmitted.
sMhmeReqRsp.uParam.sReqStartDevice.u8SuperFrameOrder=0x0f;
//Issue the request to the Mhme sublayer
vNetApiMhmeRequest(&sMhmeReqRsp,&sMhmeSyncCfm);
```

3.3.6 MHME-START-DEVICE.confirm

This primitive is responsible to report to the application layer the result of a MHME-START-DEVICE.request. It is generated by the MHME and issued to its application layer.

3.3.7 MHME-JOIN.request

This primitive allows the application layer to request for an association with a device of a neighbour network. It is generated by the application layer of a mesh-capable device and issued to its MHME.

An example of a possible usage of this primitive, using our implementation, is given in Listing 6.

Listing 6– Example of a MHME-JOIN.request.

```
//Calling mhme services
sMhmeReqRsp.u8Type=NET_MHME_REQ_JOIN;

//The value is set to TRUE if direct joining is chosen; otherwise, its value
is FALSE.
sMhmeReqRsp.uParam.sReqJoin.u8DirectJoin=FALSE;

//The address of the parent device we wish to join
sMhmeReqRsp.uParam.sReqJoin.u8AddrMode=sChosenDescriptor.u8AddrMode;
memcpy(&sMhmeReqRsp.uParam.sReqJoin.uParentDevAddr,&sChosenDescriptor.uAddr,si
zeof(MAC_Addr_u));

//The 16-bit PAN identifier of the network to join. This field will be read
only when the DirectJoin parameter has a value equal to TRUE.
sMhmeReqRsp.uParam.sReqJoin.u16PanId=sChosenDescriptor.u16PanId;

//This parameter controls the method of joining the network.
```

```
sMhmeReqRsp.uParam.sReqJoin.u8RejoinNetwork=FALSE;

//The parameter is set to TRUE if the device is going to function as a mesh
device; it is set to FALSE if the device is going to function as an end
device.
sMhmeReqRsp.uParam.sReqJoin.u8JoinAsMeshDevice=TRUE;

//The 27 bits (b0, b1,... b26) indicate which channels are to be scanned (1 =
scan, 0 = do not scan) for each of the 27 channels supported by the
ChannelPage parameter. This field will be read only when the DirectJoin
parameter has a value equal to FALSE.
sMhmeReqRsp.uParam.sReqJoin.u32ScanChannels=0x0f000000;
//A value used to calculate the length of time to spend scanning each channel.
sMhmeReqRsp.uParam.sReqJoin.u8ScanDuration=6;

//The channel page on which to perform the scan. This field will be read only
when the DirectJoin parameter has a value equal to FALSE.
sMhmeReqRsp.uParam.sReqJoin.u8ChannelPage=1;

//This field is set to TRUE if the joining device is a mesh device. Otherwise,
it is set to FALSE.
sMhmeReqRsp.uParam.sReqJoin.u8DeviceType=TRUE;

//This field is set to TRUE if it is mainspowered. Otherwise, it is set to
FALSE.
sMhmeReqRsp.uParam.sReqJoin.u8PowerSource=TRUE;

//This field is set to TRUE if the receiver is enabled when the device is
idle. Otherwise, it is set to FALSE.
sMhmeReqRsp.uParam.sReqJoin.u8ReceiverOnWhenIdle=TRUE;

//This field is always set to TRUE in the implementations of this recommended
practice, indicating that the joining device should be issued a 16-bit network
address.
sMhmeReqRsp.uParam.sReqJoin.u8AllocateAddress=TRUE;

//Issue the request to the Mhme sublayer
vNetApiMhmeRequest(&sMhmeReqRsp,&sMhmeSyncCfm);
```

3.3.8 MHME-JOIN.confirm

This primitive allows the report of the result of a MHME-JOIN.request to the application layer. It is generated by the MHME and issued to its application layer.

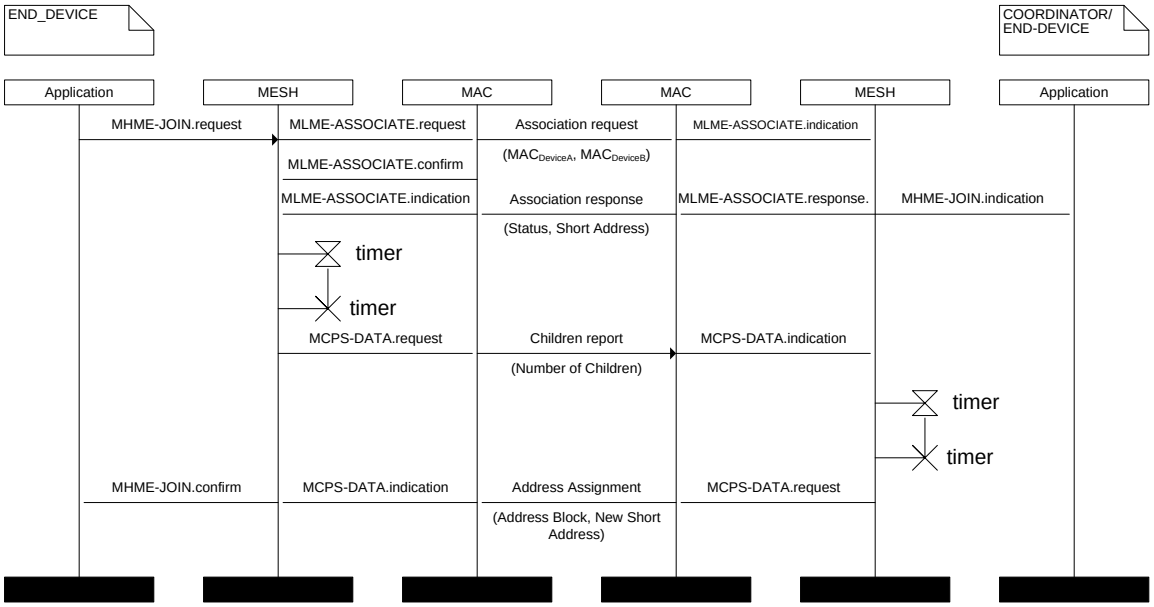


Figure 13 – The Join process.

3.3.9 MHME-LEAVE.request

This primitive allows the application layer to request for an association termination with a device of a neighbour network. It is generated by the application layer of a mesh-capable device and issued to its MHME.

An example of a possible usage of this primitive, using our implementation, is given in Listing 7.

Listing 7– Example of a MHME-LEAVE.request.

```
sMhmeReq.u8Type=NET_MHME_REQ_LEAVE;

//This parameter has a value of TRUE if the device is asked to remove itself
from the network. Otherwise it has a value of FALSE.
sMhmeReq.uParam.sReqLeave.u8RemoveSelf=FALSE;

//This parameter has a value of TRUE if the device being asked to leave the
network is also being asked to remove its child devices, if any. Otherwise it
has a value of FALSE.
sMhmeReq.uParam.sReqLeave.u8RemoveChildren=FALSE;

//The 64-bit IEEE address of a child device to be removed from the network.
memcpy(&sMhmeReq.uParam.sReqLeave.sDeviceAddress,&psNeighbor[j].sExt,sizeof(MA
C_ExtAddr_s));

//Issue the request to the Mhme sublayer
vNetApiMhmeRequest(&sMhmeReq,&sMhmeSyncCfm)
```

3.3.10 MHME-LEAVE.confirm

This primitive allows the report of the result of a MHME-LEAVE.request to the application layer. It is generated by the MHME and issued to its application layer.

3.3.11 MHME-RESET.request

This primitive allows the application layer to reset the MHME. It is generated by the application layer of a mesh-capable device and issued to its MHME.

3.3.12 MHME-RESET.confirm

This primitive allows the report of the result of a MHME-LEAVE.request to the application layer. It is generated by the MHME and issued to its application layer.

An example of a possible usage of this primitive, using our implementation, is given in Listing 8.

Listing 8– Example of a MHME-RESET.request.

```
sMhmeReqRsp.u8Type=NET_MHME_REQ_RESET;

//Issue the request to the Mhme sublayer
vNetApiMhmeRequest(&sMhmeReqRsp,&sMhmeSyncCfm);
```

3.3.13 MHME-GET.request

This primitive allows the application layer to request MIB information. It is generated by the application layer of a mesh-capable device and issued to its MHME.

An example of a possible usage of this primitive, using our implementation, is given in Listing 9.

Listing 9– Example of a MHME-GET.request.

```
NET_Neighbor_s*     psNeighbor;
sMhmeReq.u8Type=NET_MHME_REQ_GET;

//The identifier of the MeshIB attribute to read.
sMhmeReq.uParam.sReqGet.u8MeshIBAttribute=MESH_NEIGHBOR_LIST;

//Issue the request to the Mhme sublayer
vNetApiMhmeRequest(&sMhmeReq,&sMhmeSyncCfm);

if(sMhmeSyncCfm.uParam.sCfmGet.eStatus!=SUCCESS) break;
psNeighbor=(NET_Neighbor_s*)sMhmeSyncCfm.uParam.sCfmGet.psMibAttributeValue;
```

3.3.14 MHME-GET.confirm

This primitive allows the report of the result of a MHME-GET.request to the application layer. It is generated by the MHME and issued to its application layer.

3.3.15 MHME-SET.request

This primitive allows the application layer to set MIB information. It is generated by the application layer of a mesh-capable device and issued to its MHME.

3.3.16 MHME-SET.confirm

This primitive allows the report of the result of a MHME-SET.request to the application layer. It is generated by the MHME and issued to its application layer.

4 FRAMES

This section describes frame types and their fields.

4.1 The general frame format

Similar to the MAC sublayer frame format, the mesh PDU will be divided in a mesh sublayer header and in a mesh sublayer payload as shown in Figure 14.

<table>
<tr><td>Octets: 2</td><td>8/2</td><td>8/2</td><td>Variable</td></tr>
<tr><td>Frame Control</td><td>Destination Address</td><td>Source Address</td><td rowspan="2">Mesh Sublayer payload</td></tr>
<tr><td colspan="3">Mesh Sublayer Header</td></tr>
</table>

Figure 14 – General Mesh frame format.

Bits: 0-3	4	5	6	7-10	11-15
Protocol Version	Frame Type	Destination Address Mode	Source Address Mode	Transmission Options	Reserved

Figure 15 – Frame Control field.

The frame control field of mesh PDU is composed of the fields presented in Figure 15.

Depending on the value of the frame type field, a mesh PDU can be:

- 0 – Data Frame.
- 1 – Command Frame.

It should be noted that besides this two types there are also beacon frames.

4.2 Data Frame

The structure of the Data Frame is the same as the general structure, but now with a defined payload, as shown in Figure 16.

<table>
<tr><td>Octets: 2</td><td>8/2</td><td>8/2</td><td>1</td><td>1</td><td>Variable</td></tr>
<tr><td>Frame Control</td><td>Destination Address</td><td>Source Address</td><td>Sequence Number</td><td>Routing Control</td><td>Data Payload</td></tr>
<tr><td colspan="3">Mesh Sublayer Header</td><td colspan="3">Mesh Sublayer Payload</td></tr>
</table>

Figure 16 – Format of the data frame.

4.3 Command Frame

The general command frame format is presented in Figure 17. The command frame identifier sets the command payload structure.

Octets: 2	8/2	8/2	1	Variable
Frame Control	Destination Address	Source Address	Command Frame Identifier	Command Payload
Mesh Sublayer Header			Mesh Sublayer Payload	

Figure 17 – Format of the command frame.

The command is type is identified by the value of command frame identifier field, as given in Table 3.

Table 3 – Command frame identifier value and the corresponding type.

Command frame identifier	Command name
0x01	Children number report
0x02	Address assignment
0x03	Hello
0x17	Leave

If the command is a children number report, then the command payload has the format presented in Figure 18.

1	2	3
Command Frame Identifier	Number of Descendants	Number of Requested Addresses
Data Frame Mesh Sublayer Payload		

Figure 18 – Children number report command frame payload format.

If the command is an address assignment, then the command payload has the format presented in Figure 19.

1	2	3	4
Command Frame Identifier	Beginning Address	Ending Address	Tree Level of Parent Device
Data Frame Mesh Sublayer Payload			

Figure 19 – Address assignment command frame payload format.

If the command is a hello, then the command payload has the format presented in Figure 20.

Octets: 1	1	2	2	2	1	1	1	Variable	Variable
Command Frame Identifier	TTL	Beginning Address	Ending Address	Tree Level	Hello Control	Number of One-hop Neighbors	Number of Multicast Groups	Addresses of One-hop Neighbors	Addresses of Multicast Groups
Data Frame Mesh Sublayer Payload									

Figure 20 – Hello command frame payload format.

If the command is a leave, then the command payload has the format presented in Figure 21.

Octets: 1	1
Command Frame Identifier	Leave Control
Data Frame Mesh Sublayer Payload	

Figure 21 – Leave command frame payload format.

5 MESH STACK API

Our implementation of IEEE 802.15.5 gives developers an application programming interface (API). Through this API developers can easily make use of the IEEE 802.15.5 mesh capabilities in their applications.

The basic structure of an application using the Mesh Stack is provided in Listing 10. The application initializes the Integrated Peripherals API, followed by the MeshStack. Next the application enters the main loop, where it will call `vAppPerformTasks()`, a function defined by the programmer where the bulk of the processing by the application is done, and requests to the Mesh Stack are issued. Next the application checks whether a confirm or response was sent by the lower layers, if so it deals with then within the `vAppProccessMessage(&sMhmeDcfmInd)` and `vAppProccessData()` functions, usually by setting flags that define a state machine.

Listing 10 – Basic structure for an application using our Mesh Stack.

```
PUBLIC void App(void)
{
    NET_MhmeDcfmInd_s sMhmeDcfmInd;
    NET_MeshDcfmInd_s sMeshDcfmInd;

    //Initialize Integrated Peripherals API
    u32AHI_Init();
    //Initialize Mesh Stack
    vInitMeshStack();

    //App Main Loop
    while(1)
    {
        //Application management – State Machine based recommended
        vAppPerformTasks();

        // Get message from MHME sublayer
        if(u16MhmeGetMessage(&sMhmeDcfmInd))
        {
            //Get message from the MHME queue
            vAppProccessMessage(&sMhmeDcfmInd);
        }

        //Get message from MESH sublayer
        if(u16MeshGetMessage(&sMeshDcfmInd))
        {
            //Get message from the MESH queue
            vAppProcessData(&sMeshDcfmInd);
        }

    }

}
```

The process of starting a network has been described in section 2.2-Network Establishment, and the necessary primitives are described with examples in section 3, as also are basic functions such as sending packets.

In the annex a listing of the code of the Mesh Stack is presented. The code includes a simple application where which node periodically sends a packet with topology information to the coordinator. This allows the network's monitorization and visualization, as depicted in Figure 22.

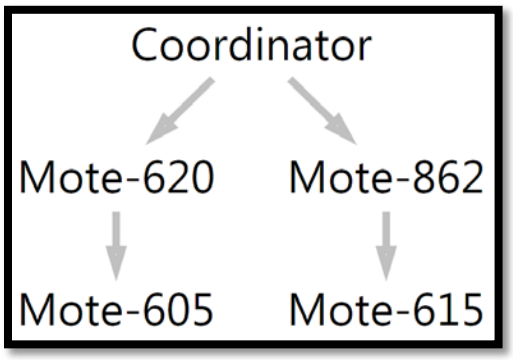


Figure 22 – Topology of a network using our implementation of IEEE 802.15.5.

ACRONYMS

API	Application Programming Interface
LQI	Link Quality Indication
LR	Low Rate
LST	Link State Table
MAC	Medium Access Control
MHME	Mesh Sublayer Management Entity
MeshIB	Mesh Information Base
TTL	Time to Live
WPAN	Wireless Personal Area Network
WSN	Wireless sensor networks

BIBLIOGRAPHY

1. **IEEE Computer Society.** *IEEE std 802.15.5.* New York : s.n., 2009.

2. **JENNIC.** About JENNIC. *JENNIC.* [Online] 2010. [Cited: 18 March 2010.] http://www.jennic.com/company/index.php.

3. *A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols.* **Trung, Ha Duyen and Benjapolakul, Watit and Duc, Phan Minh.** Newton : Butterworth-Heinemann, 2007, Vol. 30. 0140-3664.

4. **Myung Lee and Rui Zhang, City University of New York, et al.** Meshing Wireless Personal Area Networks: Introducing IEEE 802.15.5. *IEEE Communications Magazine.* 2009.

5. **IEEE Computer Society.** *IEEE Std 802.15.4.* New York : IEEE, 2006.

6. **JENNIC.** *802.15.4 Stack API.* s.l. : JENNIC, 2008.

7. —. JENNIC. [Online] 2010. [Cited: 18 March 2010.] http://www.jennic.com/.

LISTINGS

I. App.c

```
0001 /**
0002  *  @defgroup   gNetImp IEEE 802.15.5 low-rate WPAN mesh Partial Implementation
0003  */
0004
0005 /**
0006  * @defgroup gNetMesh MESH Sublayer
0007  * @ingroup  gNetImp
0008  */
0009
0010  /**
0011  * @defgroup gNetMHME MHME
0012  * @ingroup  gNetImp
0013  */
0014
0015 /**
0016  *  @defgroup   gGenericApp        Generic Application
0017  *  A generic application using the mesh library, mostly used for test purposes.
0018  *  @ingroup    gNetImp
0019  *  @{
0020  */
0021 #define ROM_HANDLER 0x00012D70
0022
0023 #include <jendefs.h>
0024 #include <AppHardwareApi.h>
0025 #include <mac_sap.h>
0026 #include <mac_pib.h>
0027 #include <string.h>
0028 #include "mhme.h"
0029 #include "mesh.h"
0030 #include "app.h"
0031
0032 #include "config.h"
0033
0034 #ifdef DEBUG
0035 #include "Debugger.h"
0036 #endif
0037
0038 #include "VirtualTimer.h"
0039
0040 extern NET_MeshData_s sMeshData;
0041
0042 PUBLIC uint16 u16UTIL_NumToString(uint32 u32Data, uint8* pcString)
0043 {
0044     int    i;
0045     uint8  u8Nybble;
0046     uint16 u16Counter=0;
0047
0048     for (i = 28; i >= 0; i -= 4)
0049     {
0050         u8Nybble = (uint8)((u32Data >> i) & 0x0f);
0051         u8Nybble += 0x30;
0052         if (u8Nybble > 0x39)
0053             u8Nybble += 7;
0054
0055         *pcString = u8Nybble;
0056         pcString++;
0057         u16Counter++;
0058     }
0059     *pcString = 0;
0060     return u16Counter;
0061 }
```

```
0062
0063
0064 /**
0065  *  \brief The application starts running here.
0066  *  First function called (equivalent to usual main()).
0067  */
0068
0069  NET_AppData_s sAppData;
0070
0071 PRIVATE void vAppInit(void)
0072 {
0073     sAppData.u8AppHandle=0;
0074     sAppData.i8DataTimer=-1;
0075     sAppData.u8DataTime=15;
0076     sAppData.i8LeaveTimer=-1;
0077     sAppData.u8LeaveTime=40;
0078     sAppData.i8RejoinTimer=-1;
0079     sAppData.u8RejoinTime=10;
0080     sAppData.i8OrderLeaveChildTimer=-1;
0081     sAppData.u8OrderLeaveChildTime=255;
0082     sAppData.i8OrderLeaveParentTimer=-1;
0083     sAppData.u8OrderLeaveParentTime=255;
0084 }
0085
0086
0087
0088 PRIVATE void vAppPerformTasks(void)
0089 {
0090     //Register this function on the debugger
0091     #ifdef DEBUG
0092     u8StackPushIdentifier("vAppPerformTasks",strlen("vAppPerformTasks"),FALSE);
0093     #endif
0094
0095     //Decide what to do
0096     switch(u8Flag)
0097     {
0098         //If we are begining
0099         case BEGIN:
0100         {
0101             NET_MhmeReqRsp_s    sMhmeReqRsp;
0102             NET_MhmeSyncCfm_s   sMhmeSyncCfm;
0103
0104             #ifdef DEBUG
0105             u8StackPushIdentifier("BEGIN",strlen("BEGIN"),FALSE);
0106             #endif
0107
0108             //If it is a coordinator, start the network
0109             if(COORDINATOR)
0110             {
0111                 #ifdef DEBUG
0112                 vStackPrintf(__FILE__,__LINE__,"Starting network.");
0113                 #endif
0114
0115                 //Calling mhme services
0116                 sMhmeReqRsp.u8Type = NET_MHME_REQ_START_NETWORK;
0117                 sMhmeReqRsp.uParam.sReqStartNetwork.u8BeaconOrder=0x0f;
0118                 sMhmeReqRsp.uParam.sReqStartNetwork.u8SuperFrameOrder=0x0f;
0119                 vNetApiMhmeRequest(&sMhmeReqRsp,&sMhmeSyncCfm);
0120                 if(sMhmeSyncCfm.uParam.sCfmStartNetwork.u8Status==MAC_MLME_CFM_OK)
0121                 {
0122                     #ifdef DEBUG
0123                     vStackPrintf(__FILE__,__LINE__,"Network Started.");
0124                     #endif
0125                     u8Flag=NETWORK_STARTED;
0126                 }
```

```
0127                else
0128                {
0129                    #ifdef DEBUG
0130                    vStackPrintf(__FILE__,__LINE__,"Could not start the Network.");
0131                    #endif
0132                    u8Flag=START_NETWORK_FAILED;
0133                }
0134            }
0135
0136
0137
0138            //If it is a mesh device, discover networks
0139            else
0140            {
0141                #ifdef DEBUG
0142                vStackPrintf(__FILE__,__LINE__,"Discovering networks.");
0143                #endif
0144
0145                //Calling mhme services
0146                sMhmeReqRsp.u8Type=NET_MHME_REQ_DISCOVER;
0147                sMhmeReqRsp.uParam.sReqDiscover.u32ScanChannels=0x02000000;
0148                sMhmeReqRsp.uParam.sReqDiscover.u8ScanDuration=6;
0149                sMhmeReqRsp.uParam.sReqDiscover.u8ChannelPage=1;
0150                sMhmeReqRsp.uParam.sReqDiscover.eReportCriteria=LINK_QUALITY;
0151                vNetApiMhmeRequest(&sMhmeReqRsp, &sMhmeSyncCfm);
0152                u8Flag=DISCOVERING;
0153            }
0154
0155            #ifdef DEBUG
0156            vStackPopIdentifier();
0157            #endif
0158        }break;
0159
0160        //If network is started
0161        case NETWORK_STARTED:
0162        {
0163        }break;
0164
0165        //If start network had failed
0166        case START_NETWORK_FAILED:
0167        {
0168            //Is this case really necessary?
0169            u8Flag=BEGIN;
0170        }break;
0171
0172        //If we do not have the responses of the descover
0173        case DISCOVERING:
0174        {
0175        }break;
0176
0177        //If the results of the descover request have come back
0178        case DISCOVERED:
0179        {
0180            #ifdef DEBUG
0181            vStackPrintf(__FILE__,__LINE__,"Discovering phase is over.");
0182            #endif
0183            u8Flag=JOINING;
0184        }break;
0185
0186        //If the device is trying to join a network
0187        case JOINING:
0188        {
0189            NET_MhmeReqRsp_s    sMhmeReqRsp;
0190            NET_MhmeSyncCfm_s   sMhmeSyncCfm;
0191
```

```
0192            #ifdef DEBUG
0193            vStackPrintf(__FILE__,__LINE__,"Trying to join a network.");
0194            #endif
0195
0196            //Calling mhme services
0197            sMhmeReqRsp.u8Type=NET_MHME_REQ_JOIN;
0198            sMhmeReqRsp.uParam.sReqJoin.u8DirectJoin=FALSE;
0199            sMhmeReqRsp.uParam.sReqJoin.u8AddrMode=sChosenDescriptor.u8AddrMode;
0200
memcpy(&sMhmeReqRsp.uParam.sReqJoin.uParentDevAddr,&sChosenDescriptor.uAddr,sizeof(MAC_Addr_u));
0201            sMhmeReqRsp.uParam.sReqJoin.u16PanId=sChosenDescriptor.u16PanId;
0202            sMhmeReqRsp.uParam.sReqJoin.u8RejoinNetwork=FALSE;
0203            sMhmeReqRsp.uParam.sReqJoin.u8JoinAsMeshDevice=TRUE;
0204            sMhmeReqRsp.uParam.sReqJoin.u32ScanChannels=0x0f000000; //OIIII!
0205            sMhmeReqRsp.uParam.sReqJoin.u8ScanDuration=6;
0206            sMhmeReqRsp.uParam.sReqJoin.u8ChannelPage=1;
0207            sMhmeReqRsp.uParam.sReqJoin.u8DeviceType=TRUE;
0208            sMhmeReqRsp.uParam.sReqJoin.u8PowerSource=TRUE;
0209            sMhmeReqRsp.uParam.sReqJoin.u8ReceiverOnWhenIdle=TRUE;
0210            sMhmeReqRsp.uParam.sReqJoin.u8AllocateAddress=TRUE;
0211            vNetApiMhmeRequest(&sMhmeReqRsp,&sMhmeSyncCfm);
0212            #ifdef DEBUG
0213            vStackPrintf(__FILE__,__LINE__,"Join Request Sent to MHME");
0214            #endif
0215            if(sMhmeSyncCfm.uParam.sCfmJoin.u8Status==SUCCESS)
0216                u8Flag=READY;
0217            else
0218                u8Flag=BEGIN;
0219        }break;
0220
0221        //If the device is initializing its network capabilities
0222        case START_DEVICE:
0223        {
0224            NET_MhmeReqRsp_s    sMhmeReqRsp;
0225            NET_MhmeSyncCfm_s   sMhmeSyncCfm;
0226
0227            #ifdef DEBUG
0228            vStackPrintf(__FILE__,__LINE__,"Trying to start the device.");
0229            #endif
0230
0231            sMhmeReqRsp.u8Type=NET_MHME_REQ_START_DEVICE;
0232            sMhmeReqRsp.uParam.sReqStartDevice.u8BeaconOrder=0x0f;
0233            sMhmeReqRsp.uParam.sReqStartDevice.u8SuperFrameOrder=0x0f;
0234            vNetApiMhmeRequest(&sMhmeReqRsp,&sMhmeSyncCfm);
0235            u8Flag=READY;
0236        }break;
0237
0238        //If there is data to send to other devices
0239        case DATA_READY:
0240        {
0241            NET_MeshReq_s       sMeshReq;
0242            NET_MeshSyncCfm_s   sMeshSyncCfm;
0243            NET_MhmeReqRsp_s    sMhmeReq;
0244            NET_MhmeSyncCfm_s   sMhmeSyncCfm;
0245            NET_Neighbor_s*     psNeighbor;
0246            uint16              u16NetworkAddress;
0247            MAC_ExtAddr_s       sMacAddress;
0248            uint16              u16ParentShort=0xffff;
0249            MAC_ExtAddr_s       sParentExt;
0250            uint8               i;
0251            uint16              u16MsgLengh;
0252
0253
0254            //Getting the neighbor list, the network address and the mac address
0255            sMhmeReq.u8Type=NET_MHME_REQ_GET;
```

```
0256            sMhmeReq.uParam.sReqGet.u8MeshIBAttribute=MESH_NEIGHBOR_LIST;
0257            vNetApiMhmeRequest(&sMhmeReq,&sMhmeSyncCfm);
0258            if(sMhmeSyncCfm.uParam.sCfmGet.eStatus!=SUCCESS) break;
0259            psNeighbor=(NET_Neighbor_s*)sMhmeSyncCfm.uParam.sCfmGet.psMibAttributeValue;
0260
0261            sMhmeReq.u8Type=NET_MHME_REQ_GET;
0262            sMhmeReq.uParam.sReqGet.u8MeshIBAttribute=MESH_NETWORK_ADDRESS;
0263            vNetApiMhmeRequest(&sMhmeReq,&sMhmeSyncCfm);
0264            if(sMhmeSyncCfm.uParam.sCfmGet.eStatus!=SUCCESS) break;
0265            u16NetworkAddress=*(uint16*)sMhmeSyncCfm.uParam.sCfmGet.psMibAttributeValue;
0266
0267            sMhmeReq.u8Type=NET_MHME_REQ_GET;
0268            sMhmeReq.uParam.sReqGet.u8MeshIBAttribute=MESH_ADDRESS_MAPPING;
0269            vNetApiMhmeRequest(&sMhmeReq,&sMhmeSyncCfm);
0270            if(sMhmeSyncCfm.uParam.sCfmGet.eStatus!=SUCCESS) break;
0271            memcpy(&sMacAddress,&((NET_AddrMapping_s*)sMhmeSyncCfm.uParam.sCfmGet.psMibAttributeValue)-
>sExtAddr,sizeof(MAC_ExtAddr_s));
0272
0273            //Processing the neighbor litst to get the parents
0274            for(i=0;i<MAX_NEIGHBORS;i++)
0275            {
0276                if(psNeighbor[i].u8Relationship==PARENT)
0277                {
0278                    memcpy(&sParentExt,&psNeighbor[i].sExt,sizeof(MAC_ExtAddr_s));
0279                    u16ParentShort=psNeighbor[i].u16BeginningAddress;
0280                    break;
0281                }
0282            }
0283
0284            //Creating the message to the coordinator
0285            u16MsgLengh=0;
0286            sMeshReq.uParam.sMeshReqData.pu8Mhsdu[u16MsgLengh++]='$';
0287
u16MsgLengh+=u16UTIL_NumToString(u16NetworkAddress,&sMeshReq.uParam.sMeshReqData.pu8Mhsdu[u16MsgLengh]);
0288            sMeshReq.uParam.sMeshReqData.pu8Mhsdu[u16MsgLengh++]=',';
0289
u16MsgLengh+=u16UTIL_NumToString(sMacAddress.u32H,&sMeshReq.uParam.sMeshReqData.pu8Mhsdu[u16MsgLengh]);
0290
u16MsgLengh+=u16UTIL_NumToString(sMacAddress.u32L,&sMeshReq.uParam.sMeshReqData.pu8Mhsdu[u16MsgLengh]);
0291            sMeshReq.uParam.sMeshReqData.pu8Mhsdu[u16MsgLengh++]=',';
0292
u16MsgLengh+=u16UTIL_NumToString(u16ParentShort,&sMeshReq.uParam.sMeshReqData.pu8Mhsdu[u16MsgLengh]);
0293            sMeshReq.uParam.sMeshReqData.pu8Mhsdu[u16MsgLengh++]=',';
0294
u16MsgLengh+=u16UTIL_NumToString(sParentExt.u32H,&sMeshReq.uParam.sMeshReqData.pu8Mhsdu[u16MsgLengh]);
0295
u16MsgLengh+=u16UTIL_NumToString(sParentExt.u32L,&sMeshReq.uParam.sMeshReqData.pu8Mhsdu[u16MsgLengh]);
0296            sMeshReq.uParam.sMeshReqData.pu8Mhsdu[u16MsgLengh++]='$';
0297            sMeshReq.uParam.sMeshReqData.pu8Mhsdu[u16MsgLengh++]='\n';
0298            sMeshReq.uParam.sMeshReqData.pu8Mhsdu[u16MsgLengh++]='\r';
0299            sMeshReq.uParam.sMeshReqData.pu8Mhsdu[u16MsgLengh++]='\0';
0300
0301            //Sending the message to the coordinator
0302            sMeshReq.u8Type=NET_MESH_REQ_DATA;
0303            sMeshReq.uParam.sMeshReqData.u8SrcAddrMode=0x02;
0304            sMeshReq.uParam.sMeshReqData.u8DstAddrMode=0x02;
0305            sMeshReq.uParam.sMeshReqData.uDstAddr.u16Short=0x0;
0306            sMeshReq.uParam.sMeshReqData.u8MhsduLength=u16MsgLengh;
0307            sMeshReq.uParam.sMeshReqData.u8AckTransmission=TRUE;
0308            sMeshReq.uParam.sMeshReqData.u8McstTransmission=FALSE;
0309            sMeshReq.uParam.sMeshReqData.u8BcstTransmission=FALSE;
0310            sMeshReq.uParam.sMeshReqData.u8ReliableBcst=FALSE;
0311            sMeshReq.uParam.sMeshReqData.u8MhsduHandle=sAppData.u8AppHandle++;
0312            vNetApiMeshRequest(&sMeshReq,&sMeshSyncCfm);
0313
```

```
0314            //Return to idle
0315            u8Flag=READY;
0316        }break;
0317
0318        //If there is a request to leave
0319        case LEAVING:
0320        {
0321            NET_MhmeReqRsp_s    sMhmeReqRsp;
0322            NET_MhmeSyncCfm_s   sMhmeSyncCfm;
0323
0324            #ifdef DEBUG
0325            vStackPrintf(__FILE__,__LINE__,"Trying to join a network.");
0326            #endif
0327
0328            //Calling mhme services to issiue a reset (with leaving the network)
0329            sMhmeReqRsp.u8Type=NET_MHME_REQ_RESET;
0330            vNetApiMhmeRequest(&sMhmeReqRsp,&sMhmeSyncCfm);
0331
0332            //Starting the rejoin timer
0333            sAppData.i8RejoinTimer=VirtualTimer_i8New(10*sAppData.u8RejoinTime);
0334            VirtualTimer_bReset(sAppData.i8RejoinTimer);
0335            VirtualTimer_bCount(sAppData.i8RejoinTimer);
0336            #ifdef DEBUG
0337            vPrintf("REJOIN TIMER: %d\n",sAppData.i8RejoinTimer);
0338            #endif
0339
0340            u8Flag=READY;
0341        }break;
0342
0343        //If there is a leave by command request from a child
0344        case ORDER_CHILD_LEAVING:
0345        {
0346            NET_MhmeReqRsp_s    sMhmeReq;
0347            NET_MhmeSyncCfm_s   sMhmeSyncCfm;
0348            NET_Neighbor_s*     psNeighbor;
0349            uint8               i;
0350
0351            //Getting the neighbor list, the network address and the mac address
0352            sMhmeReq.u8Type=NET_MHME_REQ_GET;
0353            sMhmeReq.uParam.sReqGet.u8MeshIBAttribute=MESH_NEIGHBOR_LIST;
0354            vNetApiMhmeRequest(&sMhmeReq,&sMhmeSyncCfm);
0355            if(sMhmeSyncCfm.uParam.sCfmGet.eStatus!=SUCCESS) break;
0356            psNeighbor=(NET_Neighbor_s*)sMhmeSyncCfm.uParam.sCfmGet.psMibAttributeValue;
0357
0358            for(i=0;i<MAX_NEIGHBORS;i++)
0359            {
0360                if(psNeighbor[i].u8Relationship==CHILD)
0361                {
0362                    //Send an order to the child to leave the network
0363                    sMhmeReq.u8Type=NET_MHME_REQ_LEAVE;
0364                    sMhmeReq.uParam.sReqLeave.u8RemoveSelf=FALSE;
0365                    sMhmeReq.uParam.sReqLeave.u8RemoveChildren=FALSE;
0366
memcpy(&sMhmeReq.uParam.sReqLeave.sDeviceAddress,&psNeighbor[i].sExt,sizeof(MAC_ExtAddr_s));
0367                    vNetApiMhmeRequest(&sMhmeReq,&sMhmeSyncCfm);
0368                }
0369            }
0370
0371            u8Flag=READY;
0372        }break;
0373
0374        //If there is a leave by command request from the parent/coordinator
0375        case ORDER_PARENT_LEAVING:
0376        {
0377            NET_MhmeReqRsp_s    sMhmeReq;
```

```
0378            NET_MhmeSyncCfm_s   sMhmeSyncCfm;
0379            NET_Neighbor_s*     psNeighbor;
0380            uint8               i;
0381
0382            //Getting the neighbor list, the network address and the mac address
0383            sMhmeReq.u8Type=NET_MHME_REQ_GET;
0384            sMhmeReq.uParam.sReqGet.u8MeshIBAttribute=MESH_NEIGHBOR_LIST;
0385            vNetApiMhmeRequest(&sMhmeReq,&sMhmeSyncCfm);
0386            if(sMhmeSyncCfm.uParam.sCfmGet.eStatus!=SUCCESS) break;
0387            psNeighbor=(NET_Neighbor_s*)sMhmeSyncCfm.uParam.sCfmGet.psMibAttributeValue;
0388
0389            for(i=0;i<MAX_NEIGHBORS;i++)
0390            {
0391                if(psNeighbor[i].u8Relationship==CHILD)
0392                {
0393                    //Send an order to the child to leave the network
0394                    sMhmeReq.u8Type=NET_MHME_REQ_LEAVE;
0395                    sMhmeReq.uParam.sReqLeave.u8RemoveSelf=FALSE;
0396                    sMhmeReq.uParam.sReqLeave.u8RemoveChildren=TRUE;
0397
memcpy(&sMhmeReq.uParam.sReqLeave.sDeviceAddress,&psNeighbor[i].sExt,sizeof(MAC_ExtAddr_s));
0398                    vNetApiMhmeRequest(&sMhmeReq,&sMhmeSyncCfm);
0399                }
0400            }
0401
0402            u8Flag=READY;
0403        }break;
0404
0405        //If the device is waiting for leaving the network
0406        case WAITING_DCFM_LEAVE:
0407        {
0408        }break;
0409
0410
0411        //If the device is not performing any tasks
0412        case READY:
0413        {
0414        }break;
0415
0416        /** . . . **/
0417
0418        //If the flag has a state that is not recoginzed
0419        default:
0420        {
0421            #ifdef DEBUG
0422            vStackPrintf(__FILE__,__LINE__,"Unhandled device state.");
0423            #endif
0424        }
0425    }
0426
0427    //Deregister this function off the debugger
0428    #ifdef DEBUG
0429    vStackPopIdentifier();
0430    #endif
0431 }
0432
0433
0434
0435 void vAppProccessMessage(NET_MhmeDcfmInd_s* psMhmeDcfmInd)
0436 {
0437    //Register this function on the debugger
0438    #ifdef DEBUG
0439    u8StackPushIdentifier("vAppProccessMessage",strlen("vAppProccessMessage"),FALSE);
0440    #endif
0441
```

```
0442      switch(psMhmeDcfmInd->u8Type)
0443      {
0444          case NET_MHME_DCFM_DISCOVER:
0445          {
0446              #ifdef DEBUG
0447              vStackPrintf(__FILE__,__LINE__,"The discover request has been done");
0448              #endif
0449
0450              //If there aren't networks to join
0451              if(psMhmeDcfmInd->uParam.sCfmDiscover.u8Status!=SUCCESS)
0452              {
0453                  #ifdef DEBUG
0454                  vStackPrintf(__FILE__,__LINE__,"The discover has failed. Trying again. . ." );
0455                  #endif
0456                  u8Flag=BEGIN;
0457              }
0458
0459              //If there are networks to join
0460              else
0461              {
0462                  uint16 i,u32numberOfDiscoveredNetworks=psMhmeDcfmInd-
>uParam.sCfmDiscover.u8NetworkCount;
0463                  bool bPanMatch=FALSE;
0464
0465                  //Choose the network which has the same panId the device needs to join
0466                  for(i=0;i<u32numberOfDiscoveredNetworks;i++)
0467                      if(psMhmeDcfmInd->uParam.sCfmDiscover.psMeshDescriptorList[i].u16PanId==PAN_ID)
0468                      {
0469                          sChosenDescriptor=psMhmeDcfmInd->uParam.sCfmDiscover.psMeshDescriptorList[i];
0470                          bPanMatch=TRUE;
0471                      }
0472
0473                  //If there was a match continue. Otherwise start all over again
0474                  if(bPanMatch)
0475                      u8Flag=DISCOVERED;
0476                  else
0477                  {
0478                      #ifdef DEBUG
0479                      vStackPrintf(__FILE__,__LINE__,"The required panId was not found between the
discovered networks. Trying again. . ." );
0480                      #endif
0481                      u8Flag=BEGIN;
0482                  }
0483              }
0484          }break;
0485
0486          case NET_MHME_DCFM_JOIN:
0487          {
0488              switch(psMhmeDcfmInd->uParam.sCfmJoin.u8Status)
0489              {
0490                  case SUCCESS:
0491                  {
0492                      #ifdef DEBUG
0493                      vStackPrintf(__FILE__,__LINE__,"The join request has been done");
0494                      #endif
0495
0496                      //Start a data timer
0497                      sAppData.i8DataTimer=VirtualTimer_i8New(10*sAppData.u8DataTime); //x * (10 * 100)ms
0498                      VirtualTimer_bReset(sAppData.i8DataTimer);
0499                      VirtualTimer_bCount(sAppData.i8DataTimer);
0500                      #ifdef DEBUG
0501                      vPrintf("DATA TIMER: %d\n",sAppData.i8DataTimer);
0502                      #endif
0503
0504                      //Start a leaving timer
```

```
0505                    sAppData.i8LeaveTimer=VirtualTimer_i8New(10*sAppData.u8LeaveTime); //x * (10 *
100)ms
0506                    VirtualTimer_bReset(sAppData.i8LeaveTimer);
0507                    VirtualTimer_bCount(sAppData.i8LeaveTimer);
0508
0509                    #ifdef DEBUG
0510                    vPrintf("LEAVE TIMER: %d\n",sAppData.i8LeaveTimer);
0511                    #endif
0512
0513                    if(sAppData.i8OrderLeaveChildTimer==-1)
0514                    {
0515                        //Start a child order leaving timer
0516
sAppData.i8OrderLeaveChildTimer=VirtualTimer_i8New(10*sAppData.u8OrderLeaveChildTime); //x * (10 * 100)ms
0517                        VirtualTimer_bReset(sAppData.i8OrderLeaveChildTimer);
0518                        VirtualTimer_bCount(sAppData.i8OrderLeaveChildTimer);
0519                    }
0520
0521                    u8Flag=READY;
0522                }break;
0523                case DCFM_JOIN:
0524                {
0525                    #ifdef DEBUG
0526                    vStackPrintf(__FILE__,__LINE__,"The join request has been done (No ADDR)");
0527                    #endif
0528                    u8Flag=START_DEVICE;
0529                }break;
0530                default:
0531                {
0532                    #ifdef DEBUG
0533                    vStackPrintf(__FILE__,__LINE__,"The join request has failed. Trying again. . .");
0534                    #endif
0535                    u8Flag=BEGIN;
0536                }
0537            }
0538        }break;
0539
0540        case NET_MHME_IND_JOIN:
0541        {
0542            #ifdef DEBUG
0543            vStackPrintf(__FILE__,__LINE__,"Received Join Indication");
0544            #endif
0545
0546            if(COORDINATOR)
0547            {
0548                //Start a parent/coordinator order leaving timer
0549                if(sAppData.i8OrderLeaveParentTimer==-1)
0550                {
0551
sAppData.i8OrderLeaveParentTimer=VirtualTimer_i8New(10*sAppData.u8OrderLeaveParentTime); //x * (10 * 100)ms
0552                    VirtualTimer_bReset(sAppData.i8OrderLeaveParentTimer);
0553                    VirtualTimer_bCount(sAppData.i8OrderLeaveParentTimer);
0554                }
0555            }
0556
0557            u8Flag=READY;
0558        }break;
0559
0560        case NET_MHME_IND_TIMER:
0561        {
0562            #ifdef DEBUG
0563            vStackPrintf(__FILE__,__LINE__,"Received a timer expired indication");
0564            #endif
0565
0566            if(psMhmeDcfmInd->uParam.sIndTmer.u8TriggeredTimer==sAppData.i8DataTimer)
```

```
0567            {
0568
0569                //Reset the timer (data will be sent again)
0570                VirtualTimer_bReset(sAppData.i8DataTimer);
0571                VirtualTimer_bCount(sAppData.i8DataTimer);
0572
0573                //Send the data
0574                u8Flag=DATA_READY;
0575
0576                #ifdef DEBUG
0577                vStackPrintf(__FILE__,__LINE__,"Ready to send data.");
0578                #endif
0579
0580            }
0581            else if(psMhmeDcfmInd->uParam.sIndTmer.u8TriggeredTimer==sAppData.i8LeaveTimer)
0582            {
0583                //Stop the leaving and the data timers
0584                VirtualTimer_bStop(sAppData.i8LeaveTimer);
0585                VirtualTimer_bDelete(sAppData.i8LeaveTimer);
0586                sAppData.i8LeaveTimer=-1;
0587                VirtualTimer_bStop(sAppData.i8DataTimer);
0588                VirtualTimer_bDelete(sAppData.i8DataTimer);
0589                sAppData.i8DataTimer=-1;
0590                VirtualTimer_bStop(sAppData.i8OrderLeaveChildTimer);
0591                VirtualTimer_bDelete(sAppData.i8OrderLeaveChildTimer);
0592                sAppData.i8OrderLeaveChildTimer=-1;
0593
0594                //Leaving the network
0595                u8Flag=LEAVING;
0596
0597                #ifdef DEBUG
0598                vStackPrintf(__FILE__,__LINE__,"Ready to leave.");
0599                #endif
0600
0601            }
0602            else if(psMhmeDcfmInd->uParam.sIndTmer.u8TriggeredTimer==sAppData.i8OrderLeaveChildTimer)
0603            {
0604                #ifdef DEBUG
0605                vStackPrintf(__FILE__,__LINE__,"Order to leave.");
0606                #endif
0607
0608                //Clean all its children
0609                u8Flag=ORDER_CHILD_LEAVING;
0610            }
0611            else if(psMhmeDcfmInd->uParam.sIndTmer.u8TriggeredTimer==sAppData.i8OrderLeaveParentTimer)
0612            {
0613                #ifdef DEBUG
0614                vStackPrintf(__FILE__,__LINE__,"Order to leave.");
0615                #endif
0616
0617                //Clean all its children
0618                u8Flag=ORDER_PARENT_LEAVING;
0619            }
0620            else if(psMhmeDcfmInd->uParam.sIndTmer.u8TriggeredTimer==sAppData.i8RejoinTimer)
0621            {
0622                //Stop the rejoin timer
0623                VirtualTimer_bStop(sAppData.i8RejoinTimer);
0624                VirtualTimer_bDelete(sAppData.i8RejoinTimer);
0625                sAppData.i8RejoinTimer=-1;
0626
0627                //Start all over again
0628                u8Flag=BEGIN;
0629
0630                #ifdef DEBUG
0631                vStackPrintf(__FILE__,__LINE__,"Ready to rejoin.");
```

```
0632                #endif
0633            }
0634            else
0635            {
0636                #ifdef DEBUG
0637                vStackPrintf(__FILE__,__LINE__,"MHME send an unknown timer to the application.");
0638                #endif
0639            }
0640
0641        }break;
0642
0643        case NET_MHME_DCFM_LEAVE:
0644        {
0645            #ifdef DEBUG
0646            vStackPrintf(__FILE__,__LINE__,"Confimration that a Device has left.");
0647            vPrintf("Its mac is %x %x\n",psMhmeDcfmInd-
>uParam.sCfmLeave.sDeviceAddress.u32H,psMhmeDcfmInd->uParam.sCfmLeave.sDeviceAddress.u32L);
0648            #endif
0649            if(u8Flag==WAITING_DCFM_LEAVE)
0650                u8Flag=BEGIN;
0651
0652            if(COORDINATOR)
0653            {
0654                VirtualTimer_bStop(sAppData.i8OrderLeaveParentTimer);
0655                VirtualTimer_bDelete(sAppData.i8OrderLeaveParentTimer);
0656                sAppData.i8OrderLeaveParentTimer=-1;
0657            }
0658            else
0659            {
0660                VirtualTimer_bStop(sAppData.i8OrderLeaveChildTimer);
0661                VirtualTimer_bDelete(sAppData.i8OrderLeaveChildTimer);
0662                sAppData.i8OrderLeaveChildTimer=-1;
0663            }
0664        }break;
0665
0666        case NET_MHME_IND_LEAVE:
0667        {
0668            #ifdef DEBUG
0669            vStackPrintf(__FILE__,__LINE__,"Indication that a device Is leaving.");
0670            vPrintf("The address of the device: ");
0671            (psMhmeDcfmInd->uParam.sIndLeave.sDeviceAddress.u32H==0L)&&(psMhmeDcfmInd-
>uParam.sIndLeave.sDeviceAddress.u32L==0L)?
0672                vPrintf("ME!!!!\n"):
0673                vPrintf("%x %x\n",psMhmeDcfmInd->uParam.sIndLeave.sDeviceAddress.u32H,psMhmeDcfmInd-
>uParam.sIndLeave.sDeviceAddress.u32L);
0674            #endif
0675
0676            if((psMhmeDcfmInd->uParam.sIndLeave.sDeviceAddress.u32H==0L)&&(psMhmeDcfmInd-
>uParam.sIndLeave.sDeviceAddress.u32L==0L))
0677                u8Flag=WAITING_DCFM_LEAVE;
0678        }break;
0679        case NET_MHME_IND_LEFT:
0680        {
0681            #ifdef DEBUG
0682            vStackPrintf(__FILE__,__LINE__,"Indication that a device has left.");
0683            vPrintf("The address of the device: ");
0684            (psMhmeDcfmInd->uParam.sIndLeave.sDeviceAddress.u32H==0L)&&(psMhmeDcfmInd-
>uParam.sIndLeave.sDeviceAddress.u32L==0L)?
0685                vPrintf("ME!!!!\n"):
0686                vPrintf("%x %x\n",psMhmeDcfmInd->uParam.sIndLeave.sDeviceAddress.u32H,psMhmeDcfmInd-
>uParam.sIndLeave.sDeviceAddress.u32L);
0687            #endif
0688
0689            if((psMhmeDcfmInd->uParam.sIndLeave.sDeviceAddress.u32H==0L)&&(psMhmeDcfmInd-
>uParam.sIndLeave.sDeviceAddress.u32L==0L))
```

```
0690                u8Flag=BEGIN;
0691        }
0692
0693        /** . . . **/
0694
0695        //If the message is not recoginzed
0696        default:
0697        {
0698            #ifdef DEBUG
0699            vStackPrintf(__FILE__,__LINE__,"Unhandle confirm/indication");
0700            #endif
0701        }
0702    }
0703
0704    //Deregister this function off the debugger
0705    #ifdef DEBUG
0706    vStackPopIdentifier();
0707    #endif
0708 }
0709
0710
0711
0712 PRIVATE void vAppProcessData(NET_MeshDcfmInd_s* psMeshDcfmInd)
0713 {
0714
0715    //Register this function on the debugger
0716    #ifdef DEBUG
0717    u8StackPushIdentifier("vAppProccessData",strlen("vAppProccessData"),FALSE);
0718    #endif
0719
0720    switch(psMeshDcfmInd->u8Type)
0721    {
0722        case NET_MESH_IND_DATA:
0723        {
0724            #ifdef DEBUG
0725            vPrintf("\n\n\n\nIncoming data message from: %d\n",psMeshDcfmInd-
>uParam.sMeshIndData.uSrcAddr.u16Short);
0726            vPrintf("Message Received: %s\n\n\n\n",psMeshDcfmInd->uParam.sMeshIndData.au8Mhsdu);
0727            #endif
0728        }break;
0729        case NET_MESH_DCFM_DATA:
0730        {
0731            #ifdef DEBUG
0732            vPrintf("Data sent with handle: %d\n",psMeshDcfmInd->uParam.sMeshCfmData.u8MhsduHandle);
0733            #endif
0734        }break;
0735        default:
0736        {
0737
0738        }break;
0739
0740    }
0741
0742    //Deregister this function off the debugger
0743    #ifdef DEBUG
0744    vStackPopIdentifier();
0745    #endif
0746 }
0747
0748 void initErrors(void);
0749
0750 PUBLIC void AppColdStart(void)
0751 {
0752    NET_MhmeDcfmInd_s sMhmeDcfmInd;
0753    NET_MeshDcfmInd_s sMeshDcfmInd;
```

```
0754
0755     //Initialize Integrated Peripherals API
0756     u32AHI_Init();
0757
0758     //Initializing the debuger at UART0 (default port)
0759     #ifdef DEBUG
0760     vStackStart();
0761     #endif
0762
0763     //Register this function on the debugger
0764     #ifdef DEBUG
0765     u8StackPushIdentifier("AppColdStart",strlen("AppColdStart"),TRUE);
0766     vStackPrintf(__FILE__,__LINE__,"Running!");
0767     #endif
0768
0769     //Initializations
0770     vInitMeshStack();
0771     vAppInit();
0772
0773     //Hard debug
0774     initErrors();
0775
0776     while(1)
0777     {
0778         //Application management - according to flags
0779         vAppPerformTasks();
0780
0781         //Get messages from queue
0782         if(u16MhmeGetMessage(&sMhmeDcfmInd))
0783         {
0784             //We are able to alter the flags
0785             vAppProccessMessage(&sMhmeDcfmInd);
0786         }
0787
0788         //Get message from Mesh sublayer
0789         if(u16MeshGetMessage(&sMeshDcfmInd))
0790         {
0791             //Process incoming data
0792             vAppProcessData(&sMeshDcfmInd);
0793         }
0794
0795     }
0796
0797     //Deregister this function off the debugger
0798     #ifdef DEBUG
0799     vStackPopIdentifier();
0800     #endif
0801 }
0802
0803 PUBLIC void AppWarmStart(void)
0804 {
0805     AppColdStart();
0806 }
0807 //#define BUS_ERROR * ((volatile uint32 *) (0x4000008))
0808 //#define UNALIGNED_ACCESS *((volatile uint32 *)(0x4000018))
0809 //#define ILLEGAL_INSTRUCTION *((volatile uint32 *)(0x400001C))
0810 // event handler function prototypes
0811 PUBLIC void vBusErrorhandler(void);
0812 PUBLIC void vUnalignedAccessHandler(void);
0813 PUBLIC void vIllegalInstructionHandler(void);
0814 //BUS_ERROR = (uint32) vBusErrorhandler;
0815 //UNALIGNED_ACCESS = (uint32) vUnalignedAccessHandler;
0816 //ILLEGAL_INSTRUCTION = (uint32) vIllegalInstructionHandler;
0817
0818 volatile uint32 *bep=(volatile uint32 *)0x4000008;
```

```
0819 volatile uint32 *uap=(volatile uint32 *)0x4000018;
0820 volatile uint32 *iip=(volatile uint32 *)0x400001C;
0821
0822 void initErrors(void)
0823 {
0824     *bep=(uint32) vBusErrorhandler;
0825     //*uap=(uint32) vUnalignedAccessHandler;
0826     *iip=(uint32) vIllegalInstructionHandler;
0827
0828 }
0829
0830 PUBLIC void vBusErrorhandler(void)
0831 {
0832 #ifdef DEBUG
0833 volatile uint32 u32BusyWait = 1600000;
0834 // log the exception
0835 vPrintf("\nBus Error Exception");
0836 // wait for the UART write to complete
0837 while(u32BusyWait--){}
0838 #endif
0839 vAHI_SwReset ();
0840 }
0841
0842 /*PUBLIC void vUnalignedAccessHandler (void)
0843 {
0844 #ifdef DEBUG
0845 volatile uint32 u32BusyWait = 1600000;
0846 // log the exception
0847 vPrintf("\nUnaligned Error Exception");
0848 // wait for the UART write to complete
0849 while(u32BusyWait--){}
0850 #endif
0851 vAHI_SwReset ();
0852 }*/
0853 PUBLIC void vIllegalInstructionHandler(void)
0854 {
0855 #ifdef DEBUG
0856 volatile uint32 u32BusyWait = 1600000;
0857 // log the exception
0858 vPrintf("\nIllegal Instruction Error Exception");
0859 // wait for the UART write to complete
0860 while(u32BusyWait--){}
0861 #endif
0862 vAHI_SwReset ();
0863 }
0864
0865 /** @} */
0866
```

II. App.h

```
0001 #ifndef _APP_H
0002 #define _APP_H
0003
0004 #define COORDINATOR FALSE
0005 typedef enum
0006 {
0007     BEGIN,
0008     NETWORK_STARTED,
0009     START_NETWORK_FAILED,
0010     DISCOVERING,
0011     DISCOVERED,
0012     JOINING,
0013     READY,
```

```
0014     START_DEVICE,
0015     DATA_READY,
0016     LEAVING,
0017     ORDER_CHILD_LEAVING,
0018     ORDER_PARENT_LEAVING,
0019     WAITING_DCFM_LEAVE
0020
0021 }APP_flagType_e;
0022
0023 typedef struct
0024 {
0025     uint8   u8AppHandle;
0026     int8    i8DataTimer;
0027     uint8   u8DataTime;
0028     int8    i8LeaveTimer;
0029     uint8   u8LeaveTime;
0030     int8    i8RejoinTimer;
0031     uint8   u8RejoinTime;
0032     int8    i8OrderLeaveChildTimer;
0033     uint8   u8OrderLeaveChildTime;
0034     int8    i8OrderLeaveParentTimer;
0035     uint8   u8OrderLeaveParentTime;
0036     uint8   u8Pad;
0037 }
0038 NET_AppData_s;
0039
0040 NET_MeshDescriptor_s sChosenDescriptor;
0041
0042 volatile uint8 u8Flag=BEGIN;
0043
0044 #endif //_APP_H
```

III. Commandframes.c

```
0001 #include "CommandFrames.h"
0002 #include "DataFrames.h"
0003 #include "mhme.h"
0004 #include "MhmeServices.h"
0005 #include "VirtualTimer.h"
0006 #include "string.h"
0007
0008
0009 #ifdef DEBUG
0010 #include "debugger.h"
0011 #endif
0012
0013 #define SAME64ADDR(a,b) ((a.u32H==b.u32H)&&(a.u32L==b.u32L))
0014
0015 extern NET_MeshInfo_s sMeshInfo;
0016 extern NET_MeshData_s sMeshData;
0017
0018
0019 inline void vHdlCommSendChildrenNumberReport(NET_CommSyncCfm_s* psCommSyncCfm);
0020 inline void vHdlCommSendAddressAssignment(NET_CommSyncCfm_s* psCommSyncCfm);
0021 inline void vHdlCommSendHello(NET_CommSyncCfm_s* psCommSyncCfm);
0022 inline void vHdlCommSendLeave(NET_CommSyncCfm_s* psCommSyncCfm);
0023
0024
0025
0026
0027
```

```
0028 inline void vHdlCommRecvChildrenNumberReport(NET_CommDcfmInd_s* psCommDcfmInd,NET_MeshDcfmInd_s*
psMeshDcfmInd);
0029 inline void vHdlCommRecvAddressAssignment(NET_CommDcfmInd_s* psCommDcfmInd,NET_MeshDcfmInd_s*
psMeshDcfmInd);
0030 inline void vHdlCommRecvHello(NET_CommDcfmInd_s* psCommDcfmInd,NET_MeshDcfmInd_s* psMeshDcfmInd);
0031 inline void vHdlCommRecvLeave(NET_CommDcfmInd_s* psCommDcfmInd,NET_MeshDcfmInd_s* psMeshDcfmInd);
0032
0033
0034
0035
0036 void vNetApiCommRequest(uint16 u16CommandFrameNumber,NET_CommSyncCfm_s* psCommSyncCfm)
0037 {
0038     //Register this function on the debugger
0039     #ifdef DEBUG
0040     u8StackPushIdentifier("vNetApiCommRequest",strlen("vNetApiCommRequest"),FALSE);
0041     #endif
0042
0043     //Switch through the several command frame types
0044     switch(u16CommandFrameNumber)
0045     {
0046         case FRAME_CHILDREN_NUMBER_REPORT:
0047         {
0048             vHdlCommSendChildrenNumberReport(psCommSyncCfm);
0049         }break;
0050         case FRAME_ADDRESS_ASSIGNMENT:
0051         {
0052             vHdlCommSendAddressAssignment(psCommSyncCfm);
0053         }break;
0054         case FRAME_HELLO:
0055         {
0056             vHdlCommSendHello(psCommSyncCfm);
0057         }break;
0058         case FRAME_LEAVE:
0059         {
0060             vHdlCommSendLeave(psCommSyncCfm);
0061         }break;
0062         default:
0063         {
0064             psCommSyncCfm->u8Type=FRAME_UNKNOWN;
0065
0066             #ifdef DEBUG
0067             vStackPrintf(__FILE__,__LINE__,"Unhadle Command Frame\n");
0068             #endif
0069         }
0070     }
0071
0072     //Deregister this function off the debugger
0073     #ifdef DEBUG
0074     vStackPopIdentifier();
0075     #endif
0076 }
0077
0078
0079
0080
0081 void vNetApiCommTranslate(NET_CommDcfmInd_s* psCommDcfmInd,NET_MeshDcfmInd_s* psMeshDcfmInd)
0082 {
0083
0084     //Register this function on the debugger
0085     #ifdef DEBUG
0086     u8StackPushIdentifier("vNetApiCommTranslate",strlen("vNetApiCommTranslate"),FALSE);
0087     #endif
0088
0089     //Switch through the several possible received command frame types
0090     switch(psMeshDcfmInd->uParam.sMeshIndData.au8Mhsdu[0])
```

```
0091     {
0092         case FRAME_CHILDREN_NUMBER_REPORT:
0093         {
0094             vHdlCommRecvChildrenNumberReport(psCommDcfmInd,psMeshDcfmInd);
0095         }break;
0096         case FRAME_ADDRESS_ASSIGNMENT:
0097         {
0098             vHdlCommRecvAddressAssignment(psCommDcfmInd,psMeshDcfmInd);
0099         }break;
0100         case FRAME_HELLO:
0101         {
0102             if(u8MhmeFlag==MHME_TOPOLOGY_DISCOVERY)
0103                 vHdlCommRecvHello(psCommDcfmInd,psMeshDcfmInd);
0104             else
0105                 psCommDcfmInd->u8Type=FRAME_INVALID;
0106         }break;
0107         case FRAME_LEAVE:
0108         {
0109             #ifdef DEBUG
0110             vStackPrintf(__FILE__,__LINE__,"FRAME LEAVE was detected\n");
0111             #endif
0112             vHdlCommRecvLeave(psCommDcfmInd,psMeshDcfmInd);
0113         }break;
0114         default:
0115         {
0116             psCommDcfmInd->u8Type=FRAME_UNKNOWN;
0117 
0118             #ifdef DEBUG
0119             vStackPrintf(__FILE__,__LINE__,"Unhadle Command Frame\n");
0120             #endif
0121         }
0122     }
0123 
0124     //Deregister this function off the debugger
0125     #ifdef DEBUG
0126     vStackPopIdentifier();
0127     #endif
0128 }
0129 
0130 
0131 
0132 
0133 /*************************************SEND FRAMES*****************************/
0134 
0135 inline void vHdlCommSendChildrenNumberReport(NET_CommSyncCfm_s* psCommSyncCfm)
0136 {
0137     NET_DataReqRsp_s            sDataReqRsp;
0138     NET_DataSyncCfm_s           sDataSyncCfm;
0139     NET_CommFrameChildReport_s  sCommFrameChildReport;
0140     uint8                       i;
0141 
0142     //Register this function on the debugger
0143     #ifdef DEBUG
0144 
u8StackPushIdentifier("vHdlCommSendChildrenNumberReport",strlen("vHdlCommSendChildrenNumberReport"),FALSE);
0145     #endif
0146 
0147     //Filling children number report structure
0148     sCommFrameChildReport.u16NumberOfDescendants=sMeshInfo.u8NbOfChildren;
0149     sCommFrameChildReport.u16NumberOfRequestedAddresses=sMeshData.u16NumberOfRequestedAddresses;
0150 
0151     //Sending children number report frame
0152     for(i=0;i<MAX_NEIGHBORS;i++)
0153     {
0154         if(sMeshInfo.psNeighborList[i].u8Relationship==PARENT)
```

```
0155        {
0156
0157            //Sending the children number report structure
0158            sDataReqRsp.u8Type=NET_DATA_IND_COMM;
0159            sDataReqRsp.u8MhsduHandle=(uint8)i16MhmeInsertMhsduHandle(FRAME_CHILDREN_NUMBER_REPORT);
0160            memcpy(&sDataReqRsp.sExt,&sMeshInfo.psNeighborList[i].sExt,sizeof(MAC_ExtAddr_s));
0161            sDataReqRsp.sDataFrameData.u16SrcPANId=sMeshInfo.u16PanId;
0162            sDataReqRsp.sDataFrameData.u8SrcAddrMode=3;
0163            sDataReqRsp.sDataFrameData.u8DstAddrMode=3;
0164
memcpy(&sDataReqRsp.sDataFrameData.uSrcAddr.sExt,&sMeshInfo.sAddressMapping.sExtAddr,sizeof(MAC_ExtAddr_s));
0165
memcpy(&sDataReqRsp.sDataFrameData.uDstAddr.sExt,&sMeshInfo.psNeighborList[i].sExt,sizeof(MAC_ExtAddr_s));
0166            sDataReqRsp.sDataFrameData.u8MhsduLength=sizeof(NET_CommFrameChildReport_s)+1;
0167            sDataReqRsp.sDataFrameData.au8Mhsdu[0]=FRAME_CHILDREN_NUMBER_REPORT;
0168
memcpy(&sDataReqRsp.sDataFrameData.au8Mhsdu[1],&sCommFrameChildReport,sizeof(NET_CommFrameChildReport_s));
0169            sDataReqRsp.sDataFrameData.u8AckTransmission=TRUE;
0170            sDataReqRsp.sDataFrameData.u8McstTransmission=FALSE;
0171            sDataReqRsp.sDataFrameData.u8BcstTransmission=FALSE;
0172            sDataReqRsp.sDataFrameData.u8ReliableBcst=FALSE;
0173
0174            vNetApiDataRequest(&sDataReqRsp,&sDataSyncCfm);
0175            break;
0176        }
0177    }
0178
0179    //Filing the syncronous confirm
0180    psCommSyncCfm->u8Type=FRAME_CHILDREN_NUMBER_REPORT;
0181    psCommSyncCfm->u8Status=SUCCESS;
0182
0183    //Deregister this function off the debugger
0184    #ifdef DEBUG
0185    vStackPopIdentifier();
0186    #endif
0187 }
0188 inline void vHdlCommSendAddressAssignment(NET_CommSyncCfm_s* psCommSyncCfm)
0189 {
0190    NET_DataReqRsp_s                sDataReqRsp;
0191    NET_DataSyncCfm_s               sDataSyncCfm;
0192    NET_CommFrameAddrAssignment_s   sCommFrameAddrAssignment;
0193
0194    uint8                           i;
0195
0196    //Register this function on the debugger
0197    #ifdef DEBUG
0198
u8StackPushIdentifier("vHdlCommSendAddressAssignment",strlen("vHdlCommSendAddressAssignment"),FALSE);
0199    #endif
0200
0201    //For each child/orphan, send a command frame with the block of addresses
0202    for(i=0;i<MAX_NEIGHBORS;i++)
0203    {
0204        if(sMeshInfo.psNeighborList[i].u8Relationship==CHILD)
0205        {
0206            if(READBITMAP(sMeshData.sBitMapAddressAssignment,i))
0207            {
0208                //Filling address assignment structure
0209
sCommFrameAddrAssignment.u16BeginningAddress=sMeshInfo.psNeighborList[i].u16BeginningAddress;
0210                sCommFrameAddrAssignment.u16EndingAddress=sMeshInfo.psNeighborList[i].u16EndingAddress;
0211                sCommFrameAddrAssignment.u8ParentTreeLevel=sMeshInfo.u8TreeLevel;
0212
0213
0214                //Sending the address assignment structure
```

```
0215                sDataReqRsp.u8Type=NET_DATA_IND_COMM;
0216                sDataReqRsp.u8MhsduHandle=(uint8)i16MhmeInsertMhsduHandle(FRAME_ADDRESS_ASSIGNMENT);
0217                memcpy(&sDataReqRsp.sExt,&sMeshInfo.psNeighborList[i].sExt,sizeof(MAC_ExtAddr_s));
0218                sDataReqRsp.sDataFrameData.u16SrcPANId=sMeshInfo.u16PanId;
0219                sDataReqRsp.sDataFrameData.u8SrcAddrMode=3;
0220                sDataReqRsp.sDataFrameData.u8DstAddrMode=3;
0221
memcpy(&sDataReqRsp.sDataFrameData.uSrcAddr.sExt,&sMeshInfo.sAddressMapping.sExtAddr,sizeof(MAC_ExtAddr_s));
0222
memcpy(&sDataReqRsp.sDataFrameData.uDstAddr.sExt,&sMeshInfo.psNeighborList[i].sExt,sizeof(MAC_ExtAddr_s));
0223                sDataReqRsp.sDataFrameData.u8MhsduLength=sizeof(NET_CommFrameAddrAssignment_s)+1;
0224                sDataReqRsp.sDataFrameData.au8Mhsdu[0]=FRAME_ADDRESS_ASSIGNMENT;
0225
memcpy(&sDataReqRsp.sDataFrameData.au8Mhsdu[1],&sCommFrameAddrAssignment,sizeof(NET_CommFrameAddrAssignment_s
));
0226                sDataReqRsp.sDataFrameData.u8AckTransmission=TRUE;
0227                sDataReqRsp.sDataFrameData.u8McstTransmission=FALSE;
0228                sDataReqRsp.sDataFrameData.u8BcstTransmission=FALSE;
0229                sDataReqRsp.sDataFrameData.u8ReliableBcst=FALSE;
0230
0231                #ifdef DEBUG
0232                vStackPrintf(__FILE__,__LINE__,"Sending address assignment frames");
0233                vPrintf("Beginning Address:%x\n",sCommFrameAddrAssignment.u16BeginningAddress);
0234                vPrintf("Ending Address:%x\n",sCommFrameAddrAssignment.u16EndingAddress);
0235                vPrintf("Tree Level:%d\n",sCommFrameAddrAssignment.u8ParentTreeLevel);
0236                #endif
0237
0238                vNetApiDataRequest(&sDataReqRsp,&sDataSyncCfm);
0239
0240                if(READBITMAP(sMeshData.sBitMapNoNew,i)) UNPLACEBITMAP(sMeshData.sBitMapNoNew,i);
0241                UNPLACEBITMAP(sMeshData.sBitMapAddressAssignment,i);
0242            }break;
0243        }
0244    }
0245
0246    //Filing the syncronous confirm
0247    psCommSyncCfm->u8Type=FRAME_ADDRESS_ASSIGNMENT;
0248    psCommSyncCfm->u8Status=SUCCESS;
0249
0250    //Deregister this function off the debugger
0251    #ifdef DEBUG
0252    vStackPopIdentifier();
0253    #endif
0254 }
0255 inline void vHdlCommSendHello(NET_CommSyncCfm_s* psCommSyncCfm)
0256 {
0257     NET_DataReqRsp_s            sDataReqRsp;
0258     NET_DataSyncCfm_s           sDataSyncCfm;
0259     NET_CommFrameHello_s        sCommFrameHello;
0260     uint8                       i,j;
0261
0262     //Register this function on the debugger
0263     #ifdef DEBUG
0264     u8StackPushIdentifier("vHdlCommSendHello",strlen("vHdlCommSendHello"),FALSE);
0265     #endif
0266
0267     //Filling hello structure
0268     sCommFrameHello.u8TTL=sMeshInfo.u8TtlOfHello;
0269     sCommFrameHello.u16BeginningAddress=sMeshInfo.u16NetworkAddress;
0270     sCommFrameHello.u16EndingAddress=sMeshData.u16EndingAddress;
0271     sCommFrameHello.u8TreeLevel=sMeshInfo.u8TreeLevel;
0272     sCommFrameHello.u8HelloControl=0x040|((u8MhmeFlag==MHME_LEAVING)?0x080:0x000); //No broadcast
0273
0274     sCommFrameHello.u8NumberOfOneHopNeighbors=0;
0275     for(i=0,j=0;i<MAX_NEIGHBORS;i++)
```

```
0276     {
0277
if((sMeshInfo.psNeighborList[i].u8Relationship!=NO_RELATIONSHIP)&&(sMeshInfo.psNeighborList[i].u8NumberOfOps=
=1))
0278         {
0279             sCommFrameHello.u8NumberOfOneHopNeighbors++;
0280
sCommFrameHello.au16AddressesOfOneHopNeighbors[j++]=sMeshInfo.psNeighborList[i].u16BeginningAddress;
0281         }
0282     }
0283     sCommFrameHello.u8NumberOfMulticastGroups=0;
0284     sCommFrameHello.u8NbOfHello=1+sMeshData.u8NbOfHello++;
0285
0286     //Sending the hello frame
0287     sDataReqRsp.u8Type=NET_DATA_IND_COMM;
0288     sDataReqRsp.u8MhsduHandle=(uint8)i16MhmeInsertMhsduHandle(FRAME_HELLO);
0289     //No MAC destination is specified
0290     sDataReqRsp.sDataFrameData.u16SrcPANId=sMeshInfo.u16PanId;
0291     sDataReqRsp.sDataFrameData.u8SrcAddrMode=2;
0292     sDataReqRsp.sDataFrameData.u8DstAddrMode=2;
0293     sDataReqRsp.sDataFrameData.uSrcAddr.u16Short=sMeshInfo.u16NetworkAddress;
0294     sDataReqRsp.sDataFrameData.uDstAddr.u16Short=0xffff;
0295     sDataReqRsp.sDataFrameData.u8MhsduLength=sizeof(NET_CommFrameHello_s)+1;
0296     sDataReqRsp.sDataFrameData.au8Mhsdu[0]=FRAME_HELLO;
0297     memcpy(&sDataReqRsp.sDataFrameData.au8Mhsdu[1],&sCommFrameHello,sizeof(NET_CommFrameHello_s));
0298     sDataReqRsp.sDataFrameData.u8AckTransmission=FALSE;
0299     sDataReqRsp.sDataFrameData.u8McstTransmission=FALSE;
0300     sDataReqRsp.sDataFrameData.u8BcstTransmission=TRUE;
0301     sDataReqRsp.sDataFrameData.u8ReliableBcst=FALSE;
0302
0303     vNetApiDataRequest(&sDataReqRsp,&sDataSyncCfm);
0304
0305     //Filing the syncronous confirm
0306     psCommSyncCfm->u8Type=FRAME_HELLO;
0307     psCommSyncCfm->u8Status=SUCCESS;
0308
0309     //Deregister this function off the debugger
0310     #ifdef DEBUG
0311     vStackPopIdentifier();
0312     #endif
0313 }
0314 inline void vHdlCommSendLeave(NET_CommSyncCfm_s* psCommSyncCfm)
0315 {
0316      NET_DataReqRsp_s            sDataReqRsp;
0317     NET_DataSyncCfm_s           sDataSyncCfm;
0318     NET_CommFrameLeave_s        sCommFrameLeave;
0319     uint8                       i;
0320
0321     //Register this function on the debugger
0322     #ifdef DEBUG
0323     u8StackPushIdentifier("vHdlCommSendLeave",strlen("vHdlCommSendLeave"),FALSE);
0324     #endif
0325
0326     //Sending leave commands for all the children
0327     for(i=0;i<MAX_NEIGHBORS;i++)
0328     {
0329         if(READBITMAP(sMeshData.sBitMapLeave,i))
0330         {
0331             //Filling leave structure
0332             sCommFrameLeave.u8LeaveControl=sMeshInfo.psNeighborList[i].bRemoveChildren?0x080:0x000;
0333
0334             //Sending leave frame
0335             sDataReqRsp.u8Type=NET_DATA_IND_COMM;
0336             if(sMeshInfo.psNeighborList[i].i8MsduHandleLeave==-1)
0337                 sMeshInfo.psNeighborList[i].i8MsduHandleLeave=i16MhmeInsertMhsduHandle(FRAME_LEAVE);
```

```
0338            sDataReqRsp.u8MhsduHandle=sMeshInfo.psNeighborList[i].i8MsduHandleLeave;
0339            memcpy(&sDataReqRsp.sExt,&sMeshInfo.psNeighborList[i].sExt,sizeof(MAC_ExtAddr_s));
0340            sDataReqRsp.sDataFrameData.u16SrcPANId=sMeshInfo.u16PanId;
0341            sDataReqRsp.sDataFrameData.u8SrcAddrMode=2;
0342            sDataReqRsp.sDataFrameData.u8DstAddrMode=2;
0343            sDataReqRsp.sDataFrameData.uSrcAddr.u16Short=sMeshInfo.u16NetworkAddress;
0344
sDataReqRsp.sDataFrameData.uDstAddr.u16Short=sMeshInfo.psNeighborList[i].u16BeginningAddress;
0345            sDataReqRsp.sDataFrameData.u8MhsduLength=sizeof(NET_CommFrameLeave_s)+1;
0346            sDataReqRsp.sDataFrameData.au8Mhsdu[0]=FRAME_LEAVE;
0347
memcpy(&sDataReqRsp.sDataFrameData.au8Mhsdu[1],&sCommFrameLeave,sizeof(NET_CommFrameLeave_s));
0348            sDataReqRsp.sDataFrameData.u8AckTransmission=TRUE;
0349            sDataReqRsp.sDataFrameData.u8McstTransmission=FALSE;
0350            sDataReqRsp.sDataFrameData.u8BcstTransmission=FALSE;
0351            sDataReqRsp.sDataFrameData.u8ReliableBcst=FALSE;
0352            vNetApiDataRequest(&sDataReqRsp,&sDataSyncCfm);
0353        }
0354    }
0355
0356    //Filing the syncronous confirm
0357    psCommSyncCfm->u8Type=FRAME_LEAVE;
0358    psCommSyncCfm->u8Status=SUCCESS;
0359
0360    //Deregister this function off the debugger
0361    #ifdef DEBUG
0362    vStackPopIdentifier();
0363    #endif
0364 }
0365
0366
0367
0368 /*********************************RECEIVING FRAMES*************************/
0369
0370 inline void vHdlCommRecvChildrenNumberReport(NET_CommDcfmInd_s* psCommDcfmInd,NET_MeshDcfmInd_s*
psMeshDcfmInd)
0371 {
0372    NET_CommFrameChildReport_s  sCommFrameChildReport;
0373    uint8                       i;
0374
0375    //Register this function on the debugger
0376    #ifdef DEBUG
0377
u8StackPushIdentifier("vHdlCommRecvChildrenNumberReport",strlen("vHdlCommRecvChildrenNumberReport"),FALSE);
0378    #endif
0379
0380    //Getting the children number report frame
0381    memcpy(&sCommFrameChildReport,&psMeshDcfmInd-
>uParam.sMeshIndData.au8Mhsdu[1],sizeof(NET_CommFrameChildReport_s));
0382
0383    //Mark the child for processing
0384    for(i=0;i<MAX_NEIGHBORS;i++)
0385    {
0386        if(sMeshInfo.psNeighborList[i].u8Relationship==CHILD)
0387        {
0388            if(sMeshInfo.psNeighborList[i].u16BeginningAddress==0xfffe)
PLACEBITMAP(sMeshData.sBitMapNew,i);
0389            else if(!READBITMAP(sMeshData.sBitMapRejoin,i)) PLACEBITMAP(sMeshData.sBitMapUpdate,i);
0390
sMeshInfo.psNeighborList[i].u16RequestedAddresses=sCommFrameChildReport.u16NumberOfRequestedAddresses;
0391        }
0392    }
0393
0394    #ifdef DEBUG
0395    vStackPrintf(__FILE__,__LINE__,"Printing Children Report.");
```

```
0396
vPrintf("u16NumberOfDescendants=%d\nu16NumberOfRequestedAddresses=%d\n",sCommFrameChildReport.u16NumberOfDesc
endants,
0397
sCommFrameChildReport.u16NumberOfRequestedAddresses);
0398     #endif
0399
0400     //Filing the defered confirm
0401     psCommDcfmInd->u8Type=FRAME_CHILDREN_NUMBER_REPORT;
0402     psCommDcfmInd->u8Status=SUCCESS;
0403
0404     //Deregister this function off the debugger
0405     #ifdef DEBUG
0406     vStackPopIdentifier();
0407     #endif
0408 }
0409 inline void vHdlCommRecvAddressAssignment(NET_CommDcfmInd_s* psCommDcfmInd,NET_MeshDcfmInd_s*
psMeshDcfmInd)
0410 {
0411     NET_CommFrameAddrAssignment_s  sCommFrameAddrAssignment;
0412     bool                           bAddressesChanged=FALSE;
0413     uint16                         u16Counter;
0414     uint8                          i;
0415
0416     //Register this function on the debugger
0417     #ifdef DEBUG
0418
u8StackPushIdentifier("vHdlCommRecvAddressAssignment",strlen("vHdlCommRecvAddressAssignment"),FALSE);
0419     #endif
0420
0421     //Getting the address assignment frame
0422     memcpy(&sCommFrameAddrAssignment,&psMeshDcfmInd-
>uParam.sMeshIndData.au8Mhsdu[1],sizeof(NET_CommFrameAddrAssignment_s));
0423
0424     //Processing received frame
0425     if(sCommFrameAddrAssignment.u16BeginningAddress==0xfffe)
0426         //Prepair to leave
0427         sMeshInfo.u16NetworkAddress=0xfffe;
0428     else
0429     {
0430         //Find if the addresses have changed
0431         bAddressesChanged =
(sMeshInfo.u16NetworkAddress!=sCommFrameAddrAssignment.u16BeginningAddress)||
0432                             (sMeshData.u16EndingAddress!=sCommFrameAddrAssignment.u16EndingAddress);
0433
0434         //Performing the updates
0435         sMeshInfo.u16NetworkAddress=sCommFrameAddrAssignment.u16BeginningAddress;
0436         sMeshInfo.u8TreeLevel=sCommFrameAddrAssignment.u8ParentTreeLevel+1;
0437         sMeshData.u16EndingAddress=sCommFrameAddrAssignment.u16EndingAddress;
0438
0439         //Renewing parent tree level
0440         for(i=0;i<MAX_NEIGHBORS;i++)
0441         {
0442             if(sMeshInfo.psNeighborList[i].u8Relationship==PARENT)
0443             {
0444                 sMeshInfo.psNeighborList[i].u8TreeLevel=sMeshInfo.u8TreeLevel-1;
0445                 break;
0446             }
0447         }
0448
0449         //Dealing with the waiting child report children devices
0450         for(i=0,u16Counter=0;i<MAX_NEIGHBORS;i++)
0451         {
0452             if(sMeshInfo.psNeighborList[i].u8Relationship==CHILD)
0453             {
```

```
0454                if(READBITMAP(sMeshData.sBitMapWaitingReport,i))
0455                {
0456                    u16Counter+=sMeshInfo.psNeighborList[i].u16RequestedAddresses;
0457                    if((sMeshData.u16EndingAddress-sMeshInfo.u16NetworkAddress)-
sMeshData.u16AllocatedAddresses<u16Counter) break;
0458                    else PLACEBITMAP(sMeshData.sBitMapNew,i);
0459                    UNPLACEBITMAP(sMeshData.sBitMapWaitingReport,i);
0460                }
0461            }
0462        }
0463
0464
0479        if(bAddressesChanged)
0480        {
0481            //Marking all the childs for updating
0482            /** ATTENTION **/
0483            memset(&sMeshData.sBitMapNew,0xff,sizeof(NET_BitMap_s));
0484
0485            //calculate block size
0486            if((sMeshData.u16BlockSize=((sMeshData.u16EndingAddress -
sMeshInfo.u16NetworkAddress)/MAX_NEIGHBORS))==0) sMeshData.u16BlockSize=1;
0487            sMeshData.u8LeftAddr =sMeshData.u16BlockSize%MAX_NEIGHBORS;
0488            sMeshData.u16FreeBlocks=(sMeshData.u16EndingAddress-
sMeshInfo.u16NetworkAddress)/sMeshData.u16BlockSize;
0489            sMeshData.u16AllocatedAddresses=0;
0490
0491            //Updating the free block list
0492            sMeshData.sFreeBlock[0].u16BeginningBlock=0;
0493            sMeshData.sFreeBlock[0].u16NumberOfBlocks=sMeshData.u16FreeBlocks;
0494            sMeshData.u16NumberOfFreeBlocksInList=1;
0495
0496            #ifdef DEBUG
0497            vPrintf("-----------------------------------\n");
0498            vPrintf("Block Size: %x\n",sMeshData.u16BlockSize);
0499            vPrintf("Free Blocks: %x\n",sMeshData.u16FreeBlocks);
0500            vPrintf("Left Addr: %x\n",sMeshData.u8LeftAddr);
0501            vPrintf("FBL Beginning Block: %d\n",sMeshData.sFreeBlock[0].u16BeginningBlock);
0502            vPrintf("FBL Number of blocks in free mega block:
%d\n",sMeshData.sFreeBlock[0].u16NumberOfBlocks);
0503            vPrintf("-----------------------------------\n");
0504            #endif
0505        }
0506
0507        #ifdef DEBUG
0508        vPrintf("Received from parent:\n");
0509        vPrintf("My 16 Network Address: %x\n",sMeshInfo.u16NetworkAddress);
0510        vPrintf("Ending Address: %x\n",sCommFrameAddrAssignment.u16EndingAddress);
0511        vPrintf("My Tree Level: %x\n",sMeshInfo.u8TreeLevel);
0512        #endif
0513
0514        //Filing the defered confirm
0515        psCommDcfmInd->u8Type=FRAME_ADDRESS_ASSIGNMENT;
0516        psCommDcfmInd->u8Status=SUCCESS;
0517    }
0518
0519    //Deregister this function off the debugger
0520    #ifdef DEBUG
0521    vStackPopIdentifier();
0522    #endif
0523 }
0524 inline void vHdlCommRecvHello(NET_CommDcfmInd_s* psCommDcfmInd,NET_MeshDcfmInd_s* psMeshDcfmInd)
0525 {
0526    NET_CommFrameHello_s    sCommFrameHello;
0527    uint16                  i,j,k,n;
0528    bool                    bUpdated=FALSE;
```

```
0529
0530      //Register this function on the debugger
0531      #ifdef DEBUG
0532      u8StackPushIdentifier("vHdlCommRecvHello",strlen("vHdlCommRecvHello"),FALSE);
0533      #endif
0534
0535      //Getting the address assignment frame
0536      memcpy(&sCommFrameHello,&psMeshDcfmInd-
>uParam.sMeshIndData.au8Mhsdu[1],sizeof(NET_CommFrameHello_s));
0537
0538      #ifdef DEBUG
0539      vStackPrintf(__FILE__,__LINE__,"Debugging Hello Command Frame RECEIVING.");
0540      vPrintf("\n");
0541      vPrintf("TTL=%d\n",sCommFrameHello.u8TTL);
0542      vPrintf("Beginning Address=%d\n",sCommFrameHello.u16BeginningAddress);
0543      vPrintf("Ending Address=%d\n",sCommFrameHello.u16EndingAddress);
0544      vPrintf("Tree Level=%d\n",sCommFrameHello.u8TreeLevel);
0545      vPrintf("Hello Control=%d\n",sCommFrameHello.u8HelloControl);
0546      vPrintf("Number of one hop neighbors=%d\n",sCommFrameHello.u8NumberOfOneHopNeighbors);
0547      for(i=0;i<sCommFrameHello.u8NumberOfOneHopNeighbors;i++)
0548          vPrintf("\tNegibor %d address=%d\n",i,sCommFrameHello.au16AddressesOfOneHopNeighbors[i]);
0549      vPrintf("Number of one multicas groups=%d\n",sCommFrameHello.u8NumberOfMulticastGroups);
0550      for(i=0;i<sCommFrameHello.u8NumberOfMulticastGroups;i++)
0551          vPrintf("\tGroup %d address=%d\n",i,sCommFrameHello.au16AddressesOfMulticastGroups[i]);
0552      vPrintf("Hello frame id=%d\n",sCommFrameHello.u8NbOfHello);
0553      vPrintf("PERFORMING UPDATES. . . \n");
0554      #endif
0555
0556      //Processing received frame
0557      if(sCommFrameHello.u16BeginningAddress==sMeshInfo.u16NetworkAddress)
0558      {
0559          #ifdef DEBUG
0560          vPrintf("It is my own hello command frame.\n");
0561          vPrintf("\n\n");
0562          vStackPrintf(__FILE__,__LINE__,"End of debugging Hello Command Frame RECEIVING.");
0563          #endif
0564          //Filing the defered confirm
0565          psCommDcfmInd->u8Type=FRAME_HELLO;
0566          psCommDcfmInd->u8Status=SUCCESS;
0567
0568          //Deregister this function off the debugger
0569          #ifdef DEBUG
0570          vStackPopIdentifier();
0571          #endif
0572
0573          return ;
0574      }
0575
0576
0577      if((sCommFrameHello.u8HelloControl&0x080)!=0x00)
0578      {
0579          for(i=0;i<MAX_NEIGHBORS;i++)
0580          {
0581              if( (sMeshInfo.psNeighborList[i].u8Relationship!=NO_RELATIONSHIP)&&
0582                  (sMeshInfo.psNeighborList[i].u16BeginningAddress==sCommFrameHello.u16BeginningAddress))
0583              {
0584                  //Checking the device relation with this neighbor
0585                  if(sMeshInfo.psNeighborList[i].u8Relationship==CHILD)
0586                  {
0587                      //Clearing the connectivity matrix
0588                      sMeshData.p2bConnectivityMatrix[0][i+1]=\
0589                      sMeshData.p2bConnectivityMatrix[i+1][0]=FALSE;
0590
0591                      //Changing its relationship to the child
0592                      sMeshInfo.psNeighborList[i].u8Relationship=LEFT;
```

```
0593                    sMeshInfo.psNeighborList[i].u8NbOfHello=0;
0594
0595                    //Starting a rejoin timer
0596
sMeshInfo.psNeighborList[i].i8RejoinTimer=VirtualTimer_i8New(sMeshInfo.u16RejoinTimer*10);
0597                    VirtualTimer_bReset(sMeshInfo.psNeighborList[i].i8RejoinTimer);
0598                    VirtualTimer_bCount(sMeshInfo.psNeighborList[i].i8RejoinTimer);
0599
0600                    //Place the child in rejoin
0601                    PLACEBITMAP(sMeshData.sBitMapRejoin,i);
0602                }
0603                else
0604                {
0605                    //Changing its relationship to this neighbor
0606                    sMeshInfo.psNeighborList[i].u16BeginningAddress=0xfffe;
0607                    sMeshInfo.psNeighborList[i].u8TreeLevel=0xff;
0608                    sMeshInfo.psNeighborList[i].u8Relationship=NO_RELATIONSHIP;
0609                    sMeshInfo.psNeighborList[i].u8NumberOfOps=0xff;
0610                    sMeshInfo.psNeighborList[i].u8NbOfHello=0;
0611                    for(j=0;j<MAX_NEIGHBORS+1;j++)
0612                        sMeshData.p2bConnectivityMatrix[j][i+1]=\
0613                        sMeshData.p2bConnectivityMatrix[i+1][j]=FALSE;
0614                }
0615                break;
0616            }
0617        }
0618    }
0619    else
0620    {
0621        for(i=0;i<MAX_NEIGHBORS;i++)
0622        {
0623            //If there is a mach in the neighbor list
0624            if( (sMeshInfo.psNeighborList[i].u16BeginningAddress==sCommFrameHello.u16BeginningAddress)&&
0625                (sMeshInfo.psNeighborList[i].u8Relationship!=NO_RELATIONSHIP))
0626            {
0627                #ifdef DEBUG
0628                vPrintf("Neighbor matched in neighbor list\n");
0629                vPrintf("Current
NbOfHello=%d\n",(sMeshInfo.psNeighborList[i].u8NbOfHello)%(8*sizeof(uint8)-1));
0630                vPrintf("New NbOfHello=%d\n",(sCommFrameHello.u8NbOfHello)%(8*sizeof(uint8)));
0631                #endif
0632
0633                //If didnt processed this Hello frame for this neighbor
0634                if((sMeshInfo.psNeighborList[i].u8NbOfHello)%(8*sizeof(uint8)-
1)<(sCommFrameHello.u8NbOfHello)%(8*sizeof(uint8)))
0635                {
0636                    #ifdef DEBUG
0637                    vPrintf("It is a new hello command. Performing update for this neighbor\n");
0638                    #endif
0639
0640                    //Updating the neighbor list
0641                    sMeshInfo.psNeighborList[i].u16BeginningAddress=sCommFrameHello.u16BeginningAddress;
0642                    sMeshInfo.psNeighborList[i].u16EndingAddress=sCommFrameHello.u16EndingAddress;
0643                    sMeshInfo.psNeighborList[i].u8TreeLevel=sCommFrameHello.u8TreeLevel;
0644                    sMeshInfo.psNeighborList[i].u8NumberOfOps=sMeshInfo.u8TtlOfHello-
sCommFrameHello.u8TTL+1;
0645                    sMeshInfo.psNeighborList[i].u8NbOfHello=sCommFrameHello.u8NbOfHello;
0646
0647                    //Updating the connectivity matrix
0648                    if(sMeshInfo.psNeighborList[i].u8NumberOfOps==1)
0649                    {
0650                        sMeshData.p2bConnectivityMatrix[0][i+1]=\
0651                        sMeshData.p2bConnectivityMatrix[i+1][0]=TRUE;
0652                    }
0653
```

```
0654                    #ifdef DEBUG
0655                    vPrintf("\tBeginingAddress=%d\n",sMeshInfo.psNeighborList[i].u16BeginningAddress);
0656                    vPrintf("\tEndingAddress=%d\n",sMeshInfo.psNeighborList[i].u16EndingAddress);
0657                    vPrintf("\tTreeLevel=%d\n",sMeshInfo.psNeighborList[i].u8TreeLevel);
0658                    vPrintf("\tNumberOfOps=%d\n",sMeshInfo.psNeighborList[i].u8NumberOfOps);
0659                    vPrintf("\tNbOfHello=%d\n",sMeshInfo.psNeighborList[i].u8NbOfHello);
0660                    #endif
0661
0662                    //Updating the one hop neighbors
0663                    if(sCommFrameHello.u8TTL>1)
0664                    {
0665                        #ifdef DEBUG
0666                        vPrintf("Updating the neighbor's one hop neighbors. there are %d of
them\n",sCommFrameHello.u8NumberOfOneHopNeighbors);
0667                        #endif
0668
0669                        for(j=0,n=sCommFrameHello.u8NumberOfOneHopNeighbors;j<n;j++)
0670                        {
0671
if(sMeshInfo.u16NetworkAddress==sCommFrameHello.au16AddressesOfOneHopNeighbors[j])
0672                                continue;
0673
0674                            bUpdated=FALSE;
0675
0676                            //If there is a match
0677                            for(k=0;k<MAX_NEIGHBORS;k++)
0678                            {
0679                                if(
(sMeshInfo.psNeighborList[k].u16BeginningAddress==sCommFrameHello.au16AddressesOfOneHopNeighbors[j])&&
0680                                    (sMeshInfo.psNeighborList[k].u8Relationship!=NO_RELATIONSHIP))
0681                                {
0682                                    //Updating the number of hops
0683                                    if(sMeshInfo.psNeighborList[k].u8NumberOfOps>sMeshInfo.u8TtlOfHello-
sCommFrameHello.u8TTL+2)
0684
sMeshInfo.psNeighborList[k].u8NumberOfOps=sMeshInfo.u8TtlOfHello-sCommFrameHello.u8TTL+2;
0685
0686                                    //Updating the connectivity matrix
0687                                    sMeshData.p2bConnectivityMatrix[i+1][k+1]=\
0688                                    sMeshData.p2bConnectivityMatrix[k+1][i+1]=TRUE;
0689                                    bUpdated=TRUE;
0690                                    break;
0691                                }
0692                            }
0693
0694                            //If it is new information
0695                            if(bUpdated==FALSE)
0696                            {
0697                                #ifdef DEBUG
0698                                vPrintf("One hop neigbor %d is new information. Updating. .
.\n",sCommFrameHello.au16AddressesOfOneHopNeighbors[j]);
0699                                #endif
0700
0701                                for(k=0;k<MAX_NEIGHBORS;k++)
0702                                {
0703                                    //If found an empty entry
0704                                    if(sMeshInfo.psNeighborList[k].u8Relationship==NO_RELATIONSHIP)
0705                                    {
0706                                        //Updating the neighbor list
0707                                        sMeshInfo.psNeighborList[k].u8Relationship=SIBLING_DEVICE;
0708
sMeshInfo.psNeighborList[k].u16BeginningAddress=sCommFrameHello.au16AddressesOfOneHopNeighbors[j];
0709
sMeshInfo.psNeighborList[k].u8NumberOfOps=sMeshInfo.u8TtlOfHello-sCommFrameHello.u8TTL+2;
0710
```

```
0711                                        //Updating the connectivity matrix
0712                                        sMeshData.p2bConnectivityMatrix[i+1][k+1]=\
0713                                        sMeshData.p2bConnectivityMatrix[k+1][i+1]=TRUE;
0714                                        break;
0715                                    }
0716                                }
0717                            }
0718                        }
0719                    }
0720                    bUpdated=TRUE;
0721                    break;
0722                }
0723
0724                //Discard this hello frame
0725                else
0726                {
0727                    #ifdef DEBUG
0728                    vPrintf("Old hello command detected.\n");
0729                    vPrintf("\n\n");
0730                    vStackPrintf(__FILE__,__LINE__,"End of debugging Hello Command Frame RECEIVING.");
0731                    #endif
0732
0733                    //Filing the defered confirm
0734                    psCommDcfmInd->u8Type=FRAME_HELLO;
0735                    psCommDcfmInd->u8Status=SUCCESS;
0736
0737                    //Deregister this function off the debugger
0738                    #ifdef DEBUG
0739                    vStackPopIdentifier();
0740                    #endif
0741
0742                    return ;
0743                }
0744            }
0745        }
0746
0747        //If there wasn't a match in the neighbor list
0748        if(bUpdated==FALSE)
0749        {
0750            #ifdef DEBUG
0751            vPrintf("Neigbor %d is new information. Updating. .
.\n",sCommFrameHello.u16BeginningAddress);
0752            #endif
0753            for(i=0;i<MAX_NEIGHBORS;i++)
0754            {
0755                //If found an empty entry
0756                if(sMeshInfo.psNeighborList[i].u8Relationship==NO_RELATIONSHIP)
0757                {
0758                    //Updating the neighbor list
0759                    sMeshInfo.psNeighborList[i].u8Relationship=SIBLING_DEVICE;
0760                    sMeshInfo.psNeighborList[i].u16BeginningAddress=sCommFrameHello.u16BeginningAddress;
0761                    sMeshInfo.psNeighborList[i].u16EndingAddress=sCommFrameHello.u16EndingAddress;
0762                    sMeshInfo.psNeighborList[i].u8TreeLevel=sCommFrameHello.u8TreeLevel;
0763                    sMeshInfo.psNeighborList[i].u8NumberOfOps=sMeshInfo.u8TtlOfHello-
sCommFrameHello.u8TTL+1;
0764                    sMeshInfo.psNeighborList[i].u8NbOfHello=sCommFrameHello.u8NbOfHello;
0765
0766                    //Updating the connectivity matrix
0767                    if(sMeshInfo.psNeighborList[i].u8NumberOfOps==1)
0768                    {
0769                        sMeshData.p2bConnectivityMatrix[0][i+1]=\
0770                        sMeshData.p2bConnectivityMatrix[i+1][0]=TRUE;
0771                    }
0772
0773                    #ifdef DEBUG
```

```
0774                    vPrintf("\tBeginingAddress=%d\n",sMeshInfo.psNeighborList[i].u16BeginningAddress);
0775                    vPrintf("\tEndingAddress=%d\n",sMeshInfo.psNeighborList[i].u16EndingAddress);
0776                    vPrintf("\tTreeLevel=%d\n",sMeshInfo.psNeighborList[i].u8TreeLevel);
0777                    vPrintf("\tNumberOfOps=%d\n",sMeshInfo.psNeighborList[i].u8NumberOfOps);
0778                    vPrintf("\tNbOfHello=%d\n",sMeshInfo.psNeighborList[i].u8NbOfHello);
0779                    #endif
0780
0781                    //Updating the one hop neighbors
0782                    if(sCommFrameHello.u8TTL>1)
0783                    {
0784                        #ifdef DEBUG
0785                        vPrintf("Updating the neighbor's one hop neighbors. there are %d of
them\n",sCommFrameHello.u8NumberOfOneHopNeighbors);
0786                        #endif
0787
0788                        for(j=0,n=sCommFrameHello.u8NumberOfOneHopNeighbors;j<n;j++)
0789                        {
0790
if(sMeshInfo.u16NetworkAddress==sCommFrameHello.au16AddressesOfOneHopNeighbors[j])
0791                                continue;
0792
0793                            bUpdated=FALSE;
0794
0795                            //If there is a match no need for update
0796                            for(k=0;k<MAX_NEIGHBORS;k++)
0797                            {
0798                                if(
(sMeshInfo.psNeighborList[k].u16BeginningAddress==sCommFrameHello.au16AddressesOfOneHopNeighbors[j])&&
0799                                    (sMeshInfo.psNeighborList[k].u8Relationship!=NO_RELATIONSHIP))
0800                                {
0801                                    //Updating the number of hops
0802                                    if(sMeshInfo.psNeighborList[k].u8NumberOfOps>sMeshInfo.u8TtlOfHello-
sCommFrameHello.u8TTL+2)
0803
sMeshInfo.psNeighborList[k].u8NumberOfOps=sMeshInfo.u8TtlOfHello-sCommFrameHello.u8TTL+2;
0804
0805                                    //Updating the connectivity matrix
0806                                    sMeshData.p2bConnectivityMatrix[i+1][k+1]=\
0807                                    sMeshData.p2bConnectivityMatrix[k+1][i+1]=TRUE;
0808
0809                                    bUpdated=TRUE;
0810                                    break;
0811                                }
0812                            }
0813
0814                            //If it is new information
0815                            if(bUpdated==FALSE)
0816                            {
0817                                #ifdef DEBUG
0818                                vPrintf("One hop neigbor %d is new information. Updating. .
.\n",sCommFrameHello.au16AddressesOfOneHopNeighbors[j]);
0819                                #endif
0820
0821                                for(k=0;k<MAX_NEIGHBORS;k++)
0822                                {
0823                                    //If found an empty entry
0824                                    if(sMeshInfo.psNeighborList[k].u8Relationship==NO_RELATIONSHIP)
0825                                    {
0826                                        //Updating the neighbor list
0827                                        sMeshInfo.psNeighborList[k].u8Relationship=SIBLING_DEVICE;
0828
sMeshInfo.psNeighborList[k].u16BeginningAddress=sCommFrameHello.au16AddressesOfOneHopNeighbors[j];
0829
sMeshInfo.psNeighborList[k].u8NumberOfOps=sMeshInfo.u8TtlOfHello-sCommFrameHello.u8TTL+2;
0830
```

```
0831                                        //Updating the connectivity matrix
0832                                        sMeshData.p2bConnectivityMatrix[i+1][k+1]=\
0833                                        sMeshData.p2bConnectivityMatrix[k+1][i+1]=TRUE;
0834                                        break;
0835                                    }
0836                                }
0837                            }
0838                        }
0839                    }
0840                    bUpdated=TRUE;
0841                    break;
0842                }
0843            }
0844        }
0845    }
0846    #ifdef DEBUG
0847    vPrintf("The new connectivity matrix:\n");
0848    for(i=0;i<MAX_NEIGHBORS+1;i++)
0849    {
0850        for(j=0;j<MAX_NEIGHBORS+1;j++)
0851            vPrintf("%d ",sMeshData.p2bConnectivityMatrix[i][j]);
0852        vPrintf("\n");
0853    }
0854    #endif
0855
0856
0857    /** SHOULD IT SEND IF IT WAS NOT SUCCESSFULY UPDATED? **/
0858    //If it could update successfuly send the TTL of hello
0859    if((bUpdated==TRUE) && (sCommFrameHello.u8TTL>1))
0860    {
0861        NET_DataReqRsp_s                sDataReqRsp;
0862        NET_DataSyncCfm_s               sDataSyncCfm;
0863
0864        //Filling hello structure
0865        sCommFrameHello.u8TTL--;
0866
0867        //Sending hello frame
0868        sDataReqRsp.u8Type=NET_DATA_IND_COMM;
0869        sDataReqRsp.u8MhsduHandle=(uint8)i16MhmeInsertMhsduHandle(FRAME_HELLO);
0870        //No MAC destination is specified
0871        sDataReqRsp.sDataFrameData.u16SrcPANId=sMeshInfo.u16PanId;
0872        sDataReqRsp.sDataFrameData.u8SrcAddrMode=2;
0873        sDataReqRsp.sDataFrameData.u8DstAddrMode=2;
0874        sDataReqRsp.sDataFrameData.uSrcAddr.u16Short=sMeshInfo.u16NetworkAddress;
0875        sDataReqRsp.sDataFrameData.uDstAddr.u16Short=0xffff;
0876        sDataReqRsp.sDataFrameData.u8MhsduLength=sizeof(NET_CommFrameHello_s)+1;
0877        sDataReqRsp.sDataFrameData.au8Mhsdu[0]=FRAME_HELLO;
0878        memcpy(&sDataReqRsp.sDataFrameData.au8Mhsdu[1],&sCommFrameHello,sizeof(NET_CommFrameHello_s));
0879        sDataReqRsp.sDataFrameData.u8AckTransmission=FALSE;
0880        sDataReqRsp.sDataFrameData.u8McstTransmission=FALSE;
0881        sDataReqRsp.sDataFrameData.u8BcstTransmission=TRUE;
0882        sDataReqRsp.sDataFrameData.u8ReliableBcst=FALSE;
0883
0884        #ifdef DEBUG
0885        vPrintf("resending with TTL=%d\n",sCommFrameHello.u8TTL);
0886        #endif
0887
0888        vNetApiDataRequest(&sDataReqRsp,&sDataSyncCfm);
0889    }
0890
0891    #ifdef DEBUG
0892    vPrintf("\n\n");
0893    vStackPrintf(__FILE__,__LINE__,"End of debugging Hello Command Frame RECEIVING.");
0894    #endif
0895
```

```
0896     //Filing the defered confirm
0897     psCommDcfmInd->u8Type=FRAME_HELLO;
0898     psCommDcfmInd->u8Status=SUCCESS;
0899
0900     //Deregister this function off the debugger
0901     #ifdef DEBUG
0902     vStackPopIdentifier();
0903     #endif
0904 }
0905 inline void vHdlCommRecvLeave(NET_CommDcfmInd_s* psCommDcfmInd,NET_MeshDcfmInd_s* psMeshDcfmInd)
0906 {
0907     NET_CommFrameLeave_s    sCommFrameLeave;
0908     NET_MhmeReqLeave_s      sMhmeReqLeave;
0909     NET_MhmeCfmLeave_s      sMhmeCfmLeave;
0910
0911     //Getting the address assignment frame
0912     memcpy(&sCommFrameLeave,&psMeshDcfmInd-
>uParam.sMeshIndData.au8Mhsdu[1],sizeof(NET_CommFrameLeave_s));
0913
0914     //Register this function on the debugger
0915     #ifdef DEBUG
0916     u8StackPushIdentifier("vHdlCommRecvLeave",strlen("vHdlCommRecvLeave"),FALSE);
0917     #endif
0918
0919     sMhmeReqLeave.u8RemoveSelf=TRUE;
0920     sMhmeReqLeave.u8RemoveChildren=((sCommFrameLeave.u8LeaveControl&0x080)!=0)?TRUE:FALSE;
0921
0922     //Request to leave network
0923     vHdlMhmeReqLeave(&sMhmeReqLeave, &sMhmeCfmLeave);
0924
0925     //Filing the defered confirm
0926     psCommDcfmInd->u8Type=FRAME_LEAVE;
0927     psCommDcfmInd->u8Status=sMhmeCfmLeave.u8Status;
0928
0929     //Deregister this function off the debugger
0930     #ifdef DEBUG
0931     vStackPopIdentifier();
0932     #endif
0933 }
```

IV. CommandFrames.h

```
0001 #ifndef COMMAND_FRAMES
0002 #define COMMAND_FRAMES
0003
0004 #include "Mesh.h"
0005 #include "Mhme.h"
0006
0007 #define    FRAME_UNKNOWN                        0x00
0008 #define    FRAME_CHILDREN_NUMBER_REPORT         0x01
0009 #define    FRAME_ADDRESS_ASSIGNMENT             0x02
0010 #define    FRAME_HELLO                          0x03
0011 #define    FRAME_NEIGHBOR_INFORMATION_REQUEST   0x04
0012 #define    FRAME_NEIGHBOR_INFORMATION_REPLY     0x05
0013 #define    FRAME_LINK_STATE                     0x06
0014 #define    FRAME_LINK_STATE_MISMATCH            0x07
0015 #define    FRAME_PROBE                          0x08
0016 #define    FRAME_G_JREQ                         0x09
0017 #define    FRAME_G_JREP                         0x0A
0018 #define    FRAME_G_LREQ                         0x0B
0019 #define    FRAME_GROUP_LEAVE_REPLY              0x0C
0020 #define    FRAME_WAKEUP_NOTIFICATION            0x0D
0021 #define    FRAME_EXTENSION_REQUEST              0x0E
0022 #define    FRAME_EXTENSION_REPLY                0x0F
```

```
0023 #define    FRAME_SYNCHRONIZATION_REQUEST           0x10
0024 #define    FRAME_SYNCHRONIZATION_REPLY             0x11
0025 #define    FRAME_RESERVATION_REQUEST               0x12
0026 #define    FRAME_RESERVATION_REPLY                 0x13
0027 #define    FRAME_REJOIN_NOTIFY                     0x14
0028 #define    FRAME_TRACEROUTE_REQUEST                0x15
0029 #define    FRAME_TRACEROUTE_REPLY                  0x16
0030 #define    FRAME_LEAVE                             0x17
0031 #define     FRAME_INVALID                           0x18
0032
0033
0034 typedef struct
0035 {
0036     uint16 u16NumberOfDescendants;
0037     uint16 u16NumberOfRequestedAddresses;
0038
0039 }NET_CommFrameChildReport_s;
0040
0041 typedef struct
0042 {
0043     uint16 u16BeginningAddress;
0044     uint16 u16EndingAddress;
0045     uint8 u8ParentTreeLevel;
0046     uint8 u8Pad;
0047
0048 }NET_CommFrameAddrAssignment_s;
0049
0050 typedef struct
0051 {
0052     uint8   u8TTL;
0053     uint8   u8TreeLevel;
0054     uint16  u16Pad;
0055     uint16  u16BeginningAddress;
0056     uint16  u16EndingAddress;
0057     uint8   u8HelloControl;
0058     uint8   u8NumberOfOneHopNeighbors;
0059     uint8   u8NumberOfMulticastGroups;
0060     uint8   u8NbOfHello;
0061     uint16  au16AddressesOfOneHopNeighbors[MAX_NEIGHBORS];
0062     uint16  au16AddressesOfMulticastGroups[MAX_NEIGHBORS];
0063
0064 }NET_CommFrameHello_s;
0065
0066 typedef struct
0067 {
0068     uint8   u8LeaveControl;
0069     uint8   u8Pad;
0070     uint16  u16Pad;
0071
0072 }NET_CommFrameLeave_s;
0073
0074
0075
0076 typedef struct
0077 {
0078     NET_CommFrameChildReport_s      sCommFrameChildReport;
0079     NET_CommFrameAddrAssignment_s   sCommFrameAddrAssignment;
0080     NET_CommFrameHello_s            sCommFrameHello;
0081     NET_CommFrameLeave_s            sCommFrameLeave;
0082
0083 }NET_CommReqRspParam_u;
0084
0085 typedef struct
0086 {
0087     uint8                   u8Type;
```

```
0088     uint8                   u8Pad;
0089     uint16                  u16Pad;
0090     NET_CommReqRspParam_u   uParam;
0091
0092 }NET_CommReqRsp_s;
0093
0094
0095
0096 typedef struct
0097 {
0098     uint8                   u8Type;
0099     uint8                   u8Status;
0100     uint16                  u16Pad;
0101
0102 }NET_CommDcfmInd_s;
0103
0104
0105
0106 typedef struct
0107 {
0108     uint8                   u8Type;
0109     uint8                   u8Status;
0110     uint16                  u16Pad;
0111
0112 }NET_CommSyncCfm_s;
0113
0114
0115 void vNetApiCommRequest(uint16 u16CommandFrameNumber,NET_CommSyncCfm_s* psCommSyncCfm);
0116 void vNetApiCommTranslate(NET_CommDcfmInd_s* psCommDcfmInd,NET_MeshDcfmInd_s* psMeshDcfmInd);
0117
0118 #endif //COMMAND_FRAMES
```

V. Config.h

```
0001 #ifndef _CONFIG_H
0002 #define _CONFIG_H
0003
0004 /** If defined, enables debug printf's to UART0 **/
0005 //#define DEBUG
0006
0007 /** PAN ID **/
0008 #define PAN_ID 0xA7A6
0009 /** fc=2480MHz, more info: JN-AN-1059 v1.1 pg. 37 and JN-UG-3041 v1.3 pg. 8 **/
0010 #define CHANNEL 25
0011
0012 #define MAX_NEIGHBORS 10
0013
0014 #define ISCOORDINATOR
0015
0016 #endif //_CONFIG_H
```

VI. DataFrames.h

```
0001 #include <string.h>
0002 #include <AppApi.h>
0003
0004 #include "mesh.h"
0005 #include "DataFrames.h"
0006 #include "MeshServices.h"
0007 #include "mhme.h"
0008
0009 #ifdef DEBUG
0010 #include "debugger.h"
```

```
0011 #endif
0012
0013 #define SAME64ADDR(a,b) ((a.u32H==b.u32H)&&(a.u32L==b.u32L))
0014
0015 extern NET_MeshInfo_s sMeshInfo;
0016 extern NET_MeshMhsduHandleQueue_s sMeshMhsduHandleQueue;
0017
0018
0019 void vNetApiDataRequest(NET_DataReqRsp_s *psDataReqRsp, NET_DataSyncCfm_s *psDataSyncCfm)
0020 {
0021     MAC_McpsReqRsp_s    sMcpsReqRsp;
0022     MAC_McpsSyncCfm_s   sMcpsSyncCfm;
0023
0024     uint8               u8payload[MAC_MAX_DATA_PAYLOAD_LEN];
0025     uint8               i;
0026     int8                i8MeshHdl;
0027
0028     //Identify request type
0029     psDataSyncCfm->u8Type=psDataReqRsp->u8Type;
0030
0031     //Clean
0032     memset(u8payload,0,MAC_MAX_DATA_PAYLOAD_LEN*sizeof(uint8));
0033
0034     //reset counter
0035     i=0;
0036
0037     //Begin: Frame Control Field
0038     //1st Octet
0039     //Protocol version b 3:0
0040     u8payload[i]|= 0x01;
0041
0042     //Command or data frame b 4
0043     if(psDataReqRsp->u8Type==NET_DATA_IND_COMM) u8payload[i]|= 0x010;
0044
0045     //Destination address mode b 5
0046     if(psDataReqRsp->sDataFrameData.u8DstAddrMode == 2) u8payload[i]|= 0x020;
0047
0048     //Source address mode b 6
0049     if(psDataReqRsp->sDataFrameData.u8SrcAddrMode == 2) u8payload[i]|= 0x040;
0050
0051     //Ack transmission b 7
0052     if(psDataReqRsp->sDataFrameData.u8AckTransmission) u8payload[i]|= 0x080;
0053
0054     //2nd Octet
0055     i++;
0056
0057     //Multicast transmission b 0
0058     if(psDataReqRsp->sDataFrameData.u8McstTransmission) u8payload[i]|= 0x01;
0059
0060     //Broadcast transmission b 1
0061     if(psDataReqRsp->sDataFrameData.u8BcstTransmission) u8payload[i]|= 0x02;
0062
0063     //Reliable broadcast transmission b 2
0064     if(psDataReqRsp->sDataFrameData.u8ReliableBcst) u8payload[i]|= 0x04;
0065
0066     //Reserved bits: 11 - 15
0067
0068     //3rd Octet
0069     i++;
0070
0071     //Begin: Destination Address 2 to 8 octets
0072     switch(psDataReqRsp->sDataFrameData.u8DstAddrMode)
0073     {
0074         case 2:
0075         {
```

```
0076            //Short mode: 16 bits
0077            memcpy(&u8payload[i],&psDataReqRsp->sDataFrameData.uDstAddr.u16Short,sizeof(uint16));
0078            i+=sizeof(uint16);
0079        }break;
0080
0081        case 3:
0082        {
0083            //Extended Mode: 64 bits
0084            memcpy(&u8payload[i],&psDataReqRsp->sDataFrameData.uDstAddr.sExt,sizeof(MAC_ExtAddr_s));
0085            i+=sizeof(MAC_ExtAddr_s);
0086        }break;
0087
0088        default:
0089            //ATTENTION: error
0090            //return;
0091        break;
0092    }
0093
0094
0095    //Begin: Source Address 2 to 8 octets
0096    switch(psDataReqRsp->sDataFrameData.u8SrcAddrMode)
0097    {
0098        case 2:
0099        {
0100            //Short mode: 16 bits
0101            memcpy(&u8payload[i],&psDataReqRsp->sDataFrameData.uSrcAddr.u16Short,sizeof(uint16));
0102            i+=sizeof(uint16);
0103        }break;
0104
0105        case 3:
0106        {
0107            //Extended Mode: 64 bits
0108            memcpy(&u8payload[i],&psDataReqRsp->sDataFrameData.uSrcAddr.sExt,sizeof(MAC_ExtAddr_s));
0109            i+=sizeof(MAC_ExtAddr_s);;
0110
0111        }break;
0112
0113        default:
0114            //ATTENTION: error
0115            //return;
0116        break;
0117    }
0118
0119    //Concatenate with MHSDU
0120    memcpy(&u8payload[i],psDataReqRsp->sDataFrameData.au8Mhsdu,psDataReqRsp-
>sDataFrameData.u8MhsduLength);
0121
0122    //Create MAC frame
0123    sMcpsReqRsp.u8Type = MAC_MCPS_REQ_DATA;
0124    sMcpsReqRsp.u8ParamLength = sizeof(MAC_McpsReqData_s);
0125
0126    //Set handle so we can match confirmation to request  **** ATTENTION ****
0127    if( (i8MeshHdl=i16InsertMhsduHandle(psDataReqRsp->u8MhsduHandle,
0128        (psDataReqRsp-
>u8Type==NET_DATA_IND_COMM)?NET_MESH_MSDU_HANDLE_COMMAND:NET_MESH_MSDU_HANDLE_DATA))==-1)
0129    {
0130        psDataSyncCfm->u8Status = NET_ENUM_TRANSACTION_OVERFLOW;
0131        return;
0132    }
0133
0134    sMcpsReqRsp.uParam.sReqData.u8Handle = i8MeshHdl;
0135
0136    #ifdef DEBUG
0137    vPrintf("HANDLE used=%d\n",sMcpsReqRsp.uParam.sReqData.u8Handle );
0138    #endif
```

```
0139
0140      //Put source details
0141      sMcpsReqRsp.uParam.sReqData.sFrame.sSrcAddr.u8AddrMode = 3;
0142      sMcpsReqRsp.uParam.sReqData.sFrame.sSrcAddr.u16PanId = sMeshInfo.u16PanId;
0143
memcpy(&sMcpsReqRsp.uParam.sReqData.sFrame.sSrcAddr.uAddr.sExt,&sMeshInfo.sAddressMapping.sExtAddr,sizeof(MAC
_ExtAddr_s));
0144
0145      //Put destination details
0146      sMcpsReqRsp.uParam.sReqData.sFrame.sDstAddr.u8AddrMode = psDataReqRsp-
>sDataFrameData.u8BcstTransmission?2:3;
0147      sMcpsReqRsp.uParam.sReqData.sFrame.sDstAddr.u16PanId = psDataReqRsp->sDataFrameData.u16SrcPANId;
0148      if(psDataReqRsp->sDataFrameData.u8BcstTransmission)
0149          sMcpsReqRsp.uParam.sReqData.sFrame.sDstAddr.uAddr.u16Short=0xffff;
0150      else
0151          memcpy(&sMcpsReqRsp.uParam.sReqData.sFrame.sDstAddr.uAddr.sExt,&psDataReqRsp-
>sExt,sizeof(MAC_ExtAddr_s));
0152
0153      //ATTENTION!! Field must be updated according to the upper layer
0154      if (psDataReqRsp->sDataFrameData.u8AckTransmission)
0155          sMcpsReqRsp.uParam.sReqData.sFrame.u8TxOptions = MAC_TX_OPTION_ACK;
0156      else
0157          sMcpsReqRsp.uParam.sReqData.sFrame.u8TxOptions = 0x0; //NO ACK
0158
0159      sMcpsReqRsp.uParam.sReqData.sFrame.u8SduLength = i + psDataReqRsp->sDataFrameData.u8MhsduLength;
0160      ///flag
0161
memcpy(sMcpsReqRsp.uParam.sReqData.sFrame.au8Sdu,u8payload,sMcpsReqRsp.uParam.sReqData.sFrame.u8SduLength);
0162
0163      vAppApiMcpsRequest(&sMcpsReqRsp, &sMcpsSyncCfm);
0164      if(sMcpsSyncCfm.u8Status==MAC_MCPS_CFM_DEFERRED)
0165      {
0166          #ifdef DEBUG
0167          vPrintf("Ok we are sending. . .\n");
0168          vPrintf("Source: %x %x\n",  sMcpsReqRsp.uParam.sReqData.sFrame.sSrcAddr.uAddr.sExt.u32H,
0169                                      sMcpsReqRsp.uParam.sReqData.sFrame.sSrcAddr.uAddr.sExt.u32L);
0170          vPrintf("Destination: %x %x\n",  sMcpsReqRsp.uParam.sReqData.sFrame.sDstAddr.uAddr.sExt.u32H,
0171                                      sMcpsReqRsp.uParam.sReqData.sFrame.sDstAddr.uAddr.sExt.u32L);
0172          vPrintf("DATA
PARSED:\nu16MhdrLength=%d\nu8mhsduLength=%d\n",i,sMcpsReqRsp.uParam.sReqData.sFrame.u8SduLength);
0173          vPrintf("Max data allowed=%d\n",MAC_MAX_DATA_PAYLOAD_LEN);
0174          vPrintf("Message Sent = ");
0175          for(i=0;i<sMcpsReqRsp.uParam.sReqData.sFrame.u8SduLength;i++)
0176              vPrintf("%x ",sMcpsReqRsp.uParam.sReqData.sFrame.au8Sdu[i]);
0177          vPrintf("\n");
0178          #endif
0179      }
0180
0181      psDataSyncCfm->u8Status=SUCCESS;
0182  }
0183
0184
0185
0186  bool bIsCommandFrame(uint8* pu8Payload)
0187  {
0188      return pu8Payload[0]&0x010;
0189  }
0190
0191
0192
0193  void vNetApiDataTranslate(NET_DataDcfmInd_s* psDataDcfmInd,MAC_McpsDcfmInd_s* psMcpsDcfmInd)
0194  {
0195      uint16  u16MhdrLength=0;
0196      uint16  i;
0197      bool    bSrcIsShort;
```

```
0198     bool    bDstIsShort;
0199     bool    bUpdated;
0200
0201     #ifdef DEBUG
0202     vPrintf("Handling Mesh Indication Data.\n");
0203     #endif
0204
0205     //Check for data type
0206     psDataDcfmInd->u8Type=bIsCommandFrame(psMcpsDcfmInd-
>uParam.sIndData.sFrame.au8Sdu)?NET_DATA_IND_COMM:NET_DATA_IND_DATA;
0207
0208     //Transmission flags
0209     psDataDcfmInd->sDataInd.sDataFrameData.u8AckTransmission=((psMcpsDcfmInd-
>uParam.sIndData.sFrame.au8Sdu[0]&0x080)!=0);
0210     psDataDcfmInd->sDataInd.sDataFrameData.u8McstTransmission=((psMcpsDcfmInd-
>uParam.sIndData.sFrame.au8Sdu[1]&0x01)!=0);
0211     psDataDcfmInd->sDataInd.sDataFrameData.u8BcstTransmission=((psMcpsDcfmInd-
>uParam.sIndData.sFrame.au8Sdu[1]&0x02)!=0);
0212     psDataDcfmInd->sDataInd.sDataFrameData.u8ReliableBcst=((psMcpsDcfmInd-
>uParam.sIndData.sFrame.au8Sdu[1]&0x04)!=0);
0213
0214     //Control bytes
0215     u16MhdrLength+=2;
0216
0217     //Destination address
0218     bDstIsShort=(psMcpsDcfmInd->uParam.sIndData.sFrame.au8Sdu[0]&0x020)!=0;
0219     psDataDcfmInd->sDataInd.sDataFrameData.u8DstAddrMode=(bDstIsShort?2:3);
0220     memcpy( bDstIsShort?(void*)&psDataDcfmInd-
>sDataInd.sDataFrameData.uDstAddr.u16Short:(void*)&psDataDcfmInd->sDataInd.sDataFrameData.uDstAddr.sExt,
0221             &psMcpsDcfmInd->uParam.sIndData.sFrame.au8Sdu[u16MhdrLength],
0222             bDstIsShort?sizeof(uint16):sizeof(MAC_ExtAddr_s));
0223     #ifdef DEBUG
0224     bDstIsShort?    vPrintf("\n\n\nDst Addr: %d\n\n\n",psDataDcfmInd-
>sDataInd.sDataFrameData.uDstAddr.u16Short):\
0225                     vPrintf("\n\n\nDst Addr: %x %x\n\n\n",psDataDcfmInd-
>sDataInd.sDataFrameData.uDstAddr.sExt.u32H,\
0226                                                     psDataDcfmInd-
>sDataInd.sDataFrameData.uDstAddr.sExt.u32L);
0227     #endif
0228
0229     //Destination address mode
0230     u16MhdrLength+=(bDstIsShort?2:8);
0231
0232     //Source address
0233     bSrcIsShort=(psMcpsDcfmInd->uParam.sIndData.sFrame.au8Sdu[0]&0x040)!=0;
0234     psDataDcfmInd->sDataInd.sDataFrameData.u8SrcAddrMode=(bSrcIsShort?2:3);
0235     memcpy( bSrcIsShort?(void*)&psDataDcfmInd-
>sDataInd.sDataFrameData.uSrcAddr.u16Short:(void*)&psDataDcfmInd->sDataInd.sDataFrameData.uSrcAddr.sExt,
0236             &psMcpsDcfmInd->uParam.sIndData.sFrame.au8Sdu[u16MhdrLength],
0237             bSrcIsShort?sizeof(uint16):sizeof(MAC_ExtAddr_s));
0238     #ifdef DEBUG
0239     bSrcIsShort?    vPrintf("\n\n\nSrc Addr: %d\n\n\n",psDataDcfmInd-
>sDataInd.sDataFrameData.uSrcAddr.u16Short):\
0240                     vPrintf("\n\n\nSrc Addr: %x %x\n\n\n",psDataDcfmInd-
>sDataInd.sDataFrameData.uSrcAddr.sExt.u32H,\
0241                                                     psDataDcfmInd-
>sDataInd.sDataFrameData.uSrcAddr.sExt.u32L);
0242     #endif
0243
0244     //Source address mode
0245     u16MhdrLength+=(bSrcIsShort?2:8);
0246
0247     //Filling the pan id filed
0248     psDataDcfmInd->sDataInd.sDataFrameData.u16SrcPANId=psMcpsDcfmInd-
>uParam.sIndData.sFrame.sSrcAddr.u16PanId;
```

```
0249
0250      //length and Mhsdu
0251      psDataDcfmInd->sDataInd.sDataFrameData.u8MhsduLength=psMcpsDcfmInd-
>uParam.sIndData.sFrame.u8SduLength-u16MhdrLength;
0252      memcpy(psDataDcfmInd->sDataInd.sDataFrameData.au8Mhsdu,&psMcpsDcfmInd-
>uParam.sIndData.sFrame.au8Sdu[u16MhdrLength],
0253            (psMcpsDcfmInd->uParam.sIndData.sFrame.u8SduLength-u16MhdrLength)*sizeof(uint8));
0254
0255      #ifdef DEBUG
0256      vPrintf("DATA PARSED:\nu16MhdrLength=%d\nu8mhsduLength=%d\n",u16MhdrLength,psDataDcfmInd-
>sDataInd.sDataFrameData.u8MhsduLength);
0257      vPrintf("au8Sdu = ");
0258      for(i=0;i<psMcpsDcfmInd->uParam.sIndData.sFrame.u8SduLength;i++)
0259          vPrintf("%x ",psMcpsDcfmInd->uParam.sIndData.sFrame.au8Sdu[i]);
0260      vPrintf("\n");
0261      #endif
0262
0263      //Updating neighbor list with possible new information
0264      bUpdated=FALSE;
0265
0266      //Updating mac address of the sender in the neighbor list
0267      if(psMcpsDcfmInd->uParam.sIndData.sFrame.sSrcAddr.u8AddrMode==3)
0268      {
0269          for(i=0;i<MAX_NEIGHBORS;i++)
0270          {
0277              if((sMeshInfo.psNeighborList[i].u8Relationship!=NO_RELATIONSHIP)&&
0278                  SAME64ADDR(psMcpsDcfmInd-
>uParam.sIndData.sFrame.sSrcAddr.uAddr.sExt,sMeshInfo.psNeighborList[i].sExt))
0279              {
0280                  bUpdated=TRUE;
0284
0285                  //Updating the u16address of the sender in the neighbor list
0286                  if(bSrcIsShort)
0287                  {
0288                      sMeshInfo.psNeighborList[i].u16BeginningAddress=psDataDcfmInd-
>sDataInd.sDataFrameData.uSrcAddr.u16Short;
0289                      #ifdef DEBUG
0296                  }
0297                  break;
0298              }
0299          }
0300      }
0301
0302      //If the neighbor is not in the neighbor list
0303      if(!bUpdated)
0304      {
0305          for(i=0;i<MAX_NEIGHBORS;i++)
0306          {
0307              if(sMeshInfo.psNeighborList[i].u8Relationship==NO_RELATIONSHIP)
0308              {
0309                  sMeshInfo.psNeighborList[i].u8Relationship=SIBLING_DEVICE;
0310                  memcpy(&sMeshInfo.psNeighborList[i].sExt,&psMcpsDcfmInd-
>uParam.sIndData.sFrame.sSrcAddr.uAddr.sExt,sizeof(MAC_ExtAddr_s));
0314
0315                  //Updating the u16address of the sender in the neighbor list
0316                  if(bSrcIsShort)
0317                  {
0318                      sMeshInfo.psNeighborList[i].u16BeginningAddress=psDataDcfmInd-
>sDataInd.sDataFrameData.uSrcAddr.u16Short;
0319
0327                  }
0328                  break;
0329              }
0330          }
0331      }
```

```
0332 }
```

VII. DataFrames.h

```
0001 #ifndef DATA_FRAMES
0002 #define DATA_FRAMES
0003
0004 #include "Mesh.h"
0005 #include "Mhme.h"
0006
0007 #define NET_DATA_IND_DATA                    0x00
0008 #define NET_DATA_IND_COMM                    0x01
0009
0010
0011 typedef struct
0012 {
0013     uint16          u16SrcPANId;
0014     uint8           u8SrcAddrMode;
0015     uint8           u8DstAddrMode;
0016     uint8           u8MhsduLength;
0017     uint8           u8AckTransmission;
0018     uint8           u8McstTransmission;
0019     uint8           u8BcstTransmission;
0020     uint8           u8ReliableBcst;
0021     uint8           u8Pad;
0022     uint16          u16Pad;
0023     uint8           au8Mhsdu[MAC_MAX_DATA_PAYLOAD_LEN];
0024     MAC_Addr_u      uSrcAddr;
0025     MAC_Addr_u      uDstAddr;
0026
0027 }NET_DataFrameData_s;
0028
0029
0030
0031 typedef struct
0032 {
0033     uint8                   u8Type;
0034     uint8                   u8MhsduHandle;
0035     uint16                  u16Pad;
0036     MAC_ExtAddr_s           sExt;
0037     NET_DataFrameData_s     sDataFrameData;
0038
0039 }NET_DataReqRsp_s;
0040
0041
0042
0043 typedef struct
0044 {
0045     NET_DataFrameData_s     sDataFrameData;
0046
0047 }NET_DataInd_s;
0048
0049
0050 typedef struct
0051 {
0052     uint8                   u8Type;
0053     uint8                   u8Pad;
0054     uint16                  u16Pad;
0055     NET_DataInd_s           sDataInd;
0056
0057 }NET_DataDcfmInd_s;
0058
0059
0060
```

```
0061 typedef struct
0062 {
0063     uint8                   u8Type;
0064     uint8                   u8Status;
0065     uint16                  u16Pad;
0066
0067 }NET_DataSyncCfm_s;
0068
0069
0070 void vNetApiDataRequest(NET_DataReqRsp_s *psDataReqRsp, NET_DataSyncCfm_s *psDataSyncCfm);
0071 void vNetApiDataTranslate(NET_DataDcfmInd_s* psDataDcfmInd,MAC_McpsDcfmInd_s* psMcpsDcfmInd);
0072
0073 #endif //DATA_FRAMES
```

VIII. Debugger.c

```
0001 #include "Debugger.h"
0002 #include <string.h>
0003
0004 void vStackStart(void)
0005 {
0006     vUART_printInit();
0007     sStackIdentifier.u16StackPosition=0;
0008     sStackIdentifier.u8IdentifierPosition=0;
0009     sStackIdentifier.u8CheckpointPosition=0;
0010 }
0011
0012 uint8 u8StackCheck(void)
0013 {
0014     if(sStackIdentifier.u8CheckpointPosition<MAX_CHECKPOINTS)
0015     {
0016         if(sStackIdentifier.u8IdentifierPosition>0)
0017
sStackIdentifier.au16StackCheckpoints[sStackIdentifier.u8CheckpointPosition++]=sStackIdentifier.u16StackPosit
ion+1;
0018         return 1;
0019     }
0020     return 0;
0021 }
0022
0023 uint8 u8StackUncheck(void)
0024 {
0025     if(sStackIdentifier.u8CheckpointPosition>0)
0026     {
0027         sStackIdentifier.u8CheckpointPosition--;
0028         return 1;
0029     }
0030     return 0;
0031 }
0032
0033 uint8 u8StackPushIdentifier(char* pcIdentifier,uint16 u16Size,bool check)
0034 {
0035     if(check)
0036         if(!u8StackCheck())
0037             return 0;
0038     if(sStackIdentifier.u16StackPosition+u16Size<MAX_STACK_LENGTH-2)
0039     {
0040         if(sStackIdentifier.u8IdentifierPosition>0)
0041             sStackIdentifier.acStackIdentifier[sStackIdentifier.u16StackPosition++]='.';
0042
memcpy(&sStackIdentifier.acStackIdentifier[sStackIdentifier.u16StackPosition],pcIdentifier,u16Size);
0043         sStackIdentifier.u16StackPosition+=u16Size;
0044         if(check)
0045         {
```

```
0046            /*uint16 stackPosition=   sStackIdentifier.u8CheckpointPosition>0?\
0047
sStackIdentifier.au16StackCheckpoints[sStackIdentifier.u8CheckpointPosition-1]:\
0048                                    0;
0049            sStackIdentifier.acStackIdentifier[sStackIdentifier.u16StackPosition]='\0';*/
0050            //vPrintf("\nDEBUG: Entering %s\n\n",&sStackIdentifier.acStackIdentifier[stackPosition]);
0051        }
0052        sStackIdentifier.au16StackSize[sStackIdentifier.u8IdentifierPosition++]=u16Size;
0053        return 1;
0054    }
0055    return 0;
0056 }
0057 uint16 u16StackPopIdentifier(char* pcIdentifier)
0058 {
0059    if(sStackIdentifier.u8IdentifierPosition>0)
0060    {
0061        uint16 u16IdentifierLength=sStackIdentifier.au16StackSize[--
sStackIdentifier.u8IdentifierPosition];
0062        memcpy( pcIdentifier, &sStackIdentifier.acStackIdentifier[sStackIdentifier.u16StackPosition-
u16IdentifierLength],
0063                u16IdentifierLength);
0064        sStackIdentifier.u16StackPosition-=u16IdentifierLength;
0065
if(sStackIdentifier.u16StackPosition<=sStackIdentifier.au16StackCheckpoints[sStackIdentifier.u8CheckpointPosi
tion-1])
0066        {
0067            /*uint16 stackPosition=   sStackIdentifier.u8CheckpointPosition>0?\
0068
sStackIdentifier.au16StackCheckpoints[sStackIdentifier.u8CheckpointPosition-1]:\
0069                                    0;*/
0070            u8StackUncheck();
0071
//sStackIdentifier.acStackIdentifier[sStackIdentifier.u16StackPosition+u16IdentifierLength]='\0';
0072            //vPrintf("\nDEBUG: Exeting %s\n\n",&sStackIdentifier.acStackIdentifier[stackPosition]);
0073        }
0074        if(sStackIdentifier.u8IdentifierPosition>0)
0075            sStackIdentifier.u16StackPosition--;
0076        return u16IdentifierLength;
0077    }
0078    return 0;
0079 }
0080 void vStackPopIdentifier(void)
0081 {
0082    if(sStackIdentifier.u8IdentifierPosition>0)
0083    {
0084        uint16 u16IdentifierLength=sStackIdentifier.au16StackSize[--
sStackIdentifier.u8IdentifierPosition];
0085        sStackIdentifier.u16StackPosition-=u16IdentifierLength;
0086
if(sStackIdentifier.u16StackPosition<=sStackIdentifier.au16StackCheckpoints[sStackIdentifier.u8CheckpointPosi
tion-1])
0087        {
0088            /*uint16 stackPosition=   sStackIdentifier.u8CheckpointPosition>0?\
0089
sStackIdentifier.au16StackCheckpoints[sStackIdentifier.u8CheckpointPosition-1]:\
0090                                    0;*/
0091            u8StackUncheck();
0092
//sStackIdentifier.acStackIdentifier[sStackIdentifier.u16StackPosition+u16IdentifierLength]='\0';
0093            //vPrintf("\nDEBUG: Exeting %s\n\n",&sStackIdentifier.acStackIdentifier[stackPosition]);
0094        }
0095        if(sStackIdentifier.u8IdentifierPosition>0)
0096            sStackIdentifier.u16StackPosition--;
0097    }
0098 }
```

```
0099
0100 void vStackPrintf(char* file,uint16 line,const char* information,...)
0101 {
0102     va_list ap;
0103     uint16 stackPosition=   sStackIdentifier.u8CheckpointPosition>0?\
0104                             sStackIdentifier.au16StackCheckpoints[sStackIdentifier.u8CheckpointPosition-
1]:\
0105                             0;
0106     sStackIdentifier.acStackIdentifier[sStackIdentifier.u16StackPosition]='\0';
0107     vPrintf("%s:",&sStackIdentifier.acStackIdentifier[stackPosition]);
0108     va_start(ap,information);
0109     vPrintf(information,ap);
0110     va_end(ap);
0111     vPrintf(" (%s: %d)\n",file,line);
0112 }
```

IX. Debugger.h

```
0001 #ifndef DEBUGER_H
0002 #define DEBUGER_H
0003
0004 #include <printf.h>
0005
0006 #define MAX_STACK_LENGTH           2048
0007 #define MAX_STACK_IDENTIFIERS      30
0008 #define MAX_CHECKPOINTS            10
0009
0010 typedef struct
0011 {
0012     char    acStackIdentifier[MAX_STACK_LENGTH];
0013     uint8   u8IdentifierPosition;
0014     uint8   u8CheckpointPosition;
0015     uint16  u16Pad1;
0016     uint16  u16StackPosition;
0017     uint16  u16Pad2;
0018     uint16  au16StackSize[MAX_STACK_IDENTIFIERS];
0019     uint16  au16StackCheckpoints[MAX_CHECKPOINTS];
0020 }DEB_stackIdentifier_s;
0021
0022 DEB_stackIdentifier_s sStackIdentifier;
0023
0024 PUBLIC void vStackStart(void);
0025
0026 PUBLIC uint8 u8StackPushIdentifier(char* pcIdentifier,uint16 u16Size,bool check);
0027 PUBLIC uint16 u16StackPopIdentifier(char* pcIdentifier);
0028 PUBLIC void vStackPopIdentifier(void);
0029
0030 PUBLIC void vStackPrintf(char* file,uint16 line,const char* information,...);
0031
0032 #endif
```

X. Mesh.c

```
0001 #include <string.h>
0002
0003 #include "mesh.h"
0004 #include "mhme.h"
0005 #include "MeshServices.h"
0006 #include "MeshControl.h"
0007
0008 #ifdef DEBUG
0009 #include "Debugger.h"
```

```
0010 #endif
0011
0012
0013 volatile MAC_McpsBuffer_s psMcpsBuffers[N_MCPS_BUFFERS];
0014
0015 volatile NET_MeshMhsduHandleQueue_s sMeshMhsduHandleQueue;
0016
0017 NET_meshMsgQueue_s  sMeshQueue;
0018
0019 extern volatile bool bJoinNetworkStatus;
0020
0021 PUBLIC void vNetApiMeshRequest(NET_MeshReq_s* psMeshReq, NET_MeshSyncCfm_s* psMeshSyncCfm)
0022 {
0023     //Register this function on the debugger
0024     #ifdef DEBUG
0025     u8StackPushIdentifier("MESH",strlen("MESH"),TRUE);
0026     u8StackPushIdentifier("vNetApiMeshRequest",strlen("vNetApiMeshRequest"),FALSE);
0027     #endif
0028
0029     switch(psMeshReq->u8Type)
0030     {
0031         case NET_MESH_REQ_DATA:
0032         {
0033             #ifdef DEBUG
0034             vStackPrintf(__FILE__,__LINE__,"Received NET_MESH_REQ_DATA.");
0035             #endif
0036
0037             if((bJoinNetworkStatus)&&(u8MhmeFlag!=MHME_LEAVING))
0038                 vHdlMeshReqData(&psMeshReq->uParam.sMeshReqData, psMeshSyncCfm);
0039         }break;
0040
0041         case NET_MESH_REQ_PURGE:
0042         {
0043             #ifdef DEBUG
0044             vStackPrintf(__FILE__,__LINE__,"Received NET_MESH_REQ_PURGE.");
0045             #endif
0046         }break;
0047
0048         default:
0049         {
0050             #ifdef DEBUG
0051             vStackPrintf(__FILE__,__LINE__,"Unrecognized request\\response.");
0052             #endif
0053         }
0054     }
0055     psMeshSyncCfm->u8Type=psMeshReq->u8Type;
0056
0057     //Unregister this function off the debugger
0058     #ifdef DEBUG
0059     vStackPopIdentifier(); //For the vNetApiMeshRequest
0060     vStackPopIdentifier(); //For the MESH
0061     #endif
0062 }
0063
0064
0065
0066
0067 uint16 u16MeshGetMessage(NET_MeshDcfmInd_s* psMeshDcfmInd)
0068 {
0069     uint16 u16ReturnValue=0;
0070
0071     //Register this function on the debugger
0072     #ifdef DEBUG
0073     u8StackPushIdentifier("MESH",strlen("MESH"),TRUE);
0074     u8StackPushIdentifier("u16MeshGetMessage",strlen("u16MeshGetMessage"),FALSE);
```

```
0075     #endif
0076
0077     //Test if there is any message on the queue
0078     if(sMeshQueue.u8MsgInQueue>0)
0079     {
0080         //Retreive the existing message from the queue and into the application buffer
0081         sMeshQueue.u8MsgInQueue--;
0082
0083         //Note: preprocessing may be needed before passing it to the application
0084         u16ReturnValue=u16MeshTranslateMessage( psMeshDcfmInd,\
0085
&sMeshQueue.asMeshEventQueue[(sMeshQueue.u8IndexOfFirstNonProcessedMsg++)%MAX_MSG_IN_QUEUE]);
0086     }
0087
0088     //Deregister this function off the debugger
0089     #ifdef DEBUG
0090     vStackPopIdentifier(); //For the u16MeshGetMessage
0091     vStackPopIdentifier(); //For the MESH
0092     #endif
0093
0094     //The return value is either 1 (if there is a message stored in the buffer) or 0 (if it is not)
0095     return u16ReturnValue;
0096 }
0097
0098
0099
0100
0101
0102
0103 uint16 u16MeshTranslateMessage(NET_MeshDcfmInd_s* psMeshDcfmInd,NET_MeshEvent_s* psMeshEvent)
0104 {
0105     uint16 u16ReturnValue=0;
0106
0107     //Register this function on the debugger
0108     #ifdef DEBUG
0109     u8StackPushIdentifier("u16MeshTranslateMessage",strlen("u16MeshTranslateMessage"),FALSE);
0110     #endif
0111
0112     //The preprocessing that needs to be done before the message be passed to the application
0113     switch(psMeshEvent->u8Type)
0114     {
0115         case NET_MESH_MCPS_EVENT:
0116         {
0117             #ifdef DEBUG
0118             vStackPrintf(__FILE__,__LINE__,"NET_MESH_MCPS_EVENT");
0119             #endif
0120             u16ReturnValue=u16MeshTranslateMcpsMessage(psMeshDcfmInd,&psMeshEvent->uParam.sMcpsDcfmInd);
0121         } break;
0122
0123         default:
0124         {
0125             #ifdef DEBUG
0126             vStackPrintf(__FILE__,__LINE__,"Unhandle message type.");
0127             #endif
0128             u16ReturnValue=0;
0129         }
0130     }
0131
0132     //Deregister this function off the debugger
0133     #ifdef DEBUG
0134     vStackPopIdentifier();
0135     #endif
0136
0137     //The return value is either 1 (if the message is ready to be returned to the application) or 0 (if
it is not)
```

```
0138     return u16ReturnValue;
0139 }
0140
0141
0142
0143
0144
0145 inline uint16 u16MeshTranslateMcpsMessage(NET_MeshDcfmInd_s* psMeshDcfmInd,MAC_McpsDcfmInd_s*
psMcpsDcfmInd)
0146 {
0147     uint16 u16ReturnValue=0;
0148
0149     //Register this function on the debugger
0150     #ifdef DEBUG
0151     u8StackPushIdentifier("u16MeshTranslateMcpsMessage",strlen("u16MeshTranslateMcpsMessage"),FALSE);
0152     #endif
0153
0154     switch(psMcpsDcfmInd->u8Type)
0155     {
0156         case MAC_MCPS_DCFM_DATA:
0157         {
0158             #ifdef DEBUG
0159             vStackPrintf(__FILE__,__LINE__,"Received MAC_MCPS_DCFM_DATA.");
0160             #endif
0161             u16ReturnValue=u16HdlMeshCfmData(psMeshDcfmInd,psMcpsDcfmInd);
0162
0163         }break;
0164
0165         case MAC_MCPS_IND_DATA:
0166         {
0167             #ifdef DEBUG
0168             vStackPrintf(__FILE__,__LINE__,"Received MAC_MCPS_IND_DATA.");
0169             #endif
0170             u16ReturnValue=u16HdlMeshIndData(psMeshDcfmInd,psMcpsDcfmInd);
0171
0172         }break;
0173
0174         /** . . . **/
0175
0176         default:
0177         {
0178             #ifdef DEBUG
0179             vStackPrintf(__FILE__,__LINE__,"Unhandle confirm/indication.");
0180             #endif
0181             u16ReturnValue=0;
0182         }
0183     }
0184
0185     //Deregister this function off the debugger
0186     #ifdef DEBUG
0187     vStackPopIdentifier();
0188     #endif
0189
0190     //The return value is either 1 (if the message is ready to be returned to the application) or 0 (if
it is not)
0191     return u16ReturnValue;
0192 }
0193
0194
0195
0196
0197
0198
0199 PUBLIC MAC_DcfmIndHdr_s *psMcpsDcfmIndGetBuf(void *pvParam)
0200 {
```

```
0201     uint8 i;
0202
0203     for(i=0; i<N_MCPS_BUFFERS; i++)
0204     {
0205         if(psMcpsBuffers[i].u8Used == FALSE)
0206         {
0207             psMcpsBuffers[i].u8Used = TRUE;
0208             return (MAC_DcfmIndHdr_s*)&psMcpsBuffers[i].sMcpsDcfmInd;
0209         }
0210     }
0211     return NULL;
0212 }
0213
0214
0215
0216
0217
0218
0219 PUBLIC void vMcpsDcfmIndPost(void *pvParam, MAC_DcfmIndHdr_s *psDcfmIndHdr)
0220 {
0221     int i;
0222     MAC_McpsDcfmInd_s* psMcpsInd=(MAC_McpsDcfmInd_s*)psDcfmIndHdr;
0223
0224     //Put the message in the message queue
0225     if(sMeshQueue.u8MsgInQueue<MAX_MSG_IN_QUEUE)
0226     {
0227         sMeshQueue.u8MsgInQueue++;
0228
sMeshQueue.asMeshEventQueue[(sMeshQueue.u8IndexOfFirstFreeSpot)%MAX_MSG_IN_QUEUE].u8Type=NET_MESH_MCPS_EVENT;
0229
memcpy(&sMeshQueue.asMeshEventQueue[(sMeshQueue.u8IndexOfFirstFreeSpot++)%MAX_MSG_IN_QUEUE].uParam.sMcpsDcfmI
nd,psMcpsInd,sizeof(MAC_McpsDcfmInd_s));
0230         #ifdef DEBUG
0231         vPrintf("MESH.vMcpsDcfmIndPost: Message received. (%s: %d)\n",__FILE__,__LINE__);
0232         #endif
0233     }
0234     else
0235     {
0236         #ifdef DEBUG
0237         vPrintf("MESH.vMcpsDcfmIndPost: To many messages in the message queue, so current message will
be ignored. (%s: %d)\n",__FILE__,__LINE__);
0238         #endif
0239     }
0240
0241     for(i=0;i<N_MCPS_BUFFERS;i++)
0242     {
0243         if(psMcpsInd==&psMcpsBuffers[i].sMcpsDcfmInd)
0244         {
0245             psMcpsBuffers[i].u8Used=FALSE;
0246             #ifdef DEBUG
0247             vPrintf("MESH.vMcpsDcfmIndPost: BUFFER FREED!!!!!!! (%s: %d)\n",__FILE__,__LINE__);
0248             #endif
0249             break;
0250         }
0251     }
0252 }
```

XI. mesh.h

```
0001 #ifndef _MESH_H
0002 #define _MESH_H
0003
0004 #include <jendefs.h>
0005 #include <mac_sap.h>
```

```
0006
0007
0008 typedef enum
0009 {
0010
0011     NET_ENUM_SUCCESS,
0012     NET_ENUM_TRANSACTION_OVERFLOW,
0013     NET_ENUM_TRANSACTION_EXPIRED,
0014     NET_ENUM_CHANNEL_ACCESS_FAILURE,
0015     NET_ENUM_NO_ACK,
0016     NET_ENUM_UNAVAILABLE_KEY,
0017     NET_ENUM_FRAME_TOO_LONG,
0018     NET_ENUM_INVALID_PARAMETER
0019
0020 }NET_Enum_e;
0021
0022 typedef enum
0023 {
0024
0025     NET_MESH_REQ_DATA,
0026     NET_MESH_REQ_PURGE
0027
0028 }NET_MeshReqRspType_e;
0029
0030 typedef enum
0031 {
0032
0033     NET_MESH_DCFM_DATA,
0034     NET_MESH_IND_DATA
0035
0036 }NET_MeshDcmfIndType_e;
0037
0038
0039
0040 typedef struct
0041 {
0042
0043     uint8           u8MhsduHandle;  /**< The handle associated with the MHSDU being confirmed. */
0044     NET_Enum_e      eStatus;        /**< The status of the last MHSDUtransmission. */
0045     uint16          u16Pad;
0046
0047 }NET_MeshCfmData_s;
0048
0049
0050 typedef struct
0051 {
0052
0053     uint8           u8SrcAddrMode;  /**< The source addressing mode for this primitive corresponding to
the received MHPDU. */
0054     uint8           u8MhsduLength;  /**< The number of octets contained in the MHSDU being indicated by
the MESH sublayer entity. */
0055     uint8           au8Mhsdu[MAC_MAX_DATA_PAYLOAD_LEN];  /**< The set of octets forming the MHSDU being
indicated by the MESH sublayer entity. */
0056     uint8           u8Pad;
0057     uint16          u16SrcPANId;    /**< The 16-bit PAN identifier of the entity from which the MHSDU
was received. */
0058     uint16          u16Pad;
0059     MAC_Addr_u      uSrcAddr;       /**< The individual device address of the entity from which the
MHSDU was received. */
0060
0061 }NET_MeshIndData_s;
0062
0063
0064 typedef union
0065 {
```

```
0066
0067     NET_MeshCfmData_s    sMeshCfmData;
0068     NET_MeshIndData_s    sMeshIndData;
0069
0070 }NET_MeshDcfmIndParam_u;
0071
0072
0073 typedef struct
0074 {
0075
0076     uint8                    u8Type;
0077     uint8                    u8Pad;
0078     uint16                   u16Pad;
0079     NET_MeshDcfmIndParam_u   uParam;
0080
0081 }NET_MeshDcfmInd_s;
0082
0083
0084 typedef struct
0085 {
0086     uint8           u8SrcAddrMode;      /**< The source addressing mode for this primitive and
subsequent MHPDU. */
0087     uint8           u8DstAddrMode;      /**< The destination addressing mode for this primitive and
subsequent MHPDU. */
0088     uint8           u8MhsduLength;      /**< The number of octets contained in the MHSDU to be
transmitted by the mesh sublayer entity. */
0089     uint8           u8MhsduHandle;      /**< The handle associated with the MHSDU to be transmitted by
the mesh sublayer entity. */
0090     uint8           u8AckTransmission;  /**< This field is set to TRUE if an acknowledgement is required
from the receiver; otherwise, it is set to FALSE. */
0091     uint8           u8McstTransmission; /**< This field is set to TRUE if the data is to be multicast;
otherwise, it is set to FALSE. */
0092     uint8           u8BcstTransmission; /**< This field is set to TRUE if the data is to be broadcast;
otherwise, it is set to FALSE. */
0093     uint8           u8ReliableBcst;     /**< This field is set to TRUE if reliable broadcast is
required; otherwise, it is set to FALSE. It is meaningful only when BcstTransmission is set to TRUE. */
0094     uint8           pu8Mhsdu[MAC_MAX_DATA_PAYLOAD_LEN];           /**< The set of octets forming the
MHSDU to be transmitted by the mesh sublayer entity. */
0095     MAC_Addr_u      uDstAddr;           /**< The device address of the entity, or entities in the case
of multicast and broadcast, to which the MHSDU is being transferred. */
0096 }NET_MeshReqData_s;
0097
0098
0099 typedef union
0100 {
0101
0102     NET_MeshReqData_s    sMeshReqData;
0103
0104 }NET_MeshReqParam_u;
0105
0106
0107 typedef struct
0108 {
0109
0110     uint8                u8Type;
0111     uint8                u8Pad;
0112     uint16               u16Pad;
0113     NET_MeshReqParam_u   uParam;
0114
0115 }NET_MeshReq_s;
0116
0117
0118 typedef union
0119 {
0120
```

```
0121     NET_MeshCfmData_s    sMeshCfmData;
0122
0123 }NET_MeshSyncCfmParam_u;
0124
0125
0126 typedef struct
0127 {
0128
0129     uint8                   u8Type;
0130     uint8                   u8Pad;
0131     uint16                  u16Pad;
0132     NET_MeshSyncCfmParam_u  uParam;
0133
0134 }NET_MeshSyncCfm_s;
0135
0136 typedef struct
0137 {
0138
0139     uint8   u8Used;
0140     uint8   u8Pad;
0141     uint16  u16Pad;
0142     MAC_McpsDcfmInd_s sMcpsDcfmInd;
0143
0144 }MAC_McpsBuffer_s;
0145
0146 #define MAX_HND_IN_QUEUE 64
0147
0148 typedef enum
0149 {
0150     NET_MESH_MSDU_HANDLE_DATA,
0151     NET_MESH_MSDU_HANDLE_COMMAND,
0152     NET_MESH_MSDU_HANDLE_FREE
0153 }
0154 NET_MeshMsduHandleType_e;
0155
0156 typedef struct
0157 {
0158     uint8   u8Type;
0159     uint8   u8MhsduHandle;
0160     uint16  u16Pad;
0161 }
0162 NET_MeshMsduHandle_s;
0163
0164
0165 typedef struct
0166 {
0167
0168     uint8                   u8HndInQueue;
0169     uint8                   u8IndexOfFirstFreeSpot;
0170     uint16                  u16Pad;
0171     uint64                  u64MhsduHandleStatus;
0172     NET_MeshMsduHandle_s    asMhsduHandle[MAX_HND_IN_QUEUE];
0173 }
0174 NET_MeshMhsduHandleQueue_s;
0175
0176 #define MAX_MSG_IN_QUEUE 64
0177
0178 typedef enum
0179 {
0180     NET_MESH_MCPS_EVENT
0181 }
0182 NET_MeshEventType_e;
0183
0184 typedef union
0185 {
```

```
0186     MAC_McpsDcfmInd_s    sMcpsDcfmInd;
0187 }
0188 NET_MeshEvent_u;
0189
0190 typedef struct
0191 {
0192     uint8               u8Type;
0193     uint8               u8Pad;
0194     uint16              u16Pad;
0195     NET_MeshEvent_u     uParam;
0196 }
0197 NET_MeshEvent_s;
0198
0199 typedef struct
0200 {
0201     uint8               u8MsgInQueue;
0202     uint8               u8IndexOfFirstNonProcessedMsg;
0203     uint8               u8IndexOfFirstFreeSpot;
0204     uint8               u8Pad;
0205     NET_MeshEvent_s     asMeshEventQueue[MAX_MSG_IN_QUEUE];
0206 }NET_meshMsgQueue_s;
0207
0208 PUBLIC MAC_DcfmIndHdr_s *psMcpsDcfmIndGetBuf(void *pvParam);   /**< Provides a buffer to the MCPS, used
to send deferred confirms and indications to the Network Layer (this application) (does this include hardware
interrupts ?) */
0209 PUBLIC void vMcpsDcfmIndPost(void *pvParam, MAC_DcfmIndHdr_s *psDcfmIndHdr);    /**< Called by the MCPS
to send a confirm or indication to the network layer. It runs in the ISR context. */
0210
0211
0212 /** Number of available MCPS buffers **/
0213 #define N_MCPS_BUFFERS  3
0214
0215 PUBLIC void vNetApiMeshRequest(NET_MeshReq_s* sMeshReq, NET_MeshSyncCfm_s *sMeshSyncCfm);
0216
0217 uint16 u16MeshTranslateMessage(NET_MeshDcfmInd_s* psMeshDcfmInd,NET_MeshEvent_s* psMeshEvent);
0218 uint16 u16MeshGetMessage(NET_MeshDcfmInd_s* psMeshDcfmInd);
0219
0220 #endif //_MESH_H
```

XII. MeshControl.c

```
0001 #include "Mesh.h"
0002 #include "DataFrames.h"
0003 #include "MeshServices.h"
0004 #include "Mhme.h"
0005
0006 #include "string.h"
0007
0008 #ifdef DEBUG
0009 #include "Debugger.h"
0010 #endif
0011
0012 #define SAME64ADDR(a,b) ((a.u32H==b.u32H)&&(a.u32L==b.u32L))
0013
0014 extern NET_MeshInfo_s sMeshInfo;
0015 extern NET_MeshData_s sMeshData;
0016 extern NET_MeshMhsduHandleQueue_s sMeshMhsduHandleQueue;
0017
0018
0019 inline uint16 u16HdlMeshCfmData(NET_MeshDcfmInd_s* psMeshDcfmInd,MAC_McpsDcfmInd_s* psMcpsDcfmInd)
0020 {
0021     uint16 u8Handle;
0022     uint16 u16ReturnValue=0;
0023
```

```
0024      //Register this function on the debugger
0025      #ifdef DEBUG
0026      u8StackPushIdentifier("u16HdlMeshCfmData",strlen("u16HdlMeshCfmData"),FALSE);
0027      #endif
0028
0029      //Retreive handle that is associated with the confirm message
0030      u8Handle=psMcpsDcfmInd->uParam.sDcfmData.u8Handle;
0031
0032      #ifdef DEBUG
0033      vPrintf("USED HANDLE=%d\n",u8Handle);
0034      #endif
0035
0036      //Validate the retrieved handle
0037      if(bValidateMhsduHandle(u8Handle))
0038      {
0039          #ifdef DEBUG
0040          vPrintf("HANDLE %d WAS VALIDATED\n",u8Handle);
0041          #endif
0042
0043          //Filling the defered confirm structure
0044          psMeshDcfmInd->u8Type=NET_MESH_DCFM_DATA;
0045          psMeshDcfmInd-
>uParam.sMeshCfmData.u8MhsduHandle=sMeshMhsduHandleQueue.asMhsduHandle[u8Handle].u8MhsduHandle;
0046
0047          //Filling the status field
0048          switch(psMcpsDcfmInd->uParam.sDcfmData.u8Status)
0049          {
0050              case MAC_ENUM_SUCCESS:
0051              {
0052                  #ifdef DEBUG
0053                  vStackPrintf(__FILE__,__LINE__,"Data frame sent with success\n");
0054                  #endif
0055                  psMeshDcfmInd->uParam.sMeshCfmData.eStatus=NET_ENUM_SUCCESS;
0056              }break;
0057
0058              case MAC_ENUM_TRANSACTION_OVERFLOW:
0059              {
0060                  #ifdef DEBUG
0061                  vStackPrintf(__FILE__,__LINE__,"Overflow occured during transaction\n");
0062                  #endif
0063                  psMeshDcfmInd->uParam.sMeshCfmData.eStatus=NET_ENUM_TRANSACTION_OVERFLOW;
0064              }break;
0065
0066              case MAC_ENUM_TRANSACTION_EXPIRED:
0067              {
0068                  #ifdef DEBUG
0069                  vStackPrintf(__FILE__,__LINE__,"Time for transaction expired\n");
0070                  #endif
0071                  psMeshDcfmInd->uParam.sMeshCfmData.eStatus=NET_ENUM_TRANSACTION_EXPIRED;
0072              }break;
0073
0074              case MAC_ENUM_CHANNEL_ACCESS_FAILURE:
0075              {
0076                  #ifdef DEBUG
0077                  vStackPrintf(__FILE__,__LINE__,"Overflow occured during transaction\n");
0078                  #endif
0079                  psMeshDcfmInd->uParam.sMeshCfmData.eStatus=NET_ENUM_CHANNEL_ACCESS_FAILURE;
0080              }break;
0081
0082              case MAC_ENUM_NO_ACK:
0083              {
0084                  #ifdef DEBUG
0085                  vStackPrintf(__FILE__,__LINE__,"No ACK received\n");
0086                  #endif
0087                  psMeshDcfmInd->uParam.sMeshCfmData.eStatus=NET_ENUM_NO_ACK;
```

```
0088            }break;
0089
0090            case MAC_ENUM_UNAVAILABLE_KEY:
0091            {
0092                #ifdef DEBUG
0093                vStackPrintf(__FILE__,__LINE__,"Unavailable key\n");
0094                #endif
0095                psMeshDcfmInd->uParam.sMeshCfmData.eStatus=NET_ENUM_UNAVAILABLE_KEY;
0096            }break;
0097
0098            case MAC_ENUM_FRAME_TOO_LONG:
0099            {
0100                #ifdef DEBUG
0101                vStackPrintf(__FILE__,__LINE__,"Frame too long\n");
0102                #endif
0103                psMeshDcfmInd->uParam.sMeshCfmData.eStatus=NET_ENUM_FRAME_TOO_LONG;
0104            }break;
0105
0106            case MAC_ENUM_INVALID_PARAMETER:
0107            {
0108                #ifdef DEBUG
0109                vStackPrintf(__FILE__,__LINE__,"Invalide Parameter\n");
0110                #endif
0111                psMeshDcfmInd->uParam.sMeshCfmData.eStatus=NET_ENUM_INVALID_PARAMETER;
0112            }break;
0113        }
0114
0115        //Decide if the confirm message is to be sent to the mhme or the application
0116        if(sMeshMhsduHandleQueue.asMhsduHandle[u8Handle].u8Type==NET_MESH_MSDU_HANDLE_COMMAND)
0117            vMhmeCommandPost(psMeshDcfmInd);
0118        else
0119            u16ReturnValue=1;
0120
0121        //Freing the used handle
0122        vEraseMhsduHandle(u8Handle);
0123    }
0124
0125    //Deregister this function off the debugger
0126    #ifdef DEBUG
0127    vStackPopIdentifier();
0128    #endif
0129
0130    return u16ReturnValue;
0131 }
0132
0133 inline uint16 u16HdlMeshIndData(NET_MeshDcfmInd_s* psMeshDcfmInd,MAC_McpsDcfmInd_s* psMcpsDcfmInd)
0134 {
0135    uint16              u16ReturnValue=0;
0136    NET_DataDcfmInd_s   sDataDcfmInd;
0137
0138
0139
0140    //Register this function on the debugger
0141    #ifdef DEBUG
0142    u8StackPushIdentifier("u16HdlMeshIndData",strlen("u16HdlMeshIndData"),FALSE);
0143    #endif
0144
0145    //Get the received message and see if it is a command frame
0146    vNetApiDataTranslate(&sDataDcfmInd,psMcpsDcfmInd);
0147
0148    #ifdef DEBUG
0149    vPrintf("Addresses:\n\tSource: %d (%x %x)\n\tDestination: %d (%x %x)\n",
0150        sDataDcfmInd.sDataInd.sDataFrameData.u8SrcAddrMode,
0151        sMeshInfo.sAddressMapping.sExtAddr.u32H,
0152        sMeshInfo.sAddressMapping.sExtAddr.u32L,
```

```
0153          sDataDcfmInd.sDataInd.sDataFrameData.u8DstAddrMode,
0154          sDataDcfmInd.sDataInd.sDataFrameData.uDstAddr.sExt.u32H,
0155          sDataDcfmInd.sDataInd.sDataFrameData.uDstAddr.sExt.u32L);
0156      vPrintf("Broadcast? %d\n",sDataDcfmInd.sDataInd.sDataFrameData.u8BcstTransmission);
0157      #endif
0158
0159
0160      //Validating destination
0161      if( (sDataDcfmInd.sDataInd.sDataFrameData.u8BcstTransmission==TRUE)||
0162          ((sDataDcfmInd.sDataInd.sDataFrameData.u8DstAddrMode==2)?
0163              (sMeshInfo.u16NetworkAddress==sDataDcfmInd.sDataInd.sDataFrameData.uDstAddr.u16Short):
0164
(SAME64ADDR(sMeshInfo.sAddressMapping.sExtAddr,sDataDcfmInd.sDataInd.sDataFrameData.uDstAddr.sExt))))
0165      {
0166          #ifdef DEBUG
0167          vPrintf("It is a message for me\n");
0168          #endif
0169
0170          //Filling the mesh deferred comfirm indication
0171          psMeshDcfmInd->u8Type=NET_MESH_IND_DATA;
0172          psMeshDcfmInd-
>uParam.sMeshIndData.u8SrcAddrMode=sDataDcfmInd.sDataInd.sDataFrameData.u8SrcAddrMode;
0173          psMeshDcfmInd->uParam.sMeshIndData.u16SrcPANId=sDataDcfmInd.sDataInd.sDataFrameData.u16SrcPANId;
0174          memcpy(&psMeshDcfmInd-
>uParam.sMeshIndData.uSrcAddr,&sDataDcfmInd.sDataInd.sDataFrameData.uSrcAddr,sizeof(MAC_Addr_u));
0175          psMeshDcfmInd-
>uParam.sMeshIndData.u8MhsduLength=sDataDcfmInd.sDataInd.sDataFrameData.u8MhsduLength;
0176          memcpy(psMeshDcfmInd-
>uParam.sMeshIndData.au8Mhsdu,sDataDcfmInd.sDataInd.sDataFrameData.au8Mhsdu,
0177                  sDataDcfmInd.sDataInd.sDataFrameData.u8MhsduLength);
0178
0179          //Sending to the higher layers
0180          if(sDataDcfmInd.u8Type==NET_DATA_IND_COMM)
0181              vMhmeCommandPost(psMeshDcfmInd);
0182          else
0183              u16ReturnValue=1;
0184      }
0185
0186      //Resending to the appropriated device if it was a data frame
0187      else if(sDataDcfmInd.u8Type==NET_DATA_IND_DATA)
0188      {
0189          NET_DataReqRsp_s  sDataReqRsp;
0190          NET_DataSyncCfm_s sDataSyncCfm;
0191          MAC_ExtAddr_s     sExt;
0192
0193          #ifdef DEBUG
0194          vPrintf("It is not a message for me and it was a data frame. Forwarding the message to the
correct neighbor\n");
0195          #endif
0196
0197          //Sending to the device
0198          switch (sDataDcfmInd.sDataInd.sDataFrameData.u8DstAddrMode)
0199          {
0200              case(0x03):
0201              {
0202                  //Should I check if it is a neighbour?
0203                  memcpy(&sExt,&sDataDcfmInd.sDataInd.sDataFrameData.uDstAddr.sExt,sizeof(MAC_ExtAddr_s));
0204              }break;
0205
0206              case(0x02):
0207              {
0208                  //Obtains destination MAC address
0209                  vTranslateAddress(sDataDcfmInd.sDataInd.sDataFrameData.uDstAddr.u16Short,&sExt);
0210              }break;
0211
```

```
0212             default:
0213             {
0214                 #ifdef DEBUG
0215                 vPrintf("Error! Reserved Mode");
0216                 #endif
0217             }
0218         }
0219
0220         //Filling the data request structure
0221         sDataReqRsp.u8Type=NET_DATA_IND_DATA;
0222         sDataReqRsp.u8MhsduHandle=sMeshData.u8MeshHandle++;;
0223         memcpy(&sDataReqRsp.sExt,&sExt,sizeof(MAC_ExtAddr_s));
0224         sDataReqRsp.sDataFrameData.u16SrcPANId=sMeshInfo.u16PanId;
0225         sDataReqRsp.sDataFrameData.u8SrcAddrMode=sDataDcfmInd.sDataInd.sDataFrameData.u8SrcAddrMode;
0226         sDataReqRsp.sDataFrameData.u8DstAddrMode=sDataDcfmInd.sDataInd.sDataFrameData.u8DstAddrMode;
0227         sDataDcfmInd.sDataInd.sDataFrameData.u8SrcAddrMode==2?
0228
memcpy(&sDataReqRsp.sDataFrameData.uSrcAddr.u16Short,&sDataDcfmInd.sDataInd.sDataFrameData.uSrcAddr.u16Short,
sizeof(uint16)):
0229
memcpy(&sDataReqRsp.sDataFrameData.uSrcAddr.sExt,&sDataDcfmInd.sDataInd.sDataFrameData.uSrcAddr.sExt,sizeof(M
AC_ExtAddr_s));
0230         sDataDcfmInd.sDataInd.sDataFrameData.u8DstAddrMode==2?
0231
memcpy(&sDataReqRsp.sDataFrameData.uDstAddr.u16Short,&sDataDcfmInd.sDataInd.sDataFrameData.uDstAddr.u16Short,
sizeof(uint16)):
0232
memcpy(&sDataReqRsp.sDataFrameData.uDstAddr.sExt,&sDataDcfmInd.sDataInd.sDataFrameData.uDstAddr.sExt,sizeof(M
AC_ExtAddr_s));
0233         sDataReqRsp.sDataFrameData.u8MhsduLength=sDataDcfmInd.sDataInd.sDataFrameData.u8MhsduLength;
0234
memcpy(sDataReqRsp.sDataFrameData.au8Mhsdu,sDataDcfmInd.sDataInd.sDataFrameData.au8Mhsdu,sDataDcfmInd.sDataIn
d.sDataFrameData.u8MhsduLength);
0235
sDataReqRsp.sDataFrameData.u8AckTransmission=sDataDcfmInd.sDataInd.sDataFrameData.u8AckTransmission;
0236
sDataReqRsp.sDataFrameData.u8McstTransmission=sDataDcfmInd.sDataInd.sDataFrameData.u8McstTransmission;
0237
sDataReqRsp.sDataFrameData.u8BcstTransmission=sDataDcfmInd.sDataInd.sDataFrameData.u8BcstTransmission;
0238         sDataReqRsp.sDataFrameData.u8ReliableBcst=sDataDcfmInd.sDataInd.sDataFrameData.u8ReliableBcst;
0239
0240         //Send the data frame
0241         vNetApiDataRequest(&sDataReqRsp,&sDataSyncCfm);
0242
0243         ///Process Confirm
0244
0245     }
0246
0247     //Deregister this function off the debugger
0248     #ifdef DEBUG
0249     vStackPopIdentifier();
0250     #endif
0251
0252     return u16ReturnValue;
0253 }
```

XIII. MeshControl.h

```
0001 #ifndef MESH_CONTROL
0002 #define MESH_CONTROL
0003
0004 #include "Mesh.h"
0005
0006 //To translate messages to mhme messages and to perform the process needed on those messages
```

```
0007 inline uint16 u16MeshTranslateMcpsMessage(NET_MeshDcfmInd_s* psMeshDcfmInd,MAC_McpsDcfmInd_s*
psMcpsDcfmInd);
0008
0009 //Individual processing of each message
0010 inline uint16 u16HdlMeshCfmData(NET_MeshDcfmInd_s* psMeshDcfmInd,MAC_McpsDcfmInd_s* psMcpsDcfmInd);
0011 inline uint16 u16HdlMeshIndData(NET_MeshDcfmInd_s* psMeshDcfmInd,MAC_McpsDcfmInd_s* psMcpsDcfmInd);
0012
0013 #endif //MESH_CONTROL
```

XIV. MeshServices.c

```
0001
0002 #include "mesh.h"
0003 #include "mhme.h"
0004 #include "MeshServices.h"
0005 #include "DataFrames.h"
0006
0007 #include "Debugger.h"
0008
0009 #include "string.h"
0010
0011 extern NET_MeshInfo_s sMeshInfo;
0012 extern NET_MeshMhsduHandleQueue_s sMeshMhsduHandleQueue;
0013 extern NET_MeshData_s sMeshData;
0014
0015 void InitializeMhsduHandleQueue(void)
0016 {
0017     memset(&sMeshMhsduHandleQueue.u64MhsduHandleStatus,0,sizeof(uint64));
0018     sMeshMhsduHandleQueue.u8IndexOfFirstFreeSpot=0;
0019     sMeshMhsduHandleQueue.u8HndInQueue=0;
0020 }
0021
0022
0023 int16 i16InsertMhsduHandle(uint8 u8MhsduHandle,uint8 u8Type)
0024 {
0025     if (sMeshMhsduHandleQueue.u8HndInQueue<MAX_HND_IN_QUEUE)
0026     {
0027         uint64 u64Status=sMeshMhsduHandleQueue.u64MhsduHandleStatus;
0028         uint16  ret=sMeshMhsduHandleQueue.u8IndexOfFirstFreeSpot;
0029         uint64 u64StatusHp=1;
0030         uint8   i;
0031
0032
0033         sMeshMhsduHandleQueue.asMhsduHandle[ret].u8Type=u8Type;
0034         sMeshMhsduHandleQueue.asMhsduHandle[ret].u8MhsduHandle=u8MhsduHandle;
0035         u64StatusHp=u64StatusHp<<((MAX_HND_IN_QUEUE-1)-ret);
0036         u64Status|=u64StatusHp;
0037         sMeshMhsduHandleQueue.u8HndInQueue++;
0038         for (i=sMeshMhsduHandleQueue.u8IndexOfFirstFreeSpot;i<MAX_HND_IN_QUEUE;i++)
0039         {
0040             if (!(u64StatusHp&u64Status))
0041             {
0042                 sMeshMhsduHandleQueue.u8IndexOfFirstFreeSpot=i;
0043                 break;
0044             }
0045             u64StatusHp=u64StatusHp>>1;
0046         }
0047         sMeshMhsduHandleQueue.u64MhsduHandleStatus=u64Status;
0048         return ret;
0049     }
0050     return -1;
0051 }
0052
0053
```

```
0054 bool bValidateMhsduHandle(uint8 u8MhsduHandle)
0055 {
0056     uint64 u64Status=sMeshMhsduHandleQueue.u64MhsduHandleStatus;
0057     uint64 u64StatusHp=1;
0058 
0059     if (u8MhsduHandle>=MAX_HND_IN_QUEUE) return FALSE;
0060     u64StatusHp=u64StatusHp<<((MAX_HND_IN_QUEUE-1)-u8MhsduHandle);
0061     if (!(u64StatusHp&u64Status)) return FALSE;
0062     return TRUE;
0063 }
0064 
0065 void vEraseMhsduHandle(uint8 u8MhsduHandle)
0066 {
0067     if (u8MhsduHandle<MAX_HND_IN_QUEUE)
0068     {
0069         uint64 u64Status=sMeshMhsduHandleQueue.u64MhsduHandleStatus;
0070         uint64 u64StatusHp=1;
0071 
0072         u64StatusHp=u64StatusHp<<((MAX_HND_IN_QUEUE-1)-u8MhsduHandle);
0073         u64StatusHp^=u64Status;
0074         u64Status&=u64StatusHp;
0075         sMeshMhsduHandleQueue.u64MhsduHandleStatus=u64Status;
0076         sMeshMhsduHandleQueue.u8HndInQueue--;
0077 
sMeshMhsduHandleQueue.u8IndexOfFirstFreeSpot=(u8MhsduHandle<sMeshMhsduHandleQueue.u8IndexOfFirstFreeSpot)?\
0078                 u8MhsduHandle:\
0079                 sMeshMhsduHandleQueue.u8IndexOfFirstFreeSpot;
0080     }
0081 }
0082 
0083 
0084 void vTranslateAddress(uint16 u16Short, MAC_ExtAddr_s* psExt)
0085 {
0086     bool   bOneHopNeighbor=FALSE;
0087     uint8  u8NumberOfOps=255;
0088     uint8  u8NeighborNumber;
0089     uint8  u8TreeLevel=255;
0090     uint16 i;
0091     //Check the neighbour list to verify if it is an one hop neighbour
0092     for (i=0;i<MAX_NEIGHBORS;i++)
0093     {
0094         if ((sMeshInfo.psNeighborList[i].u16BeginningAddress==u16Short) &&
0095                 (sMeshInfo.psNeighborList[i].u8NumberOfOps==1))
0096         {
0097             //Put the 64 bit address
0098             #ifdef DEBUG
0099             vPrintf("\n\n\nTranslateAddr: The device is my neighbor!!!!\n");
0100             #endif
0101             memcpy(psExt,&sMeshInfo.psNeighborList[i].sExt,sizeof(MAC_ExtAddr_s));
0102             bOneHopNeighbor=TRUE;
0103             break;
0104         }
0105 
0106     }
0107     #ifdef DEBUG
0108     vPrintf("bOneHopNeighbor=%d\n",bOneHopNeighbor);
0109     #endif
0110 
0111     //if search failed, then select a path to it
0112     if (!bOneHopNeighbor)
0113     {
0114         bool    bTemp=FALSE;
0115         uint16  length=0xFFFF;
0116 
0117         #ifdef DEBUG
```

```
0118          vPrintf("\n\n\nTranslateAddr: Searching for an one hop neighbour!!!!\n");
0119          #endif
0120
0121          //dst falls in my Nbrs/Children addr block
0122          for (i=0;i<MAX_NEIGHBORS; i++)
0123          {
0124              if ((u16Short>sMeshInfo.psNeighborList[i].u16BeginningAddress) &&
0125                  (u16Short<=sMeshInfo.psNeighborList[i].u16EndingAddress)   &&
0126                  (sMeshInfo.psNeighborList[i].u8Relationship!=NO_RELATIONSHIP))
0127              {
0128                  #ifdef DEBUG
0129                  vPrintf("\n\n\nTranslateAddr: Found a neighbour [%d] with the following address block:
%d to
%d!!!!\n",i,sMeshInfo.psNeighborList[i].u16BeginningAddress,u16Short<=sMeshInfo.psNeighborList[i].u16EndingAd
dress);
0130                  #endif
0131                  if((sMeshInfo.psNeighborList[i].u16EndingAddress-
sMeshInfo.psNeighborList[i].u16BeginningAddress)<length)
0132                  {
0133                      length=sMeshInfo.psNeighborList[i].u16BeginningAddress -
sMeshInfo.psNeighborList[i].u16EndingAddress;
0134                      bTemp=TRUE;
0135                      u8NeighborNumber=i;
0136                  }
0137              }
0138          }
0139
0140
0141          if (!bTemp)
0142          {
0143              //Get node with smallest tree level + number of hops
0144              for (i=0;i<MAX_NEIGHBORS;i++)
0145              {
0146                  if (sMeshInfo.psNeighborList[i].u8TreeLevel<u8TreeLevel &&
sMeshInfo.psNeighborList[i].u8Relationship!=NO_RELATIONSHIP)
0147                  {
0148                      u8TreeLevel = sMeshInfo.psNeighborList[i].u8TreeLevel;
0149                      if (sMeshInfo.psNeighborList[i].u8NumberOfOps<u8NumberOfOps)
0150                      {
0151                          u8NumberOfOps=sMeshInfo.psNeighborList[i].u8NumberOfOps;
0152                          u8NeighborNumber=i;
0153                      }
0154                      #ifdef DEBUG
0155                      vPrintf("\n\n\nTranslateAddr: Found a neighbour [%d] with the following tree level:
%d!!!!\n",i,sMeshInfo.psNeighborList[i].u8TreeLevel);
0156                      vPrintf("TranslateAddr: Neigbor has %d hops to dst!!!!\nNb %d smallest has %d hops
to DST",sMeshInfo.psNeighborList[i].u8NumberOfOps,u8NeighborNumber,u8NumberOfOps);
0157                      #endif
0158
0159                  }
0160              }
0161          }
0162
0163          //call get onehopneighbor
0164          vGetOneHopNeighbor(&u8NumberOfOps,&u8NeighborNumber);
0165          //found one hop neighbor!
0166          #ifdef DEBUG
0167          vPrintf("SENDING:\n\tNb: %d\n\tHops:%d\n\n",u8NeighborNumber,u8NumberOfOps);
0168          #endif
0169          memcpy(psExt,&sMeshInfo.psNeighborList[u8NeighborNumber].sExt,sizeof(MAC_ExtAddr_s));
0170          //psMeshReqData.uDstAddr.u16Short=sMeshInfo.psNeighborList[u8NeighborNumber].u16NetworkAddress;
0171      }
0172
0173 }
0174
```

```
0175 inline void vHdlMeshReqData(NET_MeshReqData_s *psMeshReqData, NET_MeshSyncCfm_s *psMeshSyncCfm)
0176 {
0177     NET_DataReqRsp_s    sDataReqRsp;
0178     NET_DataSyncCfm_s   sDataSyncCfm;
0179     MAC_ExtAddr_s       sExt;
0180
0181     //Verify the addr field
0182     switch (psMeshReqData->u8DstAddrMode)
0183     {
0184         case(0x03):
0185         {
0186             //Should I check if it is a neighbour?
0187             memcpy(&sExt,&psMeshReqData->uDstAddr.sExt,sizeof(MAC_ExtAddr_s));
0188         }break;
0189
0190         case(0x02):
0191         {
0192             //Obtains destination MAC address
0193             vTranslateAddress(psMeshReqData->uDstAddr.u16Short,&sExt);
0194         }break;
0195
0196         default:
0197         {
0198             #ifdef DEBUG
0199             vPrintf("Error! Reserved Mode");
0200             #endif
0201         }
0202     }
0203
0204     //Filling the data request structure
0205     sDataReqRsp.u8Type=NET_DATA_IND_DATA;
0206     sDataReqRsp.u8MhsduHandle=psMeshReqData->u8MhsduHandle;
0207     memcpy(&sDataReqRsp.sExt,&sExt,sizeof(MAC_ExtAddr_s));
0208     sDataReqRsp.sDataFrameData.u16SrcPANId=sMeshInfo.u16PanId;
0209     sDataReqRsp.sDataFrameData.u8SrcAddrMode=psMeshReqData->u8SrcAddrMode;
0210     sDataReqRsp.sDataFrameData.u8DstAddrMode=psMeshReqData->u8DstAddrMode;
0211     psMeshReqData->u8SrcAddrMode==2?
0212     memcpy(&sDataReqRsp.sDataFrameData.uSrcAddr.u16Short,&sMeshInfo.u16NetworkAddress,sizeof(uint16)):
0213
memcpy(&sDataReqRsp.sDataFrameData.uSrcAddr.sExt,&sMeshInfo.sAddressMapping.sExtAddr,sizeof(MAC_ExtAddr_s));
0214     psMeshReqData->u8DstAddrMode==2?
0215     memcpy(&sDataReqRsp.sDataFrameData.uDstAddr.u16Short,&psMeshReqData-
>uDstAddr.u16Short,sizeof(uint16)):
0216     memcpy(&sDataReqRsp.sDataFrameData.uDstAddr.sExt,&psMeshReqData-
>uDstAddr.sExt,sizeof(MAC_ExtAddr_s));
0217     sDataReqRsp.sDataFrameData.u8MhsduLength=psMeshReqData->u8MhsduLength;
0218     memcpy(sDataReqRsp.sDataFrameData.au8Mhsdu,psMeshReqData->pu8Mhsdu,psMeshReqData->u8MhsduLength);
0219     sDataReqRsp.sDataFrameData.u8AckTransmission=psMeshReqData->u8AckTransmission;
0220     sDataReqRsp.sDataFrameData.u8McstTransmission=psMeshReqData->u8McstTransmission;
0221     sDataReqRsp.sDataFrameData.u8BcstTransmission=psMeshReqData->u8BcstTransmission;
0222     sDataReqRsp.sDataFrameData.u8ReliableBcst=psMeshReqData->u8ReliableBcst;
0223
0231     //Send the data frame
0232     vNetApiDataRequest(&sDataReqRsp,&sDataSyncCfm);
0233
0234     //Process confirm
0235     psMeshSyncCfm->uParam.sMeshCfmData.u8MhsduHandle=psMeshReqData->u8MhsduHandle;
0236     psMeshSyncCfm->uParam.sMeshCfmData.eStatus = sDataSyncCfm.u8Status;
0237
0238 }
0239
0240
0241 void  vGetOneHopNeighbor(uint8 *pu8NumberOfOps,uint8 *pu8NeighborNumber)
0242 {
0243     uint8 j=0;
```

```
0244
0249     while (j!=MAX_NEIGHBORS)
0250     {
0251         if (*pu8NumberOfOps!=1)
0252         {
0253             //neighbors connected to nb
0254             for (j=0;j<MAX_NEIGHBORS;j++)
0255             {
0256                 if
((sMeshInfo.psNeighborList[j].u8NumberOfOps<*pu8NumberOfOps)&&(sMeshData.p2bConnectivityMatrix[*pu8NeighborNu
mber+1][j+1]))
0257                 {
0258                     *pu8NeighborNumber=j;
0263                     *pu8NumberOfOps=sMeshInfo.psNeighborList[j].u8NumberOfOps;
0264                     break;
0265                 }
0266
0267             }
0268         }
0269         else
0270             break;
0271     }
0272 }
```

XV. MeshServices.h

```
0001 #ifndef MESH_SERVICES
0002 #define MESH_SERVICES
0003
0004
0005 //Mesh data queue
0006 void InitializeMhsduHandleQueue(void);
0007 int16 i16InsertMhsduHandle(uint8 u8MhsduHandle,uint8 u8Type);
0008 bool bValidateMhsduHandle(uint8 u8MhsduHandle);
0009 void vEraseMhsduHandle(uint8 u8MhsduHandle);
0010 inline void vHdlMeshReqData(NET_MeshReqData_s *psMeshReqData, NET_MeshSyncCfm_s *psMeshSyncCfm);
0011 void  vGetOneHopNeighbor(uint8 *u8NumberOfOps,uint8 *u8NeighborNumber);
0012 void vTranslateAddress(uint16 u16Short, MAC_ExtAddr_s* psExt);
0013 #endif //MESH_SERVICES
```

XVI. mhme.c

```
0001 #include <AppApi.h>
0002 #include <mac_pib.h>
0003 //#include <AppHardwareApi.h>
0004 #include <string.h>
0005
0006 #include "Mhme.h"
0007 #include "MhmeServices.h"
0008 #include "MhmeControl.h"
0009 #include "CommandFrames.h"
0010 #include "config.h"
0011
0012 #include "VirtualTimer.h"
0013
0014 #ifdef DEBUG
0015 #include "debugger.h"
0016 #endif
0017
0018
0019 volatile MAC_MlmeBuffer_s psMlmeBuffers[N_MLME_BUFFERS] __attribute__ ((aligned (4)));
0020
```

```
0021 NET_MeshInfo_s sMeshInfo;
0022 NET_MeshInfo_s sMeshInfoCopy;
0023 NET_MeshData_s sMeshData;
0024
0025 MAC_MlmeReqRsp_s sMlmeReqRsp;
0026 MAC_MlmeSyncCfm_s sMlmeSyncCfm;
0027
0028 volatile NET_MhmeMhsduHandleQueue_s sMhmeMhsduHandleQueue;
0029
0030 //change to bit field
0031 volatile bool bJoinNetworkStatus;
0032
0033 PUBLIC void vNetApiMhmeRequest(NET_MhmeReqRsp_s *psMhmeReqRsp, NET_MhmeSyncCfm_s *psMhmeSyncCfm)
0034 {
0035
0036     //Register this function on the debugger
0037     #ifdef DEBUG
0038     u8StackPushIdentifier("MHME",strlen("MHME"),TRUE);
0039     u8StackPushIdentifier("vNetApiMhmeRequest",strlen("vNetApiMhmeRequest"),FALSE);
0040     #endif
0041
0042     switch(psMhmeReqRsp->u8Type)
0043     {
0044         case NET_MHME_REQ_DISCOVER:
0045         {
0046             #ifdef DEBUG
0047             vStackPrintf(__FILE__,__LINE__,"Received NET_MHME_REQ_DISCOVER.");
0048             #endif
0049
0050             if(u8MhmeFlag!=MHME_LEAVING)
0051                 vHdlMhmeReqDiscover(&psMhmeReqRsp->uParam.sReqDiscover, &psMhmeSyncCfm-
>uParam.sCfmDiscover);
0052             else
0053                 psMhmeSyncCfm->uParam.sCfmLeave.u8Status=CFM_STATUS_INVALID_REQUEST;
0054         }break;
0055
0056         case NET_MHME_REQ_START_NETWORK:
0057         {
0058             #ifdef DEBUG
0059             vStackPrintf(__FILE__,__LINE__,"Received NET_MHME_REQ_START_NETWORK.");
0060             #endif
0061
0062             if((!bJoinNetworkStatus)&&(u8MhmeFlag!=MHME_LEAVING))
0063                 vHdlMhmeReqStartNetwork(&psMhmeReqRsp->uParam.sReqStartNetwork, &psMhmeSyncCfm-
>uParam.sCfmStartNetwork);
0064             else
0065                 psMhmeSyncCfm->uParam.sCfmLeave.u8Status=CFM_STATUS_INVALID_REQUEST;
0066         }break;
0067
0068         case NET_MHME_REQ_START_DEVICE:
0069         {
0070             #ifdef DEBUG
0071             vStackPrintf(__FILE__,__LINE__,"Received NET_MHME_REQ_START_DEVICE.");
0072             #endif
0073
0074             if((bJoinNetworkStatus)&&(u8MhmeFlag!=MHME_LEAVING))
0075                 vHdlMhmeReqStartDevice(&psMhmeReqRsp->uParam.sReqStartDevice, &psMhmeSyncCfm-
>uParam.sCfmStartDevice);
0076             else
0077                 psMhmeSyncCfm->uParam.sCfmLeave.u8Status=CFM_STATUS_INVALID_REQUEST;
0078         }break;
0079
0080         case NET_MHME_REQ_JOIN:
0081         {
0082             #ifdef DEBUG
```

```
0083            vStackPrintf(__FILE__,__LINE__,"Received NET_MHME_REQ_JOIN.");
0084            #endif
0085
0086            if(u8MhmeFlag!=MHME_LEAVING)
0087                vHdlMhmeReqJoin(&psMhmeReqRsp->uParam.sReqJoin,&psMhmeSyncCfm->uParam.sCfmJoin);
0088            else
0089                psMhmeSyncCfm->uParam.sCfmLeave.u8Status=CFM_STATUS_INVALID_REQUEST;
0090        }break;
0091
0092        case NET_MHME_REQ_LEAVE:
0093        {
0094            #ifdef DEBUG
0095            vStackPrintf(__FILE__,__LINE__,"Received NET_MHME_REQ_LEAVE.");
0096            #endif
0097
0098            if(((bJoinNetworkStatus)||(sMeshData.bIsCoordinator))&&(u8MhmeFlag!=MHME_LEAVING))
0099                vHdlMhmeReqLeave(&psMhmeReqRsp->uParam.sReqLeave,&psMhmeSyncCfm->uParam.sCfmLeave);
0100            else
0101                psMhmeSyncCfm->uParam.sCfmLeave.u8Status=CFM_STATUS_INVALID_REQUEST;
0102        }break;
0103
0104        case NET_MHME_REQ_RESET:
0105        {
0106            #ifdef DEBUG
0107            vStackPrintf(__FILE__,__LINE__,"Received NET_MHME_REQ_RESET.");
0108            #endif
0109
0110            if(u8MhmeFlag!=MHME_LEAVING)
0111                vHdlMhmeReqReset(&psMhmeSyncCfm->uParam.sCfmReset);
0112            else
0113                psMhmeSyncCfm->uParam.sCfmLeave.u8Status=CFM_STATUS_INVALID_REQUEST;
0114        }break;
0115
0116        case NET_MHME_REQ_GET:
0117        {
0118            #ifdef DEBUG
0119            vStackPrintf(__FILE__,__LINE__,"Received NET_MHME_REQ_GET.");
0120            #endif
0121
0122            vHdlMhmeReqGet(&psMhmeReqRsp->uParam.sReqGet,&psMhmeSyncCfm->uParam.sCfmGet);
0123        }break;
0124
0125        case NET_MHME_REQ_SET:
0126        {
0127            #ifdef DEBUG
0128            vStackPrintf(__FILE__,__LINE__,"Received NET_MHME_REQ_SET.");
0129            #endif
0130
0131            vHdlMhmeReqSet(&psMhmeReqRsp->uParam.sReqSet,&psMhmeSyncCfm->uParam.sCfmSet);
0132        }break;
0133        default:
0134            #ifdef DEBUG
0135            vStackPrintf(__FILE__,__LINE__,"Unrecognized request\\response.");
0136            #endif
0137            break;
0138    }
0139
0140    //Unregister this function off the debugger
0141    #ifdef DEBUG
0142    vStackPopIdentifier(); //For the vNetApiMhmeRequest
0143    vStackPopIdentifier(); //For the MHME
0144    #endif
0145 }
0146
0147
```

```
0148
0149
0150
0151 uint16 u16MhmeGetMessage(NET_MhmeDcfmInd_s* psMhmeDcfmInd)
0152 {
0153     uint16 u16ReturnValue=0;
0154
0155     //Register this function on the debugger
0156     #ifdef DEBUG
0157     u8StackPushIdentifier("MHME",strlen("MHME"),TRUE);
0158     u8StackPushIdentifier("u16MhmeGetMessage",strlen("u16MhmeGetMessage"),FALSE);
0159     #endif
0160
0161     //Test if there is any message on the queue
0162     if(sMhmeQueue.u8MsgInQueue>0)
0163     {
0164         //Retreive the existing message from the queue and into the application buffer
0165         sMhmeQueue.u8MsgInQueue--;
0166
0167         //Note: preprocessing may be needed before passing it to the application
0168         u16ReturnValue=u16MhmeTranslateMessage( psMhmeDcfmInd,\
0169
&sMhmeQueue.asMhmeEventQueue[(sMhmeQueue.u8IndexOfFirstNonProcessedMsg++)%MAX_MSG_IN_QUEUE]);
0170     }
0171
0172     //Deregister this function off the debugger
0173     #ifdef DEBUG
0174     vStackPopIdentifier(); //For the u16MhmeGetMessage
0175     vStackPopIdentifier(); //For the MHME
0176     #endif
0177
0178     //The return value is either 1 (if there is a message stored in the buffer) or 0 (if it is not)
0179     return u16ReturnValue;
0180 }
0181
0182
0183
0184
0185
0186
0187 uint16 u16MhmeTranslateMessage(NET_MhmeDcfmInd_s* psMhmeDcfmInd,NET_MhmeEvent_s* psMhmeEvent)
0188 {
0189     uint16 u16ReturnValue=0;
0190
0191     //Register this function on the debugger
0192     #ifdef DEBUG
0193     u8StackPushIdentifier("u16MhmeTranslateMessage",strlen("u16MhmeTranslateMessage"),FALSE);
0194     #endif
0195
0196     //The preprocessing that needs to be done before the message be passed to the application
0197     switch(psMhmeEvent->u8Type)
0198     {
0199         case NET_MHME_COMMD_EVENT:
0200         {
0201             #ifdef DEBUG
0202             vStackPrintf(__FILE__,__LINE__,"Received NET_MHME_COMMD_EVENT.");
0203             #endif
0204             u16ReturnValue=u16MhmeTranslateCommMessage(psMhmeDcfmInd,&psMhmeEvent->uParam.sMeshDcfmInd);
0205         }break;
0206
0207         case NET_MHME_MLME_EVENT:
0208         {
0209             #ifdef DEBUG
0210             vStackPrintf(__FILE__,__LINE__,"Received NET_MHME_MLME_EVENT.");
0211             #endif
```

```
0212            u16ReturnValue=u16MhmeTranslateMlmeMessage(psMhmeDcfmInd,&psMhmeEvent->uParam.sMlmeDcfmInd);
0213        }break;
0214
0215        case NET_MHME_TIMER_EVENT:
0216        {
0217            #ifdef DEBUG
0218            vStackPrintf(__FILE__,__LINE__,"Received NET_MHME_TIMER_EVENT.");
0219            #endif
0220            u16ReturnValue=u16MhmeTranslateTmerMessage(psMhmeDcfmInd,&psMhmeEvent->uParam.sMhmeTmerInd);
0221        }break;
0222
0223        default:
0224        {
0225            #ifdef DEBUG
0226            vStackPrintf(__FILE__,__LINE__,"Unhandle message type.");
0227            #endif
0228            u16ReturnValue=0;
0229        }
0230    }
0231
0232    //Deregister this function off the debugger
0233    #ifdef DEBUG
0234    vStackPopIdentifier();
0235    #endif
0236
0237    //The return value is either 1 (if the message is ready to be returned to the application) or 0 (if
it is not)
0238    return u16ReturnValue;
0239 }
0240
0241
0242
0243
0244
0245 inline uint16 u16MhmeTranslateCommMessage(NET_MhmeDcfmInd_s* psMhmeDcfmInd,NET_MeshDcfmInd_s*
psMeshDcfmInd)
0246 {
0247    uint16 u16ReturnValue=0;
0248
0249    //Register this function on the debugger
0250    #ifdef DEBUG
0251    u8StackPushIdentifier("u16MhmeTranslateCommMessage",strlen("u16MhmeTranslateCommMessage"),FALSE);
0252    #endif
0253
0254    #ifdef DEBUG
0255    vStackPrintf(__FILE__,__LINE__,"Processing a command event.");
0256    #endif
0257
0258
0259    switch(psMeshDcfmInd->u8Type)
0260    {
0261        case NET_MESH_DCFM_DATA:
0262        {
0263            #ifdef DEBUG
0264            vStackPrintf(__FILE__,__LINE__,"Received NET_MESH_DCFM_DATA.");
0265            #endif
0266            u16ReturnValue=u16HdlMhmeCfmData(psMhmeDcfmInd,psMeshDcfmInd);
0267        }break;
0268
0269        case NET_MESH_IND_DATA:
0270        {
0271            #ifdef DEBUG
0272            vStackPrintf(__FILE__,__LINE__,"Received NET_MESH_IND_DATA.");
0273            #endif
0274            u16ReturnValue=u16HdlMhmeIndData(psMhmeDcfmInd,psMeshDcfmInd);
```

```
0275         }break;
0276
0277         default:
0278         {
0279             #ifdef DEBUG
0280             vStackPrintf(__FILE__,__LINE__,"Unhandle command event.");
0281             #endif
0282         }
0283     }
0284
0285     //Deregister this function off the debugger
0286     #ifdef DEBUG
0287     vStackPopIdentifier();
0288     #endif
0289
0290     //The return value is either 1 (if the message is ready to be returned to the application) or 0 (if
it is not)
0291     return u16ReturnValue;
0292 }
0293
0294
0295
0296
0297
0298 inline uint16 u16MhmeTranslateMlmeMessage(NET_MhmeDcfmInd_s* psMhmeDcfmInd,MAC_MlmeDcfmInd_s*
psMlmeDcfmInd)
0299 {
0300     uint16 u16ReturnValue=0;
0301
0302     //Register this function on the debugger
0303     #ifdef DEBUG
0304     u8StackPushIdentifier("u16MhmeTranslateMlmeMessage",strlen("u16MhmeTranslateMlmeMessage"),FALSE);
0305     #endif
0306
0307     switch(psMlmeDcfmInd->u8Type)
0308     {
0309         case MAC_MLME_DCFM_SCAN:
0310         {
0311             #ifdef DEBUG
0312             vStackPrintf(__FILE__,__LINE__,"Received MAC_MLME_DCFM_SCAN.");
0313             #endif
0314             if(u8MhmeFlag==MHME_DISCOVERING)
0315                 u16ReturnValue=u16HdlMhmeCfmDiscover(psMhmeDcfmInd,psMlmeDcfmInd);
0316         } break;
0317
0318         case MAC_MLME_DCFM_ASSOCIATE:
0319         {
0320             #ifdef DEBUG
0321             vStackPrintf(__FILE__,__LINE__,"Received MAC_MLME_DCFM_ASSOCIATE.");
0322             #endif
0323             u16ReturnValue=u16HdlMhmeCfmJoin(psMhmeDcfmInd,psMlmeDcfmInd);
0324         } break;
0325
0326         case MAC_MLME_IND_ASSOCIATE:
0327         {
0328             #ifdef DEBUG
0329             vStackPrintf(__FILE__,__LINE__,"Received MAC_MLME_IND_ASSOCIATE.");
0330             #endif
0331             u16ReturnValue=u16HdlMhmeIndJoin(psMhmeDcfmInd,psMlmeDcfmInd);
0332         } break;
0333
0334         case MAC_MLME_IND_COMM_STATUS:
0335         {
0336             #ifdef DEBUG
0337             vStackPrintf(__FILE__,__LINE__,"Received MAC_MLME_IND_COMM_STATUS.");
```

```
0338             #endif
0339             u16ReturnValue=u16HdlMhmeIndCommStatus(psMhmeDcfmInd,psMlmeDcfmInd);
0340             #ifdef DEBUG
0341             vStackPrintf(__FILE__,__LINE__,"MAC_MLME_IND_COMM_STATUS Dispatched");
0342             #endif
0343         } break;
0344
0345         /** . . . **/
0346
0347         default:
0348         {
0349             #ifdef DEBUG
0350             vStackPrintf(__FILE__,__LINE__,"Unhandle confirm/indication.");
0351             #endif
0352             u16ReturnValue=0;
0353             break;
0354         }
0355     }
0356
0357     //Deregister this function off the debugger
0358     #ifdef DEBUG
0359     vStackPopIdentifier();
0360     #endif
0361
0362     //The return value is either 1 (if the message is ready to be returned to the application) or 0 (if
it is not)
0363     return u16ReturnValue;
0364 }
0365
0366
0367
0368
0369
0370 inline uint16 u16MhmeTranslateTmerMessage(NET_MhmeDcfmInd_s* psMhmeDcfmInd,NET_MhmeTmerInd_s*
psMhmeTmerInd)
0371 {
0372     uint16 u16ReturnValue=0;
0373
0374     //Register this function on the debugger
0375     #ifdef DEBUG
0376     u8StackPushIdentifier("u16MhmeTranslateTmerMessage",strlen("u16MhmeTranslateTmerMessage"),FALSE);
0377     #endif
0378
0379     #ifdef DEBUG
0380     vStackPrintf(__FILE__,__LINE__,"Processing timer event.");
0381     vPrintf("Timer Expired=%d\n",psMhmeTmerInd->u8TriggeredTimer);
0382     #endif
0383
0384     if(psMhmeTmerInd->u8TriggeredTimer==sMeshData.i8ChilderReportTimer)
0385     {
0386             #ifdef DEBUG
0387             vStackPrintf(__FILE__,__LINE__,"Creating child report. . .");
0388             #endif
0389             u16ReturnValue=u16HdlMhmeChildrenNumberReport(psMhmeDcfmInd,psMhmeTmerInd);
0390     }
0391     else if(psMhmeTmerInd->u8TriggeredTimer==sMeshData.i8AddressAssignmentTimer)
0392     {
0393             #ifdef DEBUG
0394             vStackPrintf(__FILE__,__LINE__,"Sending address assignment. . .");
0395             #endif
0396             u16ReturnValue=u16HdlMhmeAddressAssignment(psMhmeDcfmInd,psMhmeTmerInd);
0397     }
0398     else if(psMhmeTmerInd->u8TriggeredTimer==sMeshData.i8HelloTimer)
0399     {
0400         if(u8MhmeFlag==MHME_TOPOLOGY_DISCOVERY)
```

```
0401        {
0402            #ifdef DEBUG
0403            vStackPrintf(__FILE__,__LINE__,"Creating hello command frame. . .");
0404            #endif
0405            u16ReturnValue=u16HdlMhmeHello(psMhmeDcfmInd,psMhmeTmerInd);
0406        }
0407        else
0408        {
0409            #ifdef DEBUG
0410            vStackPrintf(__FILE__,__LINE__,"Creating hello command frame in the wrong stage. . .");
0411            #endif
0412            u16ReturnValue=u16HdlMhmeHello(psMhmeDcfmInd,psMhmeTmerInd);
0413        }
0414    }
0415    else if(psMhmeTmerInd->u8TriggeredTimer==sMeshData.i8LeaveTimer)
0416    {
0417        NET_CommSyncCfm_s sCommSyncCfm;
0418
0419        #ifdef DEBUG
0420        vStackPrintf(__FILE__,__LINE__,"Resending the leave command frames. . .");
0421        #endif
0422
0423        //Send again the remaining leave commands
0424        vNetApiCommRequest(FRAME_LEAVE,&sCommSyncCfm);
0425    }
0426    else
0427    {
0428        bool bIsMhmeTimer=FALSE;
0429        uint8 i;
0430
0431        for(i=0;i<MAX_NEIGHBORS;i++)
0432        {
0433            if(psMhmeTmerInd->u8TriggeredTimer==sMeshInfo.psNeighborList[i].i8RejoinTimer)
0434            {
0435                uint8 j=0;
0436
0437                #ifdef DEBUG
0438                vStackPrintf(__FILE__,__LINE__,"A Rejoin timer has expired.");
0439                #endif
0440
0441
0442
if(sMeshInfo.psNeighborList[i].u16EndingAddress<sMeshInfo.u16NetworkAddress+sMeshData.sFreeBlock[0].u16Beginn
ingBlock*sMeshData.u16BlockSize+sMeshData.sFreeBlock[j].u16BeginningBlock)
0443                {
0444                    sMeshData.u16FreeBlocks+=(sMeshInfo.psNeighborList[i].u16EndingAddress-
sMeshInfo.psNeighborList[i].u16BeginningAddress)/sMeshData.u16BlockSize;
0445                    sMeshData.u16AllocatedAddresses-=sMeshInfo.psNeighborList[i].u16EndingAddress-
sMeshInfo.psNeighborList[i].u16BeginningAddress;
0446                    sMeshData.u16NumberOfFreeBlocksInList++;
0447
memcpy(&sMeshData.sFreeBlock[1],&sMeshData.sFreeBlock[0],sizeof(NET_FreeBlock_s)*MAX_NEIGHBORS);
0448
sMeshData.sFreeBlock[0].u16BeginningBlock=sMeshInfo.psNeighborList[i].u16BeginningAddress/sMeshData.u16BlockS
ize;
0449
sMeshData.sFreeBlock[0].u16NumberOfBlocks=(sMeshInfo.psNeighborList[i].u16EndingAddress-
sMeshInfo.psNeighborList[i].u16BeginningAddress)/sMeshData.u16BlockSize;
0450                }
0451                else
0452                {
0453                    //Free of the addresses that the device had
0454                    for(j=0;j<sMeshData.u16NumberOfFreeBlocksInList;j++)
0455                    {
0456                        #ifdef DEBUG
```

```
0457                        vPrintf("Executing procedure to free child blocks\n");
0458                        vPrintf("Free block list information\n");
0459                        vPrintf("Block Size: %d\n",sMeshData.u16BlockSize);
0460                        vPrintf("First Free Block: %d\n",sMeshData.sFreeBlock[j].u16BeginningBlock);
0461                        vPrintf("Number of blocks: %d\n",sMeshData.sFreeBlock[j].u16NumberOfBlocks);
0462                        vPrintf("Lenght of Free Block:
%d\n",(sMeshData.sFreeBlock[j].u16BeginningBlock+sMeshData.sFreeBlock[j].u16NumberOfBlocks)*sMeshData.u16Bloc
kSize);
0463                        vPrintf("Number of free blocks in List:
%d\n",sMeshData.u16NumberOfFreeBlocksInList);
0464                        vPrintf("Neighbor information\n");
0465                        vPrintf("Neighbor Beginning Addr:
%d\n",sMeshInfo.psNeighborList[i].u16BeginningAddress);
0466                        vPrintf("Neighbor Ending Addr:
%d\n",sMeshInfo.psNeighborList[i].u16EndingAddress);
0467                        #endif
0468
0469
if((sMeshData.sFreeBlock[j].u16BeginningBlock+sMeshData.sFreeBlock[j].u16NumberOfBlocks)*sMeshData.u16BlockSi
ze\
0470
+sMeshData.sFreeBlock[j+1].u16BeginningBlock+sMeshInfo.u16NetworkAddress==(sMeshInfo.psNeighborList[i].u16Beg
inningAddress))
0471                        {
0472                            #ifdef DEBUG
0473                            vPrintf("CASE I\n");
0474                            #endif
0475
sMeshData.sFreeBlock[j].u16NumberOfBlocks+=(sMeshInfo.psNeighborList[i].u16EndingAddress-
sMeshInfo.psNeighborList[i].u16BeginningAddress)/sMeshData.u16BlockSize;
0476                            sMeshData.u16AllocatedAddresses-
=sMeshInfo.psNeighborList[i].u16EndingAddress-sMeshInfo.psNeighborList[i].u16BeginningAddress;
0477                            sMeshData.u16FreeBlocks+=(sMeshInfo.psNeighborList[i].u16EndingAddress-
sMeshInfo.psNeighborList[i].u16BeginningAddress)/sMeshData.u16BlockSize;
0478                            if(j==MAX_NEIGHBORS) break;
0479
if(sMeshData.sFreeBlock[j+1].u16BeginningBlock==sMeshData.sFreeBlock[j].u16BeginningBlock+sMeshData.sFreeBloc
k[j].u16NumberOfBlocks+1)
0480                            {
0481
sMeshData.sFreeBlock[j].u16NumberOfBlocks+=sMeshData.sFreeBlock[j+1].u16NumberOfBlocks;
0482                                if(j<MAX_NEIGHBORS-1)
0483
memcpy(&sMeshData.sFreeBlock[j+1],&sMeshData.sFreeBlock[j+2],sizeof(NET_FreeBlock_s)*(MAX_NEIGHBORS-j-1));
0484                                sMeshData.u16NumberOfFreeBlocksInList--;
0485                                break;
0486                            }
0487                        }
0488
if((sMeshInfo.u16NetworkAddress+sMeshData.sFreeBlock[j].u16BeginningBlock)*sMeshData.u16BlockSize\
0489
+sMeshData.sFreeBlock[j].u16BeginningBlock==sMeshInfo.psNeighborList[i].u16EndingAddress)
0490                        {
0491                            #ifdef DEBUG
0492                            vPrintf("\nCASE II\n");
0493                            #endif
0494                            sMeshData.sFreeBlock[j].u16BeginningBlock-
=(sMeshInfo.psNeighborList[i].u16EndingAddress-
sMeshInfo.psNeighborList[i].u16BeginningAddress)/sMeshData.u16BlockSize;
0495
sMeshData.sFreeBlock[j].u16NumberOfBlocks+=(sMeshInfo.psNeighborList[i].u16EndingAddress-
sMeshInfo.psNeighborList[i].u16BeginningAddress)/sMeshData.u16BlockSize;
0496                            sMeshData.u16AllocatedAddresses-
=sMeshInfo.psNeighborList[i].u16EndingAddress-sMeshInfo.psNeighborList[i].u16BeginningAddress;
```

```
0497                            sMeshData.u16FreeBlocks+=(sMeshInfo.psNeighborList[i].u16EndingAddress-
sMeshInfo.psNeighborList[i].u16BeginningAddress)/sMeshData.u16BlockSize;
0498                            break;
0499                        }
0500
0501                        if(j+1==sMeshData.u16NumberOfFreeBlocksInList) break;
0502
0503                        if(
((sMeshData.sFreeBlock[j].u16BeginningBlock+sMeshData.sFreeBlock[j].u16NumberOfBlocks)*sMeshData.u16BlockSize
\
0504
+sMeshData.sFreeBlock[j+1].u16BeginningBlock+sMeshInfo.u16NetworkAddress<(sMeshInfo.psNeighborList[i].u16Begi
nningAddress))&&
0505                            (sMeshData.sFreeBlock[j+1].u16BeginningBlock*sMeshData.u16BlockSize\
0506
+sMeshData.sFreeBlock[j+1].u16BeginningBlock+sMeshInfo.u16NetworkAddress>(sMeshInfo.psNeighborList[i].u16Endi
ngAddress)))
0507                        {
0508                            #ifdef DEBUG
0509                            vPrintf("CASE III\n");
0510                            #endif
0511                            sMeshData.u16NumberOfFreeBlocksInList++;
0512
memcpy(&sMeshData.sFreeBlock[j+2],&sMeshData.sFreeBlock[j+1],sizeof(NET_FreeBlock_s)*(MAX_NEIGHBORS-j-1));
0513
sMeshData.sFreeBlock[j+1].u16BeginningBlock=sMeshInfo.psNeighborList[i].u16BeginningAddress/sMeshData.u16Bloc
kSize;
0514
sMeshData.sFreeBlock[j+1].u16NumberOfBlocks=(sMeshInfo.psNeighborList[i].u16EndingAddress-
sMeshInfo.psNeighborList[i].u16BeginningAddress)/sMeshData.u16BlockSize;
0515                            sMeshData.u16AllocatedAddresses-
=sMeshInfo.psNeighborList[i].u16EndingAddress-sMeshInfo.psNeighborList[i].u16BeginningAddress;
0516                            sMeshData.u16FreeBlocks+=(sMeshInfo.psNeighborList[i].u16EndingAddress-
sMeshInfo.psNeighborList[i].u16BeginningAddress)/sMeshData.u16BlockSize;
0517                            break;
0518                        }
0519                    }
0520
0521                    if(j+1==sMeshData.u16NumberOfFreeBlocksInList)
0522                    {
0523                        #ifdef DEBUG
0524                        vPrintf("CASE IV\n");
0525                        #endif
0526
if((sMeshData.sFreeBlock[j].u16BeginningBlock+sMeshData.sFreeBlock[j].u16NumberOfBlocks)*sMeshData.u16BlockSi
ze+\
0527
sMeshInfo.u16NetworkAddress+sMeshData.sFreeBlock[j].u16BeginningBlock<(sMeshInfo.psNeighborList[i].u16Beginni
ngAddress))
0528                        {
0529                            sMeshData.u16NumberOfFreeBlocksInList++;
0530
sMeshData.sFreeBlock[j+1].u16BeginningBlock=sMeshInfo.psNeighborList[i].u16BeginningAddress/sMeshData.u16Bloc
kSize;
0531
sMeshData.sFreeBlock[j+1].u16NumberOfBlocks=(sMeshInfo.psNeighborList[i].u16EndingAddress-
sMeshInfo.psNeighborList[i].u16BeginningAddress)/sMeshData.u16BlockSize;
0532                            sMeshData.u16FreeBlocks+=(sMeshInfo.psNeighborList[i].u16EndingAddress-
sMeshInfo.psNeighborList[i].u16BeginningAddress)/sMeshData.u16BlockSize;
0533                            sMeshData.u16AllocatedAddresses-
=sMeshInfo.psNeighborList[i].u16EndingAddress-sMeshInfo.psNeighborList[i].u16BeginningAddress;
0534                        }
0535                    }
0536
0537                }
```

```
0538
0539                    bIsMhmeTimer=TRUE;
0540                    sMeshInfo.u8NbOfChildren--;
0541                    sMeshInfo.psNeighborList[i].u16BeginningAddress=0xfffe;
0542                    sMeshInfo.psNeighborList[i].u16EndingAddress=0x0;
0543                    sMeshInfo.psNeighborList[i].u8TreeLevel=0xff;
0544                    sMeshInfo.psNeighborList[i].u8Relationship=NO_RELATIONSHIP;
0545                    sMeshInfo.psNeighborList[i].u8NumberOfOps=0xff;
0546                    sMeshInfo.psNeighborList[i].u8NbOfHello=0;
0547                    for(j=0;j<MAX_NEIGHBORS+1;j++)
0548                        sMeshData.p2bConnectivityMatrix[j][i+1]=\
0549                        sMeshData.p2bConnectivityMatrix[i+1][j]=FALSE;
0550                    if(sMeshInfo.psNeighborList[i].i8RejoinTimer!=-1)
0551                    {
0552                        VirtualTimer_bStop(sMeshInfo.psNeighborList[i].i8RejoinTimer);
0553                        VirtualTimer_bDelete(sMeshInfo.psNeighborList[i].i8RejoinTimer);
0554                        sMeshInfo.psNeighborList[i].i8RejoinTimer=-1;
0555                    }
0556                    UNPLACEBITMAP(sMeshData.sBitMapRejoin,i);
0557
0558                    #ifdef DEBUG
0559                    vPrintf("Procedure done\n");
0560                    vPrintf("Free block list information\n");
0561                    vPrintf("Block Size: %d\n",sMeshData.u16BlockSize);
0562                    vPrintf("First Free Block: %d\n",sMeshData.sFreeBlock[j].u16BeginningBlock);
0563                    vPrintf("Number of blocks: %d\n",sMeshData.sFreeBlock[j].u16NumberOfBlocks);
0564                    vPrintf("Lenght of Free Block:
%d\n",(sMeshData.sFreeBlock[j].u16BeginningBlock+sMeshData.sFreeBlock[j].u16NumberOfBlocks)*sMeshData.u16Bloc
kSize);
0565                    vPrintf("Number of free blocks in List: %d\n",sMeshData.u16NumberOfFreeBlocksInList);
0566                    vPrintf("Neighbor information\n");
0567                    vPrintf("Neighbor Beginning Addr:
%d\n",sMeshInfo.psNeighborList[i].u16BeginningAddress);
0568                    vPrintf("Neighbor Ending Addr: %d\n",sMeshInfo.psNeighborList[i].u16EndingAddress);
0569                    #endif
0570
0571                    break;
0572                }
0573            }
0574
0595            if(!bIsMhmeTimer)
0596            {
0597                #ifdef DEBUG
0598                vStackPrintf(__FILE__,__LINE__,"Another timer has expired, sending it to application.");
0599                #endif
0600                u16ReturnValue=u16HdlMhmeAppTimerExpired(psMhmeDcfmInd,psMhmeTmerInd);
0601            }
0602        }
0603
0604        //Deregister this function off the debugger
0605        #ifdef DEBUG
0606        vStackPopIdentifier();
0607        #endif
0608
0609        //The return value is either 1 (if the message is ready to be returned to the application) or 0 (if
it is not)
0610        return u16ReturnValue;
0611    }
0612
0613
0614
0615
0616    PUBLIC void vMlmeDcfmIndPost(void *pvParam, MAC_DcfmIndHdr_s *psDcfmIndHdr)
0617    {
0618        int i;
```

```
0619     MAC_MlmeDcfmInd_s* psMlmeInd=(MAC_MlmeDcfmInd_s*)psDcfmIndHdr;
0620
0621     //Put the message in the message queue
0622     if(sMhmeQueue.u8MsgInQueue<MAX_MSG_IN_QUEUE)
0623     {
0624         sMhmeQueue.u8MsgInQueue++;
0625
sMhmeQueue.asMhmeEventQueue[(sMhmeQueue.u8IndexOfFirstFreeSpot)%MAX_MSG_IN_QUEUE].u8Type=NET_MHME_MLME_EVENT;
0626
memcpy(&sMhmeQueue.asMhmeEventQueue[(sMhmeQueue.u8IndexOfFirstFreeSpot++)%MAX_MSG_IN_QUEUE].uParam.sMlmeDcfmI
nd,psMlmeInd,sizeof(MAC_MlmeDcfmInd_s));
0627         #ifdef DEBUG
0628         vPrintf("MHME.vMlmeDcfmIndPost: Message received. (%s: %d)\n",__FILE__,__LINE__);
0629         #endif
0630     }
0631     else
0632     {
0633         #ifdef DEBUG
0634         vPrintf("MHME.vMlmeDcfmIndPost: To many messages in the message queue, so current message will
be ignored. (%s: %d)\n",__FILE__,__LINE__);
0635         #endif
0636     }
0637     #ifdef DEBUG
0638     WRITEBITMAP(sMeshData.sBitMapLeave);
0639     #endif
0640     for(i=0;i<N_MLME_BUFFERS;i++)
0641     {
0642         if(psMlmeInd==&psMlmeBuffers[i].sMlmeDcfmInd)
0643         {
0644             psMlmeBuffers[i].u8Used=FALSE;
0648             break;
0649         }
0650     }
0651     #ifdef DEBUG
0652     WRITEBITMAP(sMeshData.sBitMapLeave);
0653     #endif
0654 }
0655
0656 PUBLIC MAC_DcfmIndHdr_s *psMlmeDcfmIndGetBuf(void *pvParam)
0657 {
0658     uint8 i;
0659
0660     for(i=0; i<N_MLME_BUFFERS; i++)
0661     {
0662         if(psMlmeBuffers[i].u8Used == FALSE)
0663         {
0664             psMlmeBuffers[i].u8Used = TRUE;
0665             return (MAC_DcfmIndHdr_s*) &psMlmeBuffers[i].sMlmeDcfmInd;
0666         }
0667     }
0668     return NULL;
0669 }
0670
0671 void vMhmeCommandPost(NET_MeshDcfmInd_s* psMeshDcfmInd)
0672 {
0673     //Put the message in the message queue
0674     if(sMhmeQueue.u8MsgInQueue<MAX_MSG_IN_QUEUE)
0675     {
0676         sMhmeQueue.u8MsgInQueue++;
0677
sMhmeQueue.asMhmeEventQueue[(sMhmeQueue.u8IndexOfFirstFreeSpot)%MAX_MSG_IN_QUEUE].u8Type=NET_MHME_COMMD_EVENT
;
0678
memcpy(&sMhmeQueue.asMhmeEventQueue[(sMhmeQueue.u8IndexOfFirstFreeSpot++)%MAX_MSG_IN_QUEUE].uParam.sMeshDcfmI
nd,psMeshDcfmInd,sizeof(NET_MeshDcfmInd_s));
```

```
0679         #ifdef DEBUG
0680         vPrintf("MHME.vMhmeCommandPost: Message received. (%s: %d)\n",__FILE__,__LINE__);
0681         #endif
0682     }
0683     else
0684     {
0685         #ifdef DEBUG
0686         vPrintf("MHME.vMhmeCommandPost: To many messages in the message queue, current message will be
ignored. (%s: %d)\n",__FILE__,__LINE__);
0687         #endif
0688     }
0689 }
0690
0691 void vTickTimerISR(uint32 u32Device, uint32 u32ItemBitmap)
0692 {
0693     NET_MhmeTmerInd_s sMhmeTmerInd;
0694     uint32              u32Filter=1;
0695     uint8              i;
0696
0697     for(i=0;i<32;i++)
0698     {
0699         if((u32ItemBitmap&u32Filter)!=0L)
0700         {
0701             //Stop the timer
0702             VirtualTimer_bStop(i);
0703
0704             if(sMhmeQueue.u8MsgInQueue<MAX_MSG_IN_QUEUE)
0705             {
0706                 sMhmeTmerInd.u8TriggeredTimer=i;
0707                 sMhmeQueue.u8MsgInQueue++;
0708
sMhmeQueue.asMhmeEventQueue[(sMhmeQueue.u8IndexOfFirstFreeSpot)%MAX_MSG_IN_QUEUE].u8Type=NET_MHME_TIMER_EVENT
;
0709
memcpy(&sMhmeQueue.asMhmeEventQueue[(sMhmeQueue.u8IndexOfFirstFreeSpot++)%MAX_MSG_IN_QUEUE].uParam.sMhmeTmerI
nd,&sMhmeTmerInd,sizeof(NET_MhmeTmerInd_s));
0710                 #ifdef DEBUG
0711                 vPrintf("MHME.vTickTimerISR: Message received. (%s: %d)\n",__FILE__,__LINE__);
0712                 #endif
0713             }
0714             else
0715             {
0716                 #ifdef DEBUG
0717                 vPrintf("MHME.vTickTimerISR: To many messages in the message queue, so current message
will be ignored. (%s: %d)\n",__FILE__,__LINE__);
0718                 #endif
0719             }
0720         }
0721         u32Filter=u32Filter<<1;
0722     }
0723 }
```

XVII. mhme.h

```
0001 #ifndef _MHME_H
0002 #define _MHME_H
0003
0004 #include <AppHardwareApi.h>
0005 #include <AppApi.h>
0006 #include <jendefs.h>
0007 #include <mac_sap.h>
0008 #include <mac_pib.h>
0009
0010 #include "mesh.h"
```

```
0011
0012 #include "config.h"
0013
0014 #define     MAX_CHILDREN_REPORT_RETRIES             10
0015
0016 #define         CONST_COORDINATOR_CAPABILITY            TRUE                /**< This constant indicates
whether the device is capable of acting as a mesh coordinator (MC). It is set at device build time. TRUE
indicates it is capable of being an MC while FALSE indicates the device is not capable of being an MC. */
0017 #define         CONST_BROADCAST_ADDRESS                 0xffff              /**< The short logical
broadcast address at mesh sublayer. */
0018 #define         CONST_MAX_MESH_HEADER_LENGTH            0x12                /**< The maximum length in
octets of the mesh header (18 bytes). */
0019 #define         CONST_TIME_UNIT                         1                   /**< Time unit for mesh
sublayer power saving. The unit is millisecond. */
0020 #define         CONST_BASE_ACTIVE_DURATION              MESHCTIMEUNIT*5     /**< The time basis of the
active duration when the active order is equal to zero. */
0021 #define         CONST_MAX_LOST_SYNCHRONIZATION          3                   /**< The number of
consecutive synchronization failure. */
0022 #define         CONST_RESERVATION_SLOT_DURATION         625                 /**< The time period of
reserved data transmission in inactive duration of SES time structure (in MAC sublayer symbol periods). */
0023
0024
0025 #define         MESH_PIB_MIN                            MESH_NB_OF_CHILDREN
0026 #define         MESH_PIB_MAX                            MESH_REJOIN_TIMER
0027 #define     MESH_PIB_RANGE                          MESH_PIB_MAX-MESH_PIB_MIN
0028
0029
0030 #define     MAC_PIB_MIN                             0x40
0031 #define     MAC_PIB_MAX                             0x55
0032
0033
0034 #define         MESH_NB_OF_CHILDREN                     0xA1
0035 #define         MESH_CAPABILITY_INFORMATION             0xA2
0036 #define         MESH_TTL_OF_HELLO                       0xA3
0037 #define         MESH_TREE_LEVEL                         0xA4
0038 #define         MESH_PANID                              0xA5
0039 #define         MESH_NEIGHBOR_LIST                      0xA6
0040 #define         MESH_DEVICE_TYPE                        0xA7
0041 #define         MESH_SEQUENCE_NUMBER                    0xA8
0042 #define         MESH_NETWORK_ADDRESS                    0xA9
0043 #define         MESH_GROUP_COMM_TABLE                   0xAA
0044 #define         MESH_ADDRESS_MAPPING                    0xAB
0045 #define         MESH_ACCEPT_MESH_DEVICE                 0xAC
0046 #define         MESH_ACCEPT_END_DEVICE                  0xAD
0047 #define         MESH_CHILD_REPORT_TIME                  0xAE
0048 #define         MESH_PROBE_INTERVAL                     0xAF
0049 #define         MESH_MAX_PROBE_NUM                      0xB0
0050 #define         MESH_MAX_PROBE_INTERVAL                 0xB1
0051 #define         MESH_MAX_MULTICAST_JOIN_ATTEMPTS        0xB2
0052 #define         MESH_RB_CAST_TX_TIMER                   0xB3
0053 #define         MESH_RB_CAST_RX_TIMER                   0xB4
0054 #define         MESH_MAX_RB_CAST_TRIALS                 0xB5
0055 #define         MESH_ASES_ON                            0xB6
0056 #define         MESH_ASES_EXPECTED                      0xB7
0057 #define         MESH_WAKEUP_ORDER                       0xB8
0058 #define         MESH_ACTIVE_ORDER                       0xB9
0059 #define         MESH_DEST_ACTIVE_ORDER                  0xBA
0060 #define         MESH_EREQ_TIME                          0xBB
0061 #define         MESH_EREP_TIME                          0xBC
0062 #define         MESH_DATA_TIME                          0xBD
0063 #define         MESH_MAX_NUM_ASES_RETRIES               0xBE
0064 #define         MESH_SES_ON                             0xBF
0065 #define         MESH_SES_EXPECTED                       0xC0
0066 #define         MESH_SYNC_INTERVAL                      0xC1
0067 #define         MESH_MAX_SYNC_REQUEST_ATTEMPTS          0xC2
```

```
0068 #define        MESH_SYNC_REPLY_WAIT_TIME               0xC3
0069 #define        MESH_FIRST_TX_SYNC_TIME                 0xC4
0070 #define        MESH_FIRST_RX_SYNC_TIME                 0xC5
0071 #define        MESH_SECOND_RX_SYNC_TIME                0xC6
0072 #define        MESH_REGION_SYNCHRONIZAER_ON            undef
0073 #define        MESH_EXTENDED_NEIGHBOR_HOP_DISTANCE     0xC7
0074 #define        MESH_REJOIN_TIMER                       0xC8
0075
0076
0077
0078 #define    CFM_STATUS_INVALID_REQUEST                  0xb1        /**<  A requested service was
invalid. */
0079 #define    CFM_STATUS_NOT_PERMITTED                    0xb2        /**<  Join was failed because it was
not permitted. */
0080 #define    CFM_STATUS_NO_NETWORKS                      0xb3        /**<  No network was found. */
0081 #define    CFM_STATUS_READ_ONLY                        0xb4        /**<  The attribute to be set is a
read-only attribute. */
0082 #define    CFM_STATUS_RECEIVE_SYNC_LOSS                0xb5        /**<  A synchronization loss notify
frame was received. */
0083 #define    CFM_STATUS_RECEIVE_SYNC_RESPONSE            0xb6        /**<  A synchronization response
frame was received. */
0084 #define    CFM_STATUS_STARTUP_FAILURE                  0xb7        /**<  A network could not be started
because of no suitable channel or PAN identifier. */
0085 #define    CFM_STATUS_SYNC_FAILURE                     0xb8        /**<  Synchronization request was not
successful because at least one synchronization response frame was not received. */
0086 #define    CFM_STATUS_SYNC_LOSS                        0xb9        /**<  A synchronization request frame
was not received at the scheduled time. */
0087 #define    CFM_STATUS_SYNC_SUCCESS                     0xba        /**<  SYNC request was successful. */
0088 #define    CFM_STATUS_TRACEROUTE_TIMEOUT               0xbb        /**<  Traceroute reply command frame
was not arrived within the requested response time. */
0089 #define    CFM_STATUS_TRACEROUTE_UNREACHABLE           0xbc        /**<  Traceroute failed because the
destination device is unreachable. */
0090 #define    CFM_STATUS_UNKNOWN_CHILD_DEVICE             0xbd        /**<  A device requested to leave is
not a child device. */
0091 #define    CFM_STATUS_DEFERED                          0xbe        /**<  Notify an asyncronous confirm.
*/
0092 #define NET_MHME_DCFM_DISCOVER                     0xbf
0093 #define NET_MHME_DCFM_JOIN                         0xcf
0094 #define NET_MHME_IND_JOIN                          0xdf
0095 #define NET_MHME_IND_TIMER                         0xef
0096 #define NET_MHME_DCFM_LEAVE                        0xf1
0097 #define NET_MHME_IND_LEAVE                         0xf2
0098 #define NET_MHME_IND_LEFT                          0xf3
0099
0100
0101
0102
0103 #define NET_MESH_CHILDREN_REPORT_TIMER             5
0104 #define NET_MESH_HELLO_TIMER                       10
0105 #define NET_MESH_LEAVE_TIMER                       20
0106 #define NET_MESH_ADDRESS_ASSIGNMNET_TIMER          5
0107
0108 #define PLACEBITMAP(x,i)
(*(uint32*)((uint8*)(&x)+((i/32)*4))|=((uint32)0x00000001<<i%32))
0109 #define READBITMAP(x,i)
((*(uint32*)((uint8*)(&x)+((i/32)*4))&((uint32)0x00000001<<i%32))!=0x00)
0110 #define UNPLACEBITMAP(x,i)
(*(uint32*)((uint8*)(&x)+((i/32)*4))&=((uint32)~(0x00000001<<i%32)))
0111 #define WRITEBITMAPSINGLE(x,i)                     (vPrintf("%d",READBITMAP(x,i)?1:0))
0112 #define WRITEBITMAP(x)                             {uint8 j=0; for(j=0;j<sizeof(NET_BitMap_s)*8;j++)
WRITEBITMAPSINGLE(x,j); vPrintf("\n");}
0113
0114 typedef enum
0115 {
0116
```

```
0117     LINK_QUALITY,    /**< The neighbor with highest link quality is reported. */
0118     TREE_LEVEL       /**< The neighbor with lowest tree level is reported. */
0119
0120 }NET_MhmeReportCriteria_e;
0121
0122
0123
0124 typedef enum
0125 {
0126
0127     NEW,
0128     UPDATE,
0129     REJOIN
0130
0131 }NET_MhmeChildAddressAssignmentStatus_e;
0132
0133
0134 typedef enum
0135 {
0136
0137     SUCCESS,
0138     DCFM_JOIN,
0139     UNSUPPORTED_ATTRIBUTE,
0140     INVALID_PARAMETER,
0141     INVALID_REQUEST,
0142     UNKNOWN_CHILD_DEVICE,
0143     NOT_PERMITED
0144
0145 }NET_MhmeStatus_e;
0146
0147
0148 typedef enum
0149 {
0150
0151     NET_MHME_REQ_DISCOVER,
0152     NET_MHME_REQ_START_NETWORK,
0153     NET_MHME_REQ_START_DEVICE,
0154     NET_MHME_REQ_JOIN,
0155     NET_MHME_REQ_LEAVE,
0156     NET_MHME_REQ_RESET,
0157     NET_MHME_REQ_GET,
0158     NET_MHME_REQ_SET
0159
0160 }NET_MhmeReqRspType_e;
0161
0162
0163 typedef enum
0164 {
0165
0166     PARENT,
0167     CHILD,
0168     SIBLING_DEVICE,
0169     SYNC_PARENT,
0170     SYNC_CHILD,
0171     ORPHAN,
0172     NO_RELATIONSHIP
0173
0174 }NET_NeighborRelationship_e;
0175
0176
0177 typedef enum
0178 {
0179
0180     UNKNOWN,
0181     DOWN,
```

```
0182     LEFT
0183
0184 }NET_NeighborStatus_e;
0185
0186
0187 typedef enum
0188 {
0189     END_DEVICE,
0190     MESH_DEVICE
0191 }NET_DeviceType_e;
0192
0193
0194 typedef struct
0195 {
0196
0197     uint32                      u32ScanChannels;    /**< The 27 bits (b0, b1,... b26) indicate which
channels are to be scanned (1 = scan, 0 = do not scan) for each of the 27 channels supported by the
ChannelPage parameter as defined in IEEE Std 802.15.4-2006. */
0198     uint8                       u8ScanDuration;     /**< A value used to calculate the length of time to
spend scanning each channel. */
0199     uint8                       u8ChannelPage;      /**< The channel page on which to perform the scan.
*/
0200     NET_MhmeReportCriteria_e    eReportCriteria;    /**< The field indicates which criterion is used to
select the best neighbor to be reported to the application layer. */
0201     uint8                       u8Pad;
0202
0203 }NET_MhmeReqDiscover_s;
0204
0205
0206 typedef struct
0207 {
0208
0209     uint16          u16PanId;                   /**< The 16-bit PAN identifier of the network
discovered. */
0210     uint8           u8AddrMode;              /**< 2 if 16-bit short address or 3 if it is a 64-bit
extended address*/
0211     uint8           u8LogicalChannel;           /**< The current logical channel occupied by the
network. */
0212     uint8           u8ChannelPage;              /**< The current channel page occupied by the network.
*/
0213     uint8           u8MeshVersion;              /**< The version of the mesh sublayer protocol in use in
the discovered network. */
0214     uint8           u8BeaconOrder;              /**< This specifies how often the MAC sublayer beacon is
to be transmitted by a given device on the network. For this version of the recommended practice, the value
is always set to 0x0f indicating no periodic beacons are transmitted. */
0215     uint8           u8SuperFrameOrder;          /**< For beacon-oriented networks, that is, beacon order
< 15, this specifies the length of the active period of the superframe. For this version of the recommended
practice, the value is always set to 0x0f indicating no periodic beacons are transmitted. */
0216     uint8           u8LinkQuality;              /**< The LQI value at which the network beacon was
received. Lower values represent lower LQI. */
0217     uint8           u8TreeLevel;                /**< The tree level of the beacon sender. */
0218     uint16          u16Pad1;
0219     MAC_Addr_u      uAddr;
0220     uint8           bAcceptMeshDevice   : 1;    /**< This field indicates whether the mesh network
accepts new mesh devices. A value of TRUE indicates the device is capable of accepting join requests from
other mesh devices; the value should be set to FALSE otherwise. */
0221     uint8           bAcceptEndDevice    : 1;    /**< This field indicates whether the mesh network
accepts new end devices. A value of TRUE indicates the device is capable of accepting join requests from end
devices; the value should be set to FALSE otherwise. */
0222     uint8           bSyncEnergySaving   : 1;    /**< This field indicates whether the optional
synchronous energy saving feature is supported. A value of TRUE indicates the device is capable of supporting
synchronous energy saving; the value should be set to FALSE otherwise. */
0223     uint8           bAsyncEnergySaving : 1;     /**< This field indicates whether the optional
asynchronous energy saving feature is supported. A value of TRUE indicates the device is capable of
supporting asynchronous energy saving; the value should be set to FALSE otherwise. */
```

```
0224     uint8           u8Pad1             : 4;
0225     uint8           u8Pad2;
0226     uint16          u16Pad2;
0227 }NET_MeshDescriptor_s;
0228
0229
0230 typedef struct
0231 {
0232
0233     uint8                   u8Status;                                   /**< Defined in 7.1.11.2 of
IEEE Std 802.15.4-2006. */
0234     uint8                   u8NetworkCount;                             /**< Gives the number of
networks discovered by the search. */
0235     uint16                  u16Pad;
0236     NET_MeshDescriptor_s    psMeshDescriptorList[MAC_MAX_SCAN_PAN_DESCRS];  /**< A list of descriptors,
one for each of the mesh network discovered. */
0237
0238 }NET_MhmeCfmDiscover_s;
0239
0240
0241 typedef struct
0242 {
0243
0244     uint32      u32ScanChannels;        /**< The 27 bits (b0, b1,... b26) indicate which channels are to
be scanned (1 = scan, 0 = do not scan) for each of the 27 channels supported by the ChannelPage parameter as
defined in IEEE Std 802.15.4-2006. */
0245     uint8       u8ScanDuration;         /**< A value used to calculate the length of time to spend
scanning each channel. */
0246     uint8       u8ChannelPage;          /**< The channel page on which to perform the scan. */
0247     uint8       u8BeaconOrder;          /**< The beacon order of the network that the higher layers wish
to form. For this version of the recommended practice, the value is always set to 0x0f indicating no periodic
beacons are transmitted. */
0248     uint8       u8SuperFrameOrder;      /**< The superframe order of the network that the higher layers
wish to form. For this version of the recommended practice, the value is always set to 0x0f indicating no
periodic beacons are transmitted. */
0249
0250 }NET_MhmeReqStartNetwork_s;
0251
0252
0253 typedef struct
0254 {
0255
0256     uint8   u8Status;   /**< The result of the attempt to initialize a mesh coordinator. */
0257     uint8   u8Pad;
0258     uint16  u16Pad;
0259
0260 }NET_MhmeCfmStartNetwork_s;
0261
0262
0263 typedef struct
0264 {
0265
0266     uint8   u8BeaconOrder;          /**< The beacon order of the network that the higher layers wish to
form. For this version of the recommended practice, the value is always set to 0x0f indicating no periodic
beacons are transmitted. */
0267     uint8   u8SuperFrameOrder;      /**< The superframe order of the network that the higher layers wish
to form. For this version of the recommended practice, the value is always set to 0x0f indicating no periodic
beacons are transmitted. */
0268     uint16  u16Pad;
0269
0270 }NET_MhmeReqStartDevice_s;
0271
0272
0273 typedef struct
0274 {
```

```
0275
0276     uint8   u8Status;   /**< The result of the attempt to initialize a mesh device. */
0277     uint8   u8Pad;
0278     uint16  u16Pad;
0279
0280 }NET_MhmeCfmStartDevice_s;
0281
0282
0283 typedef struct
0284 {
0285
0286     uint8   u8DirectJoin;           /**< The value is set to TRUE if direct joining is chosen;
otherwise, its value is FALSE. */
0287     uint8   u8AddrMode;
0288     uint8   u8RejoinNetwork;        /**< This parameter controls the method of joining the network. */
0289     uint8   u8JoinAsMeshDevice;     /**< The parameter is set to TRUE if the device is going to function
as a mesh device; it is set to FALSE if the device is going to function as an end device. */
0290     MAC_Addr_u uParentDevAddr;
0291     uint32  u32ScanChannels;        /**< The 27 bits (b0, b1,... b26) indicate which channels are to be
scanned (1 = scan, 0 = do not scan) for each of the 27 channels supported by the ChannelPage parameter. This
field will be read only when the DirectJoin parameter has a value equal to FALSE. */
0292     uint16  u16PanId;               /**< The 16-bit PAN identifier of the network to join. This field
will be read only when the DirectJoin parameter has a value equal to TRUE. */
0293     uint8   u8ScanDuration;         /**< A value used to calculate the length of time to spend scanning
each channel. */
0294     uint8   u8ChannelPage;          /**< The channel page on which to perform the scan. This field will
be read only when the DirectJoin parameter has a value equal to FALSE. */
0295     uint8   u8DeviceType;           /**< This field is set to TRUE if the joining device is a mesh
device. Otherwise, it is set to FALSE. */
0296     uint8   u8PowerSource;          /**< This field is set to TRUE if it is mainspowered. Otherwise, it
is set to FALSE. */
0297     uint8   u8ReceiverOnWhenIdle;   /**< This field is set to TRUE if the receiver is enabled when the
device is idle. Otherwise, it is set to FALSE. */
0298     uint8   u8AllocateAddress;      /**< This field is always set to TRUE in the implementations of this
recommended practice, indicating that the joining device should be issued a 16-bit network address. */
0299
0300 }NET_MhmeReqJoin_s;
0301
0302
0303 typedef struct
0304 {
0305
0306     uint16          u16NetworkAddress;          /**< When a short network address of an entity has been
assigned, this will be the short address that has been added to the network. Otherwise, the value of this
parameter will equal to 0xfffe indicating the short address has not been assigned and the device can only be
reached by its 64-bit extended address. */
0307     uint16          u16CapabilityInformation;   /**< Specifies the operational capabilities of the
joining device. */
0308     MAC_ExtAddr_s   sExtendedAddress;           /**< The 64-bit IEEE address of an entity that has been
added to the network. */
0309     uint8           u8RejoinNetwork;            /**< The RejoinNetwork parameter indicating the method
used to join the network. */
0310     uint8           u8Pad;
0311     uint16          u16Pad;
0312
0313 }NET_MhmeIndJoin_s;
0314
0315
0316 typedef struct
0317 {
0318
0319     uint8       u8Status;           /**< The status of the corresponding request. */
0320     uint8       u8ChannelPage;      /**< The channel page on which the ActiveChannel was found. */
0321     uint8       u8ActiveChannel;    /**< The value of phyCurrentChannel parameter which is equal to the
current channel of the network that has been joined. */
```

```
0322     uint8       u8Pad;
0323     uint16      u16NetworkAddress;  /**< The 16-bit network address that was allocated to this device.
This parameter will be equal to 0xffff if the join attempt was unsuccessful. */
0324     uint16      u16PanId;           /**< The 16-bit PAN identifier of the network of which the device is
now a member. */
0325
0326 }NET_MhmeCfmJoin_s;
0327
0328
0329 typedef struct
0330 {
0331     MAC_ExtAddr_s   sDeviceAddress;     /**< The 64-bit IEEE address of a child device to be removed
from the network. */
0332     uint8           u8RemoveSelf;       /**< This parameter has a value of TRUE if the device is asked
to remove itself from the network. Otherwise it has a value of FALSE. */
0333     uint8           u8RemoveChildren;   /**< This parameter has a value of TRUE if the device being
asked to leave the network is also being asked to remove its child devices, if any. Otherwise it has a value
of FALSE. */
0334     uint16          u16Pad;
0335
0336 }NET_MhmeReqLeave_s;
0337
0338
0339 typedef struct
0340 {
0341
0342     MAC_ExtAddr_s   sDeviceAddress;     /**< The 64-bit IEEE address of an entity that has removed
itself from the network. */
0343
0344 }NET_MhmeIndLeave_s;
0345
0346
0347 typedef struct
0348 {
0349
0350     uint8           u8Status;           /**< The status of the corresponding request. */
0351     uint8           u8Pad;
0352     uint16          u16Pad;
0353     MAC_ExtAddr_s   sDeviceAddress;     /**< The 64-bit IEEE address in the request to which this is a
confirmation or null if the device requested to remove itself from the network. */
0354
0355 }NET_MhmeCfmLeave_s;
0356
0357 typedef struct
0358 {
0359     uint8 u8TriggeredTimer;
0360     uint8           u8Pad;
0361     uint16          u16Pad;
0362
0363 }NET_MhmeIndTmer_s;
0364
0365
0366 typedef struct
0367 {
0368
0369     uint8   u8Status;   /**< The result of the reset operation. */
0370     uint8   u8Pad;
0371     uint16  u16Pad;
0372
0373 }NET_MhmeCfmReset_s;
0374
0375
0376 typedef struct
0377 {
0378
```

```
0379     uint8   u8MeshIBAttribute;      /**< The identifier of the MeshIB attribute to read. */
0380     uint8   u8Pad;
0381     uint16  u16Pad;
0382
0383 }NET_MhmeReqGet_s;
0384
0385
0386 typedef struct
0387 {
0388
0389     NET_MhmeStatus_e    eStatus;               /**< The results of the request to read an MeshIB
attribute value. */
0390     uint8               u8MibAttribute;        /**< The identifier of the MeshIB attribute that was
read. */
0391     uint8               u8Pad;
0392     uint16              u16MibAttributeLength; /**< The length, in octets, of the attribute value being
returned. */
0393     uint8*              psMibAttributeValue;   /**< The value of the MeshIB attribute that was read. */
0394
0395 }NET_MhmeCfmGet_s;
0396
0397
0398 typedef struct
0399 {
0400     uint8   u8Pad;
0401     uint8   u8MibAttribute;         /**< The identifier of the MeshIB attribute to be written. */
0402     uint16  u16MibAttributeLength;  /**< The length, in octets, of the attribute value being set. */
0403     uint8*  psMibAttributeValue;    /**< The value of the MeshIB attribute that should be written. */
0404
0405 }NET_MhmeReqSet_s;
0406
0407
0408 typedef struct
0409 {
0410
0411     NET_MhmeStatus_e    u8Status;        /**< The results of the request to write the MeshIB attribute.
*/
0412     uint8               u8MibAttribute; /**< The identifier of the MeshIB attribute that was written. */
0413     uint16              u16Pad;
0414
0415 }NET_MhmeCfmSet_s;
0416
0417
0418 typedef union
0419 {
0420
0421     NET_MhmeCfmDiscover_s       sCfmDiscover;
0422     NET_MhmeCfmStartNetwork_s   sCfmStartNetwork;
0423     NET_MhmeCfmStartDevice_s    sCfmStartDevice;
0424     NET_MhmeCfmJoin_s           sCfmJoin;
0425     NET_MhmeCfmLeave_s          sCfmLeave;
0426     NET_MhmeCfmReset_s          sCfmReset;
0427     NET_MhmeCfmGet_s            sCfmGet;
0428     NET_MhmeCfmSet_s            sCfmSet;
0429
0430 }NET_MhmeSyncCfmParam_u;
0431
0432
0433 typedef union
0434 {
0435
0436     NET_MhmeCfmDiscover_s       sCfmDiscover;
0437     NET_MhmeCfmStartNetwork_s   sCfmStartNetwork;
0438     NET_MhmeCfmStartDevice_s    sCfmStartDevice;
0439     NET_MhmeCfmJoin_s           sCfmJoin;
```

```
0440     NET_MhmeCfmLeave_s          sCfmLeave;
0441     NET_MhmeCfmReset_s          sCfmReset;
0442     NET_MhmeCfmGet_s            sCfmGet;
0443     NET_MhmeCfmSet_s            sCfmSet;
0444     NET_MhmeIndJoin_s           sIndJoin;
0445     NET_MhmeIndLeave_s          sIndLeave;
0446     NET_MhmeIndTmer_s           sIndTmer;
0447
0448 }NET_MhmeDcfmIndParam_u;
0449
0450
0451 typedef union
0452 {
0453
0454     NET_MhmeReqDiscover_s       sReqDiscover;
0455     NET_MhmeReqStartNetwork_s   sReqStartNetwork;
0456     NET_MhmeReqStartDevice_s    sReqStartDevice;
0457     NET_MhmeReqJoin_s           sReqJoin;
0458     NET_MhmeReqLeave_s          sReqLeave;
0459     NET_MhmeReqGet_s            sReqGet;
0460     NET_MhmeReqSet_s            sReqSet;
0461
0462 }NET_MhmeReqRspParam_u;
0463
0464
0465 typedef struct
0466 {
0467
0468     uint8                       u8Type;
0469     uint8                       u8Pad;
0470     uint16                      u16Pad;
0471     NET_MhmeReqRspParam_u       uParam;
0472
0473 }NET_MhmeReqRsp_s;
0474
0475
0476 typedef struct
0477 {
0478     uint8                   u8Status;
0479     uint8                   u8Pad;
0480     uint16                  u16Pad;
0481     NET_MhmeSyncCfmParam_u  uParam;
0482
0483 }NET_MhmeSyncCfm_s;
0484
0485
0486 typedef struct
0487 {
0488
0489     uint8                   u8Type;
0490     uint8                   u8Pad;
0491     uint16                  u16Pad;
0492     NET_MhmeDcfmIndParam_u  uParam;
0493
0494 }NET_MhmeDcfmInd_s;
0495
0496
0497 typedef struct
0498 {
0499
0500     MAC_ExtAddr_s   sExtAddr;       /**< 64-bit IEEE Address. */
0501     uint16          u16ShortAddr;   /**< 16-bit Short Address. */
0502     uint16          u16Pad;
0503
0504 }NET_AddrMapping_s;
```

```
0505
0506 /** Mesh Discovery Table */
0507 typedef struct
0508 {
0509     NET_MeshDescriptor_s        asMeshDescriptors[MAC_MAX_SCAN_PAN_DESCRS];
0510     NET_MhmeReportCriteria_e    eReportCriteria;
0511     uint8                       u8Pad;
0512     uint16                      u16Pad;
0513 }NET_mdt_s;
0514
0515 NET_mdt_s sMdt;
0516
0517
0518 /** Neighbor list */
0519 typedef struct
0520 {
0521     MAC_ExtAddr_s sExt;
0522     uint16        u16BeginningAddress;    /**< The beginning address of the address block assigned to
this neighbor. It is also the address of this neighbor. */
0523     uint16        u16EndingAddress;       /**< The ending address of the address block assigned to this
neighbor. */
0524     uint8         u8TreeLevel;            /**< The tree level of this neighbor. */
0525     uint8         u8LinkQuality;          /**< The link quality from the neighbor to this device. It is
measured when the data frames are received from the neighbor and is ranged from from 0 (the lowest) to 255
(the highest). */
0526     uint8         u8Relationship;         /**< The relationship between this device and the neighbor. */
0527     uint8         u8ReliableBroadcast;    /**< This field indicates whether the neighbor device supports
reliable broadcast. */
0528     uint8         u8Status;               /**< The status of this neighbor. */
0529     uint8         u8NumberOfOps;          /**< The hop distance between this device and the neighbor. It
is calculated using the connectivity matrix described in the next subclause. */
0530     uint8         u8NbOfHello;            /**< Hello frame used to do this update*/
0531     int8          i8RejoinTimer;
0532     int8          i8MsduHandleLeave;
0533     uint8         bRemoveChildren;
0534     uint8         u8NbOfLeaveRetries;
0535     uint8         u8ad;
0536     uint16*       pu16GroupMembership;   /**< A list of multicast groups of which the neighbor
participates in. */
0537     uint16        u16RequestedAddresses;
0538     uint16        u16Pad;
0539     NET_MhmeChildAddressAssignmentStatus_e   eStatus;
0540
0541 }NET_Neighbor_s;
0542
0543
0544
0545 typedef struct
0546 {
0547     uint32 u32A;
0548     uint32 u32B;
0549     uint32 u32C;
0550 }
0551 NET_BitMap_s;
0552
0553 typedef struct
0554 {
0555     uint16  u16BeginningBlock;
0556     uint16  u16NumberOfBlocks;
0557 }
0558 NET_FreeBlock_s;
0559
0560
0561
0562 /** Mesh information base */
```

```
0563 typedef struct
0564 {
0565     uint8                   u8NbOfChildren;                                 /**< The number of devices
that have joined this device. */
0566     uint8                   u8CapabilityInformation;                        /**< This field indicates
the capability of the device. */
0567     uint8                   u8TtlOfHello;                                   /**< The number of hops the
hello command frames are allowed to travel. */
0568     uint8                   u8TreeLevel;                                    /**< The number of hops this
device is from the Mesh coordinator through its parent. */
0569     uint8                   u8DeviceType;                                   /**< This attribute is set
to END_DEVICE if the device is an End Device, MESH_DEVICE if the device is a Mesh Device. */
0570     uint8                   u8SequenceNumber;                               /**< A sequence number used
to identify outgoing frames. */
0571     uint8                   u8AcceptMeshDevice;                             /**< This attribute
indicates whether this device allows other mesh devices to join as its children. */
0572     uint8                   u8AcceptEndDevice;                              /**< This attribute
indicates whether this device allows other end devices to join as its children. */
0573     uint8                   u8MaxRbCastTrials;                              /**< This attribute
indicates the maximum number of broadcasting trials before issuing MHME-LEAVE-indication primitive to avoid
possible unreachability of one or more neighbor devices. */
0574     uint8                   u8AsesOn;                                       /**< This attribute
indicates whether this device is in ASES mode. */
0575     uint8                   u8AsesExpected;                                 /**< This attribute
indicates whether this device is capable of ASES. */
0576     uint8                   u8WakeupOrder;                                  /**< This specifies how
often a device transmits its wakeup notification.\n If the wakeup order, WO = 15, the device will not
transmit a periodic wakeup notification. */
0577     uint8                   u8ActiveOrder;                                  /**< The length of the
active duration within a wakeup interval, including the wakeup notification. If WO = 15, this value is
ignored. */
0578     uint8                   u8DestActiveOrder;                              /**< The length of the
active duration of the destination device. When meshASESOn is FALSE, this value is ignored. */
0579     uint8                   u8EreqTime;                                     /**< The maximum number of
meshcTimeUnits to wait for the next EREQ or a data frame following an EREQ. */
0580     uint8                   u8ErepTime;                                     /**< The maximum number of
meshcTimeUnits to wait for an EREP after transmitting EREQ. */
0581     uint8                   u8DataTime;                                     /**< The maximum number of
meshcTimeUnits to wait for the EREQ or a Data frame following a data frame. */
0582     uint8                   u8MaxNumAsesRetries;                            /**< The maximum number of
retransmissions in ASES. */
0583     uint8                   u8SesOn;                                        /**< This attribute
indicates whether this node is in SES mode. */
0584     uint8                   u8SesExpected;                                  /**< This attribute
indicates whether this node is capable of SES. */
0585     uint8                   u8SyncInterval;                                 /**< The number of wakeup
interval times for periodic synchronization. */
0586     uint8                   u8MaxSyncRequestAttempts;                       /**< The maximum number of
times a node should try to transmit a sync request frame before considering the sync procedure failure. */
0587     uint8                   u8SyncReplyWaitTime;                            /**< The maximum number of
meshTimeUnits to wait for synchronization reply frame after transmitting synchronization request frame. */
0588     uint8                   u8MaxProbeNum;                                  /**< The maximum number an
unknown neighbor is going to be probed. */
0589     uint8                   u8RegionSynchronizaerOn;                        /**< This attribute
indicates whether this node is a region synchronizer. */
0590     uint8                   u8ExtendedNeighborHopDistance;                  /**< The predetermined hop
distance between a device and its extended neighbors. */
0591     uint16                  u16PanId;                                       /**< The PAN ID of this mesh
network. It should have the same value of the macPANId. */
0592     uint16                  u16ChildReportTime;                             /**< This attribute
indicates the time in seconds to start reporting the number of child devices after this device has
successfully joined the network. */
0593     uint16                  u16ProbeInterval;                               /**< This attribute sets the
probe interval in seconds for a neighbor if its status in the neighbor list is unknown. */
```

```
0594     uint16                  u16RbCastTxTimer;                               /**< This attribute
indicates the time in seconds that a broadcasting device needs to wait before rebroadcasting the data frame.
*/
0595     uint16                  u16RbCastRxTimer;                               /**< This attribute
indicates the maximum waiting time in seconds that the receiving device forwards the broadcast data frame. */
0596     uint16                  u16MaxProbeInterval;                            /**< This attribute sets the
maximum probe interval in seconds for a neighbor if its status in the neighbor list is unknown. */
0597     uint16                  u16NetworkAddress;                              /**< The mesh sublayer
address of this device.\n This attribute reflects the value of the MAC PIB attribute macShortAddress as
defined in IEEE Std 802.15.4-2006 and any changes made by the higher layer will be reflected in the MAC PIB
attribute value as well. */
0598     uint16                  u16RejoinTimer;                                 /**< The time between a
child device leaves the network and the child device is deleted from the neighbor list. */
0599     uint8                   u8MaxMulticastJoinAttempts;                     /**< The maximum number of
attempts a device should try to join a multicast group before considering the joining process a failure. */
0600     uint8                   u8Pad;
0601     uint8*                  psGroupCommTable;                               /**< TODO: NOT IMPLEMENTED,
This table records all multicast addresses of which this device is a member and its status in the group. */
0602     uint32                  u32FirstTxSyncTime;                             /**< The time that the
device transmitted its first synchronization request frame, in MAC symbol periods. */
0603     uint32                  u32FirstRxSyncTime;                             /**< The time that the
device received its first synchronization request frame, in MAC symbol periods. */
0604     uint32                  u32SecondRxSyncTime;                            /**< The time that the
device received its second synchronization request frame, in MAC symbol periods. */
0605     NET_Neighbor_s          psNeighborList[MAX_NEIGHBORS];                  /**< This attribute contains
the information of neighbors of the device. */
0606     NET_AddrMapping_s       sAddressMapping;                                /**< This attribute maps 64-
bit IEEE addresses to 16-bit short address. */
0607     NET_BitMap_s            sBitMapReadOnly;
0608 }NET_MeshInfo_s;
0609
0610 typedef struct
0611 {
0612     uint8               bIsCoordinator;
0613     uint8               u8LeftAddr;
0614     uint8               u8NbOfChildReportTimers;
0615     uint8               u8NbOfChildrenReports;
0616     int8                i8ChilderReportTimer;
0617     int8                i8AddressAssignmentTimer;
0618     int8                i8HelloTimer;
0619     int8                i8LeaveTimer;
0620     uint8               u8NbOfHello;
0621     uint8               u8MhmeHandle;
0622     uint8               u8MeshHandle;
0623     uint8               p2bConnectivityMatrix[MAX_NEIGHBORS+1][MAX_NEIGHBORS+1];
0624     uint8               u8NbOfChildReportSendingTries;
0625     uint8               u8RetryTime;
0626     uint8               u8NbOfLeavingCommands;
0627     uint8               u8Pad;
0628     uint16              u16EndingAddress;
0629     uint16              u16AddressAssigner;
0630     uint16              u16BlockSize;
0631     uint16              u16FreeBlocks;
0632     uint16              u16AllocatedAddresses;
0633     uint16              u16Pad;
0634     uint16              u16NumberOfFreeBlocksInList;
0635     uint16              u16NumberOfRequestedAddresses;
0636     NET_BitMap_s        sBitMapLeave;
0637     NET_BitMap_s        sBitMapNew;
0638     NET_BitMap_s        sBitMapUpdate;
0639     NET_BitMap_s        sBitMapRejoin;
0640     NET_BitMap_s        sBitMapWaitingReport;
0641     NET_BitMap_s        sBitMapAddressAssignment;
0642     NET_BitMap_s        sBitMapNoNew;
0643     NET_FreeBlock_s     sFreeBlock[MAX_NEIGHBORS+1];
```

```
0644 }
0645 NET_MeshData_s;
0646
0647
0648 #define MAX_HND_IN_QUEUE 64
0649
0650 typedef struct
0651 {
0652     uint8   u8Type;
0653     uint8   u8Pad;
0654     uint16  u16Pad;
0655 }
0656 NET_MhmeMsduHandle_s;
0657
0658 typedef struct
0659 {
0660
0661     uint8                   u8HndInQueue;
0662     uint8                   u8IndexOfFirstFreeSpot;
0663     uint16                  u16Pad;
0664     uint64                  u64MhsduHandleStatus;
0665     NET_MhmeMsduHandle_s    asMhsduHandle[MAX_HND_IN_QUEUE];
0666 }
0667 NET_MhmeMhsduHandleQueue_s;
0668
0669
0670
0671 typedef struct
0672 {
0673
0674     uint8   u8Used;
0675     uint8   u8Pad;
0676     uint16  u16Pad;
0677     MAC_MlmeDcfmInd_s sMlmeDcfmInd;
0678
0679 }MAC_MlmeBuffer_s;
0680
0681
0682 PUBLIC void vNetApiMhmeRequest(NET_MhmeReqRsp_s *psMhmeReqRsp, NET_MhmeSyncCfm_s *psMhmeSyncCfm);
0683 PUBLIC void vInitMeshStack(void);
0684
0685 #define MAX_MSG_IN_QUEUE 64
0686
0687 typedef enum
0688 {
0689     NET_MHME_COMMD_EVENT,
0690     NET_MHME_TIMER_EVENT,
0691     NET_MHME_MLME_EVENT
0692 }
0693 NET_MhmeEventType_e;
0694
0695 typedef struct
0696 {
0697     uint8   u8TriggeredTimer;
0698     uint8   u8Pad;
0699     uint16  u16Pad;
0700 }
0701 NET_MhmeTmerInd_s;
0702
0703 typedef union
0704 {
0705     NET_MeshDcfmInd_s   sMeshDcfmInd;
0706     NET_MhmeTmerInd_s   sMhmeTmerInd;
0707     MAC_MlmeDcfmInd_s   sMlmeDcfmInd;
0708 }
```

```
0709 NET_MhmeEvent_u;
0710
0711 typedef struct
0712 {
0713     uint8               u8Type;
0714     uint8               u8Pad;
0715     uint16              u16Pad;
0716     NET_MhmeEvent_u     uParam;
0717 }
0718 NET_MhmeEvent_s;
0719
0720 typedef struct
0721 {
0722     uint8               u8MsgInQueue;
0723     uint8               u8IndexOfFirstNonProcessedMsg;
0724     uint8               u8IndexOfFirstFreeSpot;
0725     uint8               u8Pad;
0726     NET_MhmeEvent_s     asMhmeEventQueue[MAX_MSG_IN_QUEUE];
0727 }NET_msgQueue_s;
0728
0729 NET_msgQueue_s  sMhmeQueue;
0730
0731 typedef enum
0732 {
0733     MHME_BEGINNING,
0734     MHME_DISCOVERING,
0735     MHME_JOINING,
0736     MHME_ADDRESS_ASSIGNMENT,
0737     MHME_CHILD_REPORT,
0738     MHME_TOPOLOGY_DISCOVERY,
0739     MHME_LEAVING,
0740     MHME_NBNOTFOUND
0741 }NET_flagType_e;
0742
0743 volatile uint8 u8MhmeFlag;
0744
0745 uint16 u16MhmeGetMessage(NET_MhmeDcfmInd_s* psMhmeDcfmInd);
0746
0747 PUBLIC MAC_DcfmIndHdr_s *psMlmeDcfmIndGetBuf(void *pvParam);   /**< Provides a buffer to the MLME, used
to send deferred confirms and indications to the Network Layer (this application) (does this include hardware
interrupts ?) */
0748
0749
0750 PUBLIC void vMlmeDcfmIndPost(void *pvParam, MAC_DcfmIndHdr_s *psDcfmIndHdr);    /**< Called by the MLME
to send a confirm or indication to the network layer. It runs in the ISR context. */
0751 void vMhmeCommandPost(NET_MeshDcfmInd_s* psMeshDcfmInd);
0752 void vTickTimerISR(uint32 u32Device, uint32 u32ItemBitmap);
0753
0754
0755 /** Number of available MLME buffers **/
0756 #define N_MLME_BUFFERS  3
0757
0758
0759 #endif //_MHME_H
```

XVIII. MhmeControl.c

```
0001 #include "Mhme.h"
0002 #include "CommandFrames.h"
0003 #include "MhmeServices.h"
0004 #include "string.h"
0005
0006 #include "VirtualTimer.h"
0007
```

```
0008 #ifdef DEBUG
0009 #include "Debugger.h"
0010 #endif
0011
0012 #define SAME64ADDR(a,b) ((a.u32H==b.u32H)&&(a.u32L==b.u32L))
0013
0014 extern NET_MeshInfo_s sMeshInfo;
0015 extern NET_MeshData_s sMeshData;
0016 extern MAC_MlmeReqRsp_s sMlmeReqRsp;
0017 extern MAC_MlmeSyncCfm_s sMlmeSyncCfm;
0018 extern volatile NET_MhmeMhsduHandleQueue_s sMhmeMhsduHandleQueue;
0019 extern volatile bool bJoinNetworkStatus;
0020
0021 inline uint16 u16HdlMhmeCfmDiscover(NET_MhmeDcfmInd_s* psMhmeDcfmInd,MAC_MlmeDcfmInd_s* psMlmeDcfmInd)
0022 {
0023     uint16 u16NumberOfDiscoveredNetworks;
0024     NET_MhmeCfmDiscover_s sMhmeCfmDiscover;
0025
0026     //Register this function on the debugger
0027     #ifdef DEBUG
0028     u8StackPushIdentifier("vHdlMhmeReqDiscover",strlen("vHdlMhmeReqDiscover"),FALSE);
0029     #endif
0030
0031     if(psMlmeDcfmInd->uParam.sDcfmScan.u8Status!=MAC_ENUM_SUCCESS)
0032     {
0033         #ifdef DEBUG
0034         vStackPrintf(__FILE__,__LINE__,"No netowrks were found.");
0035         #endif
0036         sMhmeCfmDiscover.u8Status=CFM_STATUS_NO_NETWORKS;
0037     }
0038     else
0039     {
0040         u16NumberOfDiscoveredNetworks=psMlmeDcfmInd->uParam.sDcfmScan.u8ResultListSize;
0041
0042         switch(psMlmeDcfmInd->uParam.sDcfmScan.u8ScanType)
0043         {
0044             case MAC_MLME_SCAN_TYPE_ACTIVE:
0045             {
0046                 uint16 i;
0047                 #ifdef DEBUG
0048                 vStackPrintf(__FILE__,__LINE__,"Performing active scan result computation.");
0049                 for(i=0;i<u16NumberOfDiscoveredNetworks;i++)
0050                 {
0051                     vPrintf("Network %d:\n",i);
0052                     vPrintf("\t\tPAN id=%d\n",psMlmeDcfmInd-
>uParam.sDcfmScan.uList.asPanDescr[i].sCoord.u16PanId);
0053                     vPrintf("\t\tChannel=%d\n",psMlmeDcfmInd-
>uParam.sDcfmScan.uList.asPanDescr[i].u8LogicalChan);
0054                 }
0055                 #endif
0056
0057                 //Filling up the mesh discovering table
0058                 for(i=0;i<u16NumberOfDiscoveredNetworks;i++)
0059                 {
0060                     sMdt.asMeshDescriptors[i].u16PanId=psMlmeDcfmInd-
>uParam.sDcfmScan.uList.asPanDescr[i].sCoord.u16PanId;
0061                     sMdt.asMeshDescriptors[i].u8AddrMode=psMlmeDcfmInd-
>uParam.sDcfmScan.uList.asPanDescr[i].sCoord.u8AddrMode;
0062                     memset(&sMdt.asMeshDescriptors[i].uAddr,0,sizeof(MAC_Addr_u));
0063                     if(psMlmeDcfmInd->uParam.sDcfmScan.uList.asPanDescr[i].sCoord.u8AddrMode==2)
0064                     {
0065                         sMdt.asMeshDescriptors[i].u8AddrMode=2;
0066                         sMdt.asMeshDescriptors[i].uAddr.u16Short=psMlmeDcfmInd-
>uParam.sDcfmScan.uList.asPanDescr[i].sCoord.uAddr.u16Short;
0067                     }
```

```
0068                    else
0069                    {
0070                        sMdt.asMeshDescriptors[i].u8AddrMode=3;
0071                        memcpy(&sMdt.asMeshDescriptors[i].uAddr.sExt,&psMlmeDcfmInd-
>uParam.sDcfmScan.uList.asPanDescr[i].sCoord.uAddr.sExt,sizeof(MAC_ExtAddr_s));
0072                    }
0073                    sMdt.asMeshDescriptors[i].u8LogicalChannel=psMlmeDcfmInd-
>uParam.sDcfmScan.uList.asPanDescr[i].u8LogicalChan;
0074                    //sMdt.asMeshDescriptors[i].u8ChannelPage=1;
0075                    sMdt.asMeshDescriptors[i].u8MeshVersion=1;
0076                    sMdt.asMeshDescriptors[i].u8BeaconOrder=0x0f&(psMlmeDcfmInd-
>uParam.sDcfmScan.uList.asPanDescr[i].u16SuperframeSpec);
0077                    sMdt.asMeshDescriptors[i].u8SuperFrameOrder=0x0f&(psMlmeDcfmInd-
>uParam.sDcfmScan.uList.asPanDescr[i].u16SuperframeSpec>>4);
0078                    sMdt.asMeshDescriptors[i].u8LinkQuality=psMlmeDcfmInd-
>uParam.sDcfmScan.uList.asPanDescr[i].u8LinkQuality;
0079                    //sMdt.asMeshDescriptors[i].u8TreeLevel=0;
0080                    sMdt.asMeshDescriptors[i].bAcceptMeshDevice=\
0081                    sMdt.asMeshDescriptors[i].bAcceptEndDevice =0x01&(psMlmeDcfmInd-
>uParam.sDcfmScan.uList.asPanDescr[i].u16SuperframeSpec>>15);
0082                    sMdt.asMeshDescriptors[i].bSyncEnergySaving=\
0083                    sMdt.asMeshDescriptors[i].bAsyncEnergySaving=FALSE;
0084                }
0085
0086                //Deciding the best neigbors to be reported to the higer layer
0087                switch(sMdt.eReportCriteria)
0088                {
0089                    case LINK_QUALITY:
0090                    {
0091                        uint32  u32PanBitMap=0L;
0092                        uint32  u32Check;
0093                        uint16  u16CurrentPanID;
0094                        bool    bFaultOccurred=TRUE;
0095                        uint16  u16FirstFault=0;
0096                        uint16  counter=0;
0097
0098                        #ifdef DEBUG
0099                        vStackPrintf(__FILE__,__LINE__,"Link quality selected.");
0100                        #endif
0101
0102                        //Choosing the best PAN devices to join according to LINK_QUALITY criteria
0103
for(u16CurrentPanID=sMdt.asMeshDescriptors[u16FirstFault].u16PanId;(counter<MAC_MAX_SCAN_PAN_DESCRS)&&(bFault
Occurred);counter++)
0104                        {
0105                            bFaultOccurred=FALSE;
0106                            sMhmeCfmDiscover.psMeshDescriptorList[counter].u8LinkQuality=0x00;
0107
for(u32Check=0x01<<u16FirstFault,i=u16FirstFault;i<u16NumberOfDiscoveredNetworks;i++,u32Check=u32Check<<1)
0108                            {
0109                                if((u32Check&u32PanBitMap)==0x00)
0110                                {
0111                                    if(sMdt.asMeshDescriptors[i].u16PanId==u16CurrentPanID)
0112                                    {
0113
if(sMhmeCfmDiscover.psMeshDescriptorList[counter].u8LinkQuality<sMdt.asMeshDescriptors[i].u8LinkQuality)
0114                                        {
0115
sMhmeCfmDiscover.psMeshDescriptorList[counter]=sMdt.asMeshDescriptors[i];
0116                                        }
0117                                        u32PanBitMap=u32PanBitMap|u32Check;
0118                                    }
0119                                    else
0120                                    {
0121                                        if(!bFaultOccurred)
```

```
0122                                        {
0123                                            u16FirstFault=i;
0124                                            bFaultOccurred=TRUE;
0125                                        }
0126                                    }
0127                                }
0128                            }
0129                        }
0130
0131                        sMhmeCfmDiscover.u8Status=SUCCESS;
0132                        sMhmeCfmDiscover.u8NetworkCount=counter;
0133                        u8MhmeFlag=MHME_JOINING;
0134                    } break;
0135                    case TREE_LEVEL:
0136                    {
0137                        #ifdef DEBUG
0138                        vStackPrintf(__FILE__,__LINE__,"Tree level report type is not yet supported.");
0139                        #endif
0140                        sMhmeCfmDiscover.u8Status=UNSUPPORTED_ATTRIBUTE;
0141                    } break;
0142                    default:
0143                    {
0144                        #ifdef DEBUG
0145                        vStackPrintf(__FILE__,__LINE__,"Unhadle report type.");
0146                        #endif
0147                        sMhmeCfmDiscover.u8Status=UNSUPPORTED_ATTRIBUTE;
0148                    }
0149                }
0150            } break;
0151
0152            default:
0153            {
0154                #ifdef DEBUG
0155                vStackPrintf(__FILE__,__LINE__,"Unhandle scan confirm.");
0156                #endif
0157            }
0158        }
0159    }
0160
0161    //Delivering info to the higher layer
0162    psMhmeDcfmInd->u8Type=NET_MHME_DCFM_DISCOVER;
0163    memcpy(&psMhmeDcfmInd->uParam.sCfmDiscover,&sMhmeCfmDiscover,sizeof(NET_MhmeCfmDiscover_s));
0164
0165    //Deregister this function off the debugger
0166    #ifdef DEBUG
0167    vStackPopIdentifier();
0168    #endif
0169
0170    return 1;
0171 }
0172
0173
0174 inline uint16 u16HdlMhmeCfmJoin(NET_MhmeDcfmInd_s* psMhmeDcfmInd,MAC_MlmeDcfmInd_s* psMlmeDcfmInd)
0175 {
0176     NET_MhmeCfmJoin_s   sMhmeCfmJoin;
0177
0178     if(psMlmeDcfmInd->uParam.sDcfmAssociate.u8Status == MAC_ENUM_SUCCESS)
0179     {
0180         //Check if an address was given to the child, otherwise, try to join
0181         if(psMlmeDcfmInd->uParam.sDcfmAssociate.u16AssocShortAddr == 0xffff)
0182         {
0183             #ifdef DEBUG
0184             vPrintf("%s.Join failed %s:%d\n",__FUNCTION__,__FILE__,__LINE__);
0185             #endif
0186
```

```
0187            sMhmeCfmJoin.u8Status=NOT_PERMITED;
0188            psMhmeDcfmInd->u8Type=NET_MHME_DCFM_JOIN;
0189            memcpy(&psMhmeDcfmInd->uParam.sCfmJoin,&sMhmeCfmJoin,sizeof(NET_MhmeCfmJoin_s));
0190
0191            return 1;
0192        }
0193
0194        bJoinNetworkStatus = TRUE;
0195
0196        //Updates mesh info
0197        sMeshInfo.u16NetworkAddress=psMlmeDcfmInd->uParam.sDcfmAssociate.u16AssocShortAddr;
0198        sMeshInfo.u8AcceptMeshDevice=TRUE;
0199
0200        //Fills the structure
0201        sMhmeCfmJoin.u16NetworkAddress = psMlmeDcfmInd->uParam.sDcfmAssociate.u16AssocShortAddr;
0202        sMhmeCfmJoin.u16PanId = sMeshInfo.u16PanId;
0203        sMhmeCfmJoin.u8ChannelPage = 1;
0204        sMhmeCfmJoin.u8ActiveChannel = CHANNEL;
0205        sMhmeCfmJoin.u8Status=(sMhmeCfmJoin.u16NetworkAddress==0xfffe)? DCFM_JOIN:SUCCESS;
0206
0207        #ifdef DEBUG
0208        vPrintf("Joined Network with success!(%s: %d)\n",__FILE__,__LINE__);
0209        vPrintf("With short address: %d\n",sMhmeCfmJoin.u16NetworkAddress);
0210        #endif
0211
0212        //Start counting for the children number report frame
0213        if(sMeshData.i8ChilderReportTimer==-1)
0214        {
0215            sMeshData.i8ChilderReportTimer=VirtualTimer_i8New(NET_MESH_CHILDREN_REPORT_TIMER*10); //10 *
(10 * 100)ms
0216            VirtualTimer_bReset(sMeshData.i8ChilderReportTimer);
0217            VirtualTimer_bCount(sMeshData.i8ChilderReportTimer);
0218        }
0219
0220        //Start Address Assignment Timer
0221        if(sMeshData.i8AddressAssignmentTimer==-1)
0222        {
0223            sMeshData.i8AddressAssignmentTimer=VirtualTimer_i8New(NET_MESH_ADDRESS_ASSIGNMNET_TIMER*10);
//x * (10 * 100)ms
0224            VirtualTimer_bReset(sMeshData.i8AddressAssignmentTimer);
0225            VirtualTimer_bCount(sMeshData.i8AddressAssignmentTimer);
0226        }
0227
0228        //Proceed to address assignment stage
0229        u8MhmeFlag=MHME_ADDRESS_ASSIGNMENT;
0230    }
0231    else
0232    {
0233        uint8           i;
0234        #ifdef DEBUG
0235        vPrintf("Could not join device! Error was %x (%s: %d)\n",psMlmeDcfmInd-
>uParam.sDcfmAssociate.u8Status,__FILE__,__LINE__);
0236        #endif
0237
0238        //Taking the parent device off the neigbor list
0239        for(i=0;i<MAX_NEIGHBORS;i++)
0240        {
0241            if(sMeshInfo.psNeighborList[i].u8Relationship==PARENT)
0242            {
0243                sMeshInfo.psNeighborList[i].u8Relationship=NO_RELATIONSHIP;
0244                break;
0245            }
0246        }
0247        sMeshInfo.u8AcceptMeshDevice=FALSE;
0248
```

```
0249         //Fills the structure
0250         sMhmeCfmJoin.u8Status=psMlmeDcfmInd->uParam.sDcfmAssociate.u8Status;
0251     }
0252
0253     //Reporting to the upper layer
0254     psMhmeDcfmInd->u8Type=NET_MHME_DCFM_JOIN;
0255     memcpy(&psMhmeDcfmInd->uParam.sCfmJoin,&sMhmeCfmJoin,sizeof(NET_MhmeCfmJoin_s));
0256
0257     return 1;
0258 }
0259
0260
0261
0262
0263 inline uint16 u16HdlMhmeIndJoin(NET_MhmeDcfmInd_s* psMhmeDcfmInd, MAC_MlmeDcfmInd_s* psMlmeDcfmInd)
0264 {
0265     NET_MhmeIndJoin_s   sMhmeIndJoin;
0266     uint8               u8NbNumber;
0267     bool                bSuccess=FALSE;
0268
0269
0270
0271  //checks if device can accept child
0272     if(sMeshInfo.u8NbOfChildren<=MAX_NEIGHBORS-1)
0273     {
0274         for(u8NbNumber=0;u8NbNumber<MAX_NEIGHBORS;u8NbNumber++)
0275             if( SAME64ADDR(sMeshInfo.psNeighborList[u8NbNumber].sExt,psMlmeDcfmInd-
>uParam.sIndAssociate.sDeviceAddr)
0276                 && sMeshInfo.psNeighborList[u8NbNumber].u8Relationship!=NO_RELATIONSHIP)
0277                 {
0278                     #ifdef DEBUG
0279                     vPrintf("Rejoin! Device already present in Nb List!\n");
0280                     #endif
0281
0282                     //Update info on neighbor list
0283                     if(sMeshInfo.psNeighborList[u8NbNumber].i8RejoinTimer!=-1)
0284                     {
0285                         VirtualTimer_bDelete(sMeshInfo.psNeighborList[u8NbNumber].i8RejoinTimer);
0286                         sMeshInfo.psNeighborList[u8NbNumber].i8RejoinTimer=-1;
0287                     }
0288
0289                     sMeshInfo.psNeighborList[u8NbNumber].u8Relationship=CHILD;
0290
0291                     //treats it as a rejoin request
0292                     sMhmeIndJoin.u8RejoinNetwork = TRUE;
0293                     bSuccess=TRUE;
0294                     break;
0295                 }
0296
0297         if(!bSuccess)
0298         {
0299             for(u8NbNumber=0;u8NbNumber<MAX_NEIGHBORS;u8NbNumber++)
0300                 if(sMeshInfo.psNeighborList[u8NbNumber].u8Relationship==NO_RELATIONSHIP)
0301                 {
0302                         #ifdef DEBUG
0303                         vPrintf("New Children I had %d!\n",sMeshInfo.u8NbOfChildren);
0304                         #endif
0305                         //updates local variables
0306                         sMeshInfo.u8NbOfChildren++;
0307                         memcpy(&sMeshInfo.psNeighborList[u8NbNumber].sExt,&psMlmeDcfmInd-
>uParam.sIndAssociate.sDeviceAddr,sizeof(MAC_ExtAddr_s));
0308                         sMeshInfo.psNeighborList[u8NbNumber].u16BeginningAddress=0xfffe;
0309                         sMeshInfo.psNeighborList[u8NbNumber].u16EndingAddress=0xfffe;
0310                         sMeshInfo.psNeighborList[u8NbNumber].u8Relationship=CHILD;
0311
```

```
0312                        //Start counting for the children number report frame
0313                        if(sMeshData.i8ChilderReportTimer==-1)
0314                        {
0315
sMeshData.i8ChilderReportTimer=VirtualTimer_i8New(NET_MESH_CHILDREN_REPORT_TIMER*10); //10 * (10 * 100)ms
0316                            VirtualTimer_bReset(sMeshData.i8ChilderReportTimer);
0317                            VirtualTimer_bCount(sMeshData.i8ChilderReportTimer);
0318                        }
0319
0320                        //Start Address Assignment Timer
0321                        if(sMeshData.i8AddressAssignmentTimer==-1)
0322                        {
0323
sMeshData.i8AddressAssignmentTimer=VirtualTimer_i8New(NET_MESH_ADDRESS_ASSIGNMNET_TIMER*10); //x * (10 *
100)ms
0324                            VirtualTimer_bReset(sMeshData.i8AddressAssignmentTimer);
0325                            VirtualTimer_bCount(sMeshData.i8AddressAssignmentTimer);
0326                        }
0327
0328                        //treats it as a new join request
0329                        sMhmeIndJoin.u8RejoinNetwork = FALSE;
0330                        bSuccess=TRUE;
0331                        break;
0332                    }
0333                }
0334            }
0335
0336
0337            //Fills the remaining fields of the indication structure
0338            psMhmeDcfmInd->u8Type=NET_MHME_IND_JOIN;
0339            memcpy(&psMhmeDcfmInd->uParam.sIndJoin,&sMhmeIndJoin,sizeof(NET_MhmeIndJoin_s));
0340            sMhmeIndJoin.u16CapabilityInformation = psMlmeDcfmInd->uParam.sIndAssociate.u8Capability;
0341            memcpy(&sMhmeIndJoin.sExtendedAddress,&psMlmeDcfmInd-
>uParam.sIndAssociate.sDeviceAddr,sizeof(MAC_ExtAddr_s) );
0342            sMhmeIndJoin.u16NetworkAddress=(bSuccess?0xfffe:0xffff);
0343
0344            //Sends response to the indication
0345            sMlmeReqRsp.u8Type = MAC_MLME_RSP_ASSOCIATE;
0346            sMlmeReqRsp.u8ParamLength = sizeof(MAC_MlmeRspAssociate_s);
0347            sMlmeReqRsp.uParam.sRspAssociate.u8Status=(bSuccess?SUCCESS:INVALID_REQUEST);
0348            memcpy(&sMlmeReqRsp.uParam.sRspAssociate.sDeviceAddr,&psMlmeDcfmInd-
>uParam.sIndAssociate.sDeviceAddr,sizeof(MAC_ExtAddr_s));
0349            sMlmeReqRsp.uParam.sRspAssociate.u16AssocShortAddr = sMhmeIndJoin.u16NetworkAddress;
0350            sMlmeReqRsp.uParam.sRspAssociate.u8SecurityEnable = (sMhmeIndJoin.u16CapabilityInformation&0x40)>>6;
0351
0352            //Send associate response
0353            vAppApiMlmeRequest(&sMlmeReqRsp, &sMlmeSyncCfm);
0354
0355            #ifdef DEBUG
0356            vPrintf("Success: %s\n",bSuccess?"TRUE":"FALSE");
0357            vPrintf("%s.Associate Response sent!(%s:%d)\n\t\tNumber of children: %d
\n",__FUNCTION__,__FILE__,__LINE__,sMeshInfo.u8NbOfChildren);
0358            vPrintf("\t\tSrc Addr: %x
%x\n",sMlmeReqRsp.uParam.sRspAssociate.sDeviceAddr.u32H,sMlmeReqRsp.uParam.sRspAssociate.sDeviceAddr.u32L);
0359            #endif
0360
0361            u8MhmeFlag=MHME_ADDRESS_ASSIGNMENT;
0362            return 1;
0363 }
0364
0365
0366
0367
0368
0369
```

```
0370 inline uint16 u16HdlMhmeIndCommStatus(NET_MhmeDcfmInd_s* psMhmeDcfmInd,MAC_MlmeDcfmInd_s* psMlmeDcfmInd)
0371 {
0372     #ifdef DEBUG
0373     vPrintf("Received Confirm from association\n",__FILE__,__LINE__);
0374     #endif
0375
0376     if(psMlmeDcfmInd->u8Type == MAC_MLME_IND_COMM_STATUS)
0377     {
0378         switch(psMlmeDcfmInd->uParam.sIndCommStatus.u8Status)
0379         {
0380             case MAC_ENUM_SUCCESS:
0381             {
0382                 #ifdef DEBUG
0383                 vPrintf("Association succeeded\n");
0384                 #endif
0385             }break;
0386
0387             default:
0388             {
0389                 uint8 u8NbNumber;
0390
0391                 //removes info from NbList
0392                 for(u8NbNumber=0;u8NbNumber<MAX_NEIGHBORS;u8NbNumber++)
0393                 {
0394                     if(SAME64ADDR(sMeshInfo.psNeighborList[u8NbNumber].sExt,psMlmeDcfmInd-
>uParam.sIndCommStatus.sSrcAddr.uAddr.sExt))
0395                     {
0396                         sMeshInfo.u8NbOfChildren--;
0397                         sMeshInfo.psNeighborList[u8NbNumber].u16EndingAddress=0;
0398                         sMeshInfo.psNeighborList[u8NbNumber].u8Relationship=NO_RELATIONSHIP;
0399                         memset(&sMeshInfo.psNeighborList[u8NbNumber].sExt,0,sizeof(MAC_ExtAddr_s));
0400                         break;
0401                     }
0402                 }
0403             }break;
0404         }
0405     }
0406
0407     //No need to inform upper layer?
0408     return 0;
0409 }
0410
0411
0412 inline uint16 u16HdlMhmeChildrenNumberReport(NET_MhmeDcfmInd_s* psMhmeDcfmInd,NET_MhmeTmerInd_s*
psMhmeTmerInd)
0413 {
0414     NET_CommSyncCfm_s   sCommSyncCfm;
0415     uint16              u16RequestedBlocks;
0416     uint16              u16ChildBlocks;
0417     uint16              u16Counter;
0418     uint8               i,j,k;
0419
0420     //Register this function on the debugger
0421     #ifdef DEBUG
0422
u8StackPushIdentifier("u16HdlMhmeChildrenNumberReport",strlen("u16HdlMhmeChildrenNumberReport"),FALSE);
0423     #endif
0443
0444     if((sMeshInfo.u8NbOfChildren==0)&&(u8MhmeFlag==
MHME_ADDRESS_ASSIGNMENT)&&(!sMeshData.bIsCoordinator))
0445     {
0446         sMeshData.u16NumberOfRequestedAddresses=sMeshInfo.u8NbOfChildren+1;
0447
0448         #ifdef DEBUG
0449         vPrintf("++++++++++++++++++++++++\n");
```

```
0450          vPrintf("+++++ADDR ASSIGNMENT+++++\n");
0451          vPrintf("++++++++++++++++++++++++\n");
0452          vPrintf("Allocating Addresses\n");
0453          vPrintf("Number of children: %d\n", sMeshInfo.u8NbOfChildren);
0454          vPrintf("Addressess Requested: %d\n",sMeshData.u16NumberOfRequestedAddresses);
0455          vPrintf("Allocated Addresses: %d\n",sMeshData.u16AllocatedAddresses);
0456          vPrintf("Free Blocks: %d\n",sMeshData.u16FreeBlocks);
0457          vPrintf("++++++++++++++++++++++++\n");
0458          #endif
0459
0460          //Send child report
0461          vNetApiCommRequest(FRAME_CHILDREN_NUMBER_REPORT,&sCommSyncCfm);
0462      }
0463
0464      //Attend to the update requests - (warning, updates might become news in this stage)
0465      for(i=0;i<MAX_NEIGHBORS;i++)
0466      {
0467          if(sMeshInfo.psNeighborList[i].u8Relationship==CHILD)
0468          {
0469              if(READBITMAP(sMeshData.sBitMapUpdate,i))
0470              {
0471                  //Check if the new requested addresses are bigger than the already assigned blocks
0472                  if(sMeshInfo.psNeighborList[i].u16EndingAddress-
sMeshInfo.psNeighborList[i].u16BeginningAddress<sMeshInfo.psNeighborList[i].u16RequestedAddresses)
0473                  {
0474                      //Dealocate the addresses and assign the child to the new list
0475                      u16ChildBlocks= (sMeshInfo.psNeighborList[i].u16EndingAddress-
sMeshInfo.psNeighborList[i].u16BeginningAddress)/sMeshData.u16BlockSize
0476                                      + ((sMeshInfo.psNeighborList[i].u16EndingAddress-
sMeshInfo.psNeighborList[i].u16BeginningAddress)%sMeshData.u16BlockSize)!=0?1:0;
0477                      sMeshData.u16AllocatedAddresses-=sMeshInfo.psNeighborList[i].u16EndingAddress-
sMeshInfo.psNeighborList[i].u16BeginningAddress;
0478                      sMeshData.u16FreeBlocks+=u16ChildBlocks;
0479                      sMeshInfo.psNeighborList[i].u16BeginningAddress=0xfffe;
0480                      sMeshInfo.psNeighborList[i].u16EndingAddress=0xfffe;
0481                      PLACEBITMAP(sMeshData.sBitMapNew,i);
0482                  }
0483
0484
0485                  #ifdef DEBUG
0486                  vPrintf("++++++++++++++++++++++++\n");
0487                  vPrintf("+++++UPDATE REQUEST++++++\n");
0488                  vPrintf("++++++++++++++++++++++++\n");
0489                  vPrintf("Allocating Addresses\n");
0490                  vPrintf("Number of children: %d\n", sMeshInfo.u8NbOfChildren);
0491                  vPrintf("Addressess Requested: %d\n",sMeshData.u16NumberOfRequestedAddresses);
0492                  vPrintf("Allocated Addresses: %d\n",sMeshData.u16AllocatedAddresses);
0493                  vPrintf("Free Blcoks: %d\n",sMeshData.u16FreeBlocks);
0494                  vPrintf("++++++++++++++++++++++++\n");
0495                  #endif
0496
0497                  //Unmark this neighbor from the update list
0498                  UNPLACEBITMAP(sMeshData.sBitMapUpdate,i);
0499              }
0500          }
0501      }
0502
0503      //Attend to the rejoin requests
0504      for(i=0;i<MAX_NEIGHBORS;i++)
0505      {
0506          if(sMeshInfo.psNeighborList[i].u8Relationship==CHILD)
0507          {
0508              if(READBITMAP(sMeshData.sBitMapRejoin,i))
0509              {
0510                  //Mark the child for the address assignmnet
```

```
0511                PLACEBITMAP(sMeshData.sBitMapAddressAssignment,i);
0512
0513                //Unmark this neighbor from the update list
0514                UNPLACEBITMAP(sMeshData.sBitMapRejoin,i);
0515            }
0516        }
0517    }
0518
0519    //Attend to the new requests - (warning, attended updates and rejoins might become news in this
stage)
0520    for(i=0;i<MAX_NEIGHBORS;i++)
0521    {
0522        if(sMeshInfo.psNeighborList[i].u8Relationship==CHILD)
0523        {
0524            if(READBITMAP(sMeshData.sBitMapNew,i))
0525            {
0526                //Find out if the child isnt allready leaving
0527                if(READBITMAP(sMeshData.sBitMapNoNew,i))
0528                {
0529                    UNPLACEBITMAP(sMeshData.sBitMapNew,i);
0530
0531                    //Proceed to next child
0532                    continue;
0533                }
0534
0535                //Find out if the request fits into the free addresses
0536                if((sMeshData.u16EndingAddress-sMeshInfo.u16NetworkAddress-1)-
sMeshData.u16AllocatedAddresses<sMeshInfo.psNeighborList[i].u16RequestedAddresses)
0537                {
0538                    //Send child report
0539
sMeshData.u16NumberOfRequestedAddresses+=(sMeshInfo.psNeighborList[i].u16RequestedAddresses-
sMeshData.u16AllocatedAddresses);
0540                    vNetApiCommRequest(FRAME_CHILDREN_NUMBER_REPORT,&sCommSyncCfm);
0541                    UNPLACEBITMAP(sMeshData.sBitMapNew,i);
0542                    PLACEBITMAP(sMeshData.sBitMapWaitingReport,i);
0543
0544                    //Proceed to next child
0545                    continue;
0546                }
0547
0548                //Understanding how many blocks the child need
0549
u16RequestedBlocks=sMeshInfo.psNeighborList[i].u16RequestedAddresses/sMeshData.u16BlockSize+
0550                                    +
(sMeshInfo.psNeighborList[i].u16RequestedAddresses%sMeshData.u16BlockSize)!=0?1:0;
0551
0552
0553                #ifdef DEBUG
0554                vPrintf("++++++++++++++++++++++++\n");
0555                vPrintf("+++++BLOCK REQUEST+++++++\n");
0556                vPrintf("++++++++++++++++++++++++\n");
0557                vPrintf("Nb of Addresses Requested:
%d\n",sMeshInfo.psNeighborList[i].u16RequestedAddresses);
0558                vPrintf("Requested Blocks: %d\n", u16RequestedBlocks);
0559                vPrintf("Block Size: %d\n",sMeshData.u16BlockSize);
0560                vPrintf("++++++++++++++++++++++++");
0561                #endif
0562
0563
0564                //If there are not enough free blocks to attend to the request
0565                if(sMeshData.u16FreeBlocks<u16RequestedBlocks)
0566                {
0567                    #ifdef DEBUG
0568                    vPrintf("++++++++++++++++++++++++\n");
```

```
0569                    vPrintf("++++NEED MORE BLOCKS I+++\n");
0570                    vPrintf("++++++++++++++++++++++++++\n");
0571                    vPrintf("Requested Blocks: %d\n", u16RequestedBlocks);
0572                    vPrintf("Free Blocks: %d\n",sMeshData.u16FreeBlocks);
0573                    for(k=0;k<MAX_NEIGHBORS+1;k++)
0574                    {
0575                        vPrintf("FBS[%d] Beginning Block:
%d\n",k,sMeshData.sFreeBlock[k].u16BeginningBlock);
0576                        vPrintf("FBS Number: %d\n",sMeshData.sFreeBlock[k].u16NumberOfBlocks);
0577                    }
0578                    vPrintf("++++++++++++++++++++++++++");
0579                    #endif
0580
0581
0582                    //Duplicate the number of blocks
0583                    if((sMeshData.u16BlockSize/=2)==0) sMeshData.u16BlockSize=1;
0584                    sMeshData.u16FreeBlocks=(sMeshData.u16EndingAddress-sMeshInfo.u16NetworkAddress-
1)/sMeshData.u16BlockSize;
0585                    sMeshData.u16AllocatedAddresses=0;
0586                    sMeshData.u8LeftAddr=sMeshData.u16BlockSize%MAX_NEIGHBORS;
0587
0588                    //Marking all the childs for updating
0589                    /** ATTENTION **/
0590                    memset(&sMeshData.sBitMapNew,0xff,sizeof(NET_BitMap_s));
0591
0592                    //Updating the free block list
0593                    sMeshData.sFreeBlock[0].u16BeginningBlock=0;
0594                    sMeshData.sFreeBlock[0].u16NumberOfBlocks=sMeshData.u16FreeBlocks;
0595                    sMeshData.u16NumberOfFreeBlocksInList=1;
0596
0597
0598                    #ifdef DEBUG
0599                    vPrintf("++++++++++++++++++++++++++\n");
0600                    vPrintf("+++NEED MORE BLOCKS II+++\n");
0601                    vPrintf("++++++++++++++++++++++++++\n");
0602                    vPrintf("Requested Blocks: %d\n", u16RequestedBlocks);
0603                    vPrintf("Block Size: %d\n",sMeshData.u16BlockSize);
0604                    for(k=0;k<MAX_NEIGHBORS+1;k++)
0605                    {
0606                        vPrintf("FBS[%d] Beginning Block:
%d\n",k,sMeshData.sFreeBlock[k].u16BeginningBlock);
0607                        vPrintf("FBS Number: %d\n",sMeshData.sFreeBlock[k].u16NumberOfBlocks);
0608                    }
0609                    vPrintf("++++++++++++++++++++++++++\n");
0610                    #endif
0611
0612
0613                    //Start the new operation all over again
0614                    i=-1;
0615                    continue;
0616                }
0617
0618                //Performing address assignment procedure
0619
0620                //Creating the super block
0621                for(j=0,u16Counter=0;j<MAX_NEIGHBORS+1;j++)
0622                {
0623                    u16Counter+=sMeshData.sFreeBlock[j].u16NumberOfBlocks;
0624                    if(u16Counter>=u16RequestedBlocks) break;
0625                }
0626
0627                //Find out the children that are affected by this operation
0628                for(k=0;k<MAX_NEIGHBORS;k++)
0629                    if(sMeshInfo.psNeighborList[k].u8Relationship==CHILD)
```

```
0630                        if(
(sMeshInfo.psNeighborList[k].u16BeginningAddress<=(sMeshData.sFreeBlock[j].u16BeginningBlock+sMeshData.sFreeB
lock[j].u16NumberOfBlocks)*sMeshData.u16BlockSize)&&
0631
(sMeshInfo.psNeighborList[k].u16EndingAddress>=sMeshData.sFreeBlock[0].u16BeginningBlock*sMeshData.u16BlockSi
ze))
0632                        {
0633                            //Dealocate the addresses and assign the child to the new list
0634                            u16ChildBlocks= (sMeshInfo.psNeighborList[k].u16EndingAddress-
sMeshInfo.psNeighborList[k].u16BeginningAddress)/sMeshData.u16BlockSize
0635                                          + ((sMeshInfo.psNeighborList[k].u16EndingAddress-
sMeshInfo.psNeighborList[k].u16BeginningAddress)%sMeshData.u16BlockSize)!=0?1:0;
0636                            sMeshData.u16AllocatedAddresses-
=sMeshInfo.psNeighborList[k].u16EndingAddress-sMeshInfo.psNeighborList[k].u16BeginningAddress;
0637                            sMeshData.u16FreeBlocks+=u16ChildBlocks;
0638                            u16Counter+=u16ChildBlocks;
0639                            sMeshInfo.psNeighborList[k].u16BeginningAddress=0xfffe;
0640                            sMeshInfo.psNeighborList[k].u16EndingAddress=0xfffe;
0641                            PLACEBITMAP(sMeshData.sBitMapNew,k);
0642                        }
0643
0644                //Allocate the addresses corresponding to the first free super block
0645
sMeshInfo.psNeighborList[i].u16BeginningAddress=sMeshInfo.u16NetworkAddress+1+sMeshData.sFreeBlock[0].u16Begi
nningBlock*sMeshData.u16BlockSize;
0646
sMeshInfo.psNeighborList[i].u16EndingAddress=sMeshInfo.psNeighborList[i].u16BeginningAddress+u16RequestedBloc
ks*sMeshData.u16BlockSize;
0647                sMeshData.u16AllocatedAddresses+=u16RequestedBlocks*sMeshData.u16BlockSize;
0648                sMeshData.u16FreeBlocks-=u16RequestedBlocks;
0649
0650                //Updating the free block list
0651                sMeshData.sFreeBlock[0].u16BeginningBlock+=u16RequestedBlocks;
0652                sMeshData.sFreeBlock[0].u16NumberOfBlocks=u16Counter-u16RequestedBlocks;
0653
memcpy(&sMeshData.sFreeBlock[1],&sMeshData.sFreeBlock[j+1],sizeof(NET_FreeBlock_s)*(MAX_NEIGHBORS-j));
0654                sMeshData.u16NumberOfFreeBlocksInList-=j;
0655
0656                //Mark the child for the address assignmnet
0657                UNPLACEBITMAP(sMeshData.sBitMapNew,i);
0658                PLACEBITMAP(sMeshData.sBitMapAddressAssignment,i);
0659            }
0660        }
0661    }
0662
0663
0664    #ifdef DEBUG
0665    vPrintf("++++++++++++++++++++++++++\n");
0666    vPrintf("++++++++++LEAVING++++++++\n");
0667    vPrintf("++++++++++++++++++++++++++\n");
0668    vPrintf("Number of children: %d\n", sMeshInfo.u8NbOfChildren);
0669    vPrintf("Addressess Requested: %d\n",sMeshData.u16NumberOfRequestedAddresses);
0670    vPrintf("Allocated Addresses: %d\n",sMeshData.u16AllocatedAddresses);
0671    vPrintf("Free Blcoks: %d\n",sMeshData.u16FreeBlocks);
0672    vPrintf("++++++++++++++++++++++++++\n");
0673    #endif
0674
0675    //Reset the timer for the next child number report checking
0676    VirtualTimer_bReset(sMeshData.i8ChilderReportTimer);
0677    VirtualTimer_bCount(sMeshData.i8ChilderReportTimer);
0678
0679    //Deregister this function off the debugger
0680    #ifdef DEBUG
0681    vStackPopIdentifier();
0682    #endif
```

```
0683
0684     return 0;
0685 }
0686
0687 inline uint16 u16HdlMhmeAddressAssignment(NET_MhmeDcfmInd_s* psMhmeDcfmInd,NET_MhmeTmerInd_s*
psMhmeTmerInd)
0688 {
0689     NET_CommSyncCfm_s   sCommSyncCfm;
0690
0691     //Register this function on the debugger
0692     #ifdef DEBUG
0693
u8StackPushIdentifier("u16HdlMhmeChildrenNumberReport",strlen("u16HdlMhmeChildrenNumberReport"),FALSE);
0694     #endif
0695
0696     //Attend to the address assignment requests
0697     vNetApiCommRequest(FRAME_ADDRESS_ASSIGNMENT,&sCommSyncCfm);
0698
0699     //Moving to the topology discovery phase
0700     if(u8MhmeFlag==MHME_ADDRESS_ASSIGNMENT)
0701         u8MhmeFlag=MHME_TOPOLOGY_DISCOVERY;
0702
0703     //Start hello timer
0704     if(sMeshData.i8HelloTimer==-1)
0705     {
0706         sMeshData.i8HelloTimer=VirtualTimer_i8New(NET_MESH_HELLO_TIMER*10);
0707         VirtualTimer_bReset(sMeshData.i8HelloTimer);
0708         VirtualTimer_bCount(sMeshData.i8HelloTimer);
0709     }
0710
0711     VirtualTimer_bReset(sMeshData.i8AddressAssignmentTimer);
0712     VirtualTimer_bCount(sMeshData.i8AddressAssignmentTimer);
0713
0714     #ifdef DEBUG
0715     vPrintf("Hello Timer=%d\n",sMeshData.i8HelloTimer);
0716     #endif
0717
0718     //Deregister this function off the debugger
0719     #ifdef DEBUG
0720     vStackPopIdentifier();
0721     #endif
0722
0723     return 0;
0724 }
0725
0726
0727 inline uint16 u16HdlMhmeHello(NET_MhmeDcfmInd_s* psMhmeDcfmInd,NET_MhmeTmerInd_s* psMhmeTmerInd)
0728 {
0729     NET_CommSyncCfm_s   sCommSyncCfm;
0730
0731     //Register this function on the debugger
0732     #ifdef DEBUG
0733     u8StackPushIdentifier("u16HdlMhmeHello",strlen("u16HdlMhmeHello"),FALSE);
0734     #endif
0735
0736     //Send the hello command frame
0737     vNetApiCommRequest(FRAME_HELLO,&sCommSyncCfm);
0738
0739     //Reset the timer for the next hello frame
0740     VirtualTimer_bReset(sMeshData.i8HelloTimer);
0741     VirtualTimer_bCount(sMeshData.i8HelloTimer);
0742
0743     //Deregister this function off the debugger
0744     #ifdef DEBUG
0745     vStackPopIdentifier();
```

```
0746     #endif
0747
0748     return 0;
0749 }
0750
0751
0752 inline uint16 u16HdlMhmeAppTimerExpired(NET_MhmeDcfmInd_s* psMhmeDcfmInd,NET_MhmeTmerInd_s*
psMhmeTmerInd)
0753 {
0754     NET_MhmeIndTmer_s sMhmeIndTmer;
0755
0756     //Register this function on the debugger
0757     #ifdef DEBUG
0758     u8StackPushIdentifier("u16HdlMhmeAppTimerExpired",strlen("u16HdlMhmeAppTimerExpired"),FALSE);
0759     #endif
0760
0761     //Sends timer expired
0762     sMhmeIndTmer.u8TriggeredTimer=psMhmeTmerInd->u8TriggeredTimer;
0763     psMhmeDcfmInd->u8Type=NET_MHME_IND_TIMER;
0764     memcpy(&psMhmeDcfmInd->uParam.sIndTmer,&sMhmeIndTmer,sizeof(NET_MhmeIndTmer_s));
0765
0766     //Deregister this function off the debugger
0767     #ifdef DEBUG
0768     vStackPopIdentifier();
0769     #endif
0770
0771     return 1;
0772 }
0773
0774
0775
0776 inline uint16 u16HdlMhmeCfmChildrenNumberReport(NET_MhmeDcfmInd_s* psMhmeDcfmInd,NET_MeshDcfmInd_s*
psMeshDcfmInd);
0777 inline uint16 u16HdlMhmeCfmAddressAssignment(NET_MhmeDcfmInd_s* psMhmeDcfmInd,NET_MeshDcfmInd_s*
psMeshDcfmInd);
0778 inline uint16 u16HdlMhmeCfmHello(NET_MhmeDcfmInd_s* psMhmeDcfmInd,NET_MeshDcfmInd_s* psMeshDcfmInd);
0779 inline uint16 u16HdlMhmeCfmLeave(NET_MhmeDcfmInd_s* psMhmeDcfmInd,NET_MeshDcfmInd_s* psMeshDcfmInd);
0780
0781 inline uint16 u16HdlMhmeCfmData(NET_MhmeDcfmInd_s* psMhmeDcfmInd,NET_MeshDcfmInd_s* psMeshDcfmInd)
0782 {
0783     uint8               u8Handle;
0784     uint16              u16CommFrame;
0785     uint16              u16ReturnValue=0;
0786
0787     //Register this function on the debugger
0788     #ifdef DEBUG
0789     u8StackPushIdentifier("u16HdlMhmeCfmData",strlen("u16HdlMhmeCfmData"),FALSE);
0790     #endif
0791
0792     //Getting msdu handle
0793     u8Handle=psMeshDcfmInd->uParam.sMeshCfmData.u8MhsduHandle;
0794
0795     #ifdef DEBUG
0796     vPrintf("HANDLE IS %d\n",u8Handle);
0797     #endif
0798
0799     //Validating msdu handle
0800     if(bMhmeValidateMhsduHandle(u8Handle))
0801     {
0802         //Unmapping the command sender handle to a command frame
0803         u16CommFrame=sMhmeMhsduHandleQueue.asMhsduHandle[u8Handle].u8Type;
0804
0805
0806         //Switching the command frames
0807         switch(u16CommFrame)
```

```
0808         {
0809             case FRAME_CHILDREN_NUMBER_REPORT:
0810             {
0811                 #ifdef DEBUG
0812                 vStackPrintf(__FILE__,__LINE__,"Received FRAME_CHILDREN_NUMBER_REPORT.");
0813                 #endif
0814 
0815                 u16ReturnValue=u16HdlMhmeCfmChildrenNumberReport(psMhmeDcfmInd,psMeshDcfmInd);
0816             }break;
0817             case FRAME_ADDRESS_ASSIGNMENT:
0818             {
0819                 #ifdef DEBUG
0820                 vStackPrintf(__FILE__,__LINE__,"Received FRAME_ADDRESS_ASSIGNMENT.");
0821                 #endif
0822 
0823                 u16ReturnValue=u16HdlMhmeCfmAddressAssignment(psMhmeDcfmInd,psMeshDcfmInd);
0824             }break;
0825             case FRAME_HELLO:
0826             {
0827                 #ifdef DEBUG
0828                 vStackPrintf(__FILE__,__LINE__,"Received FRAME_HELLO.");
0829                 #endif
0830 
0831                  u16ReturnValue=u16HdlMhmeCfmHello(psMhmeDcfmInd,psMeshDcfmInd);
0832             }break;
0833             case FRAME_LEAVE:
0834             {
0835                 #ifdef DEBUG
0836                 vStackPrintf(__FILE__,__LINE__,"Received FRAME_LEAVE.");
0837                 #endif
0838 
0839                 u16ReturnValue=u16HdlMhmeCfmLeave(psMhmeDcfmInd,psMeshDcfmInd);
0840             }break;
0841             default:
0842             {
0843                 #ifdef DEBUG
0844                 vStackPrintf(__FILE__,__LINE__,"Unhandeled comfirm data.");
0845                 #endif
0846             }
0847         }
0848 
0849         //Erase the handle from the list
0850         vMhmeEraseMhsduHandle(u8Handle);
0851     }
0852     else
0853     {
0854         #ifdef DEBUG
0855         vStackPrintf(__FILE__,__LINE__,"Invalid Handle Detected.");
0856         #endif
0857     }
0858 
0859     //Deregister this function off the debugger
0860     #ifdef DEBUG
0861     vStackPopIdentifier();
0862     #endif
0863 
0864     return u16ReturnValue;
0865 }
0866 
0867 uint16 u16HdlMhmeCfmChildrenNumberReport(NET_MhmeDcfmInd_s* psMhmeDcfmInd,NET_MeshDcfmInd_s*
psMeshDcfmInd)
0868 {
0869     uint16              u16ReturnValue=0;
0870 
0871     //Register this function on the debugger
```

```
0872     #ifdef DEBUG
0873
u8StackPushIdentifier("u16HdlMhmeCfmChildrenNumberReport",strlen("u16HdlMhmeCfmChildrenNumberReport"),FALSE);
0874     #endif
0875
0876     //Deregister this function off the debugger
0877     #ifdef DEBUG
0878     vStackPopIdentifier();
0879     #endif
0880
0881     return u16ReturnValue;
0882 }
0883
0884 inline uint16 u16HdlMhmeCfmAddressAssignment(NET_MhmeDcfmInd_s* psMhmeDcfmInd,NET_MeshDcfmInd_s*
psMeshDcfmInd)
0885 {
0886     uint16              u16ReturnValue=0;
0887
0888     //Register this function on the debugger
0889     #ifdef DEBUG
0890
u8StackPushIdentifier("u16HdlMhmeCfmAddressAssignment",strlen("u16HdlMhmeCfmAddressAssignment"),FALSE);
0891     #endif
0892
0893     //Deregister this function off the debugger
0894     #ifdef DEBUG
0895     vStackPopIdentifier();
0896     #endif
0897
0898     return u16ReturnValue;
0899 }
0900
0901 inline uint16 u16HdlMhmeCfmHello(NET_MhmeDcfmInd_s* psMhmeDcfmInd,NET_MeshDcfmInd_s* psMeshDcfmInd)
0902 {
0903     uint16              u16ReturnValue=0;
0904
0905     //Register this function on the debugger
0906     #ifdef DEBUG
0907     u8StackPushIdentifier("u16HdlMhmeCfmHello",strlen("u16HdlMhmeCfmHello"),FALSE);
0908     #endif
0909
0910     //Deregister this function off the debugger
0911     #ifdef DEBUG
0912     vStackPopIdentifier();
0913     #endif
0914
0915     return u16ReturnValue;
0916 }
0917
0918 inline uint16 u16HdlMhmeCfmLeave(NET_MhmeDcfmInd_s* psMhmeDcfmInd,NET_MeshDcfmInd_s* psMeshDcfmInd)
0919 {
0920     uint16              u16ReturnValue=0;
0921     uint8               u8Handle;
0922     uint8               i;
0923
0924     //Register this function on the debugger
0925     #ifdef DEBUG
0926     u8StackPushIdentifier("u16HdlMhmeCfmLeave",strlen("u16HdlMhmeCfmLeave"),FALSE);
0927     #endif
0928
0929     //Identifying the msdu handle
0930     u8Handle=psMeshDcfmInd->uParam.sMeshCfmData.u8MhsduHandle;
0931
0932     //Finding the corresponding neighbor
0933     for(i=0;i<MAX_NEIGHBORS;i++)
```

```
0934         if(sMeshInfo.psNeighborList[i].i8MsduHandleLeave==u8Handle)
0935         {
0936             //Renewing the handle
0937             sMeshInfo.psNeighborList[i].i8MsduHandleLeave=-1;
0938 
0939             //Was the packet well sent?
0940             if(psMeshDcfmInd->uParam.sMeshCfmData.eStatus!=NET_ENUM_SUCCESS)
0941                 sMeshInfo.psNeighborList[i].u8NbOfLeaveRetries--;
0942 
0943             //A success has ocurred or a device took to long to answer
0944             if((sMeshInfo.psNeighborList[i].u8NbOfLeaveRetries==0)||(psMeshDcfmInd-
>uParam.sMeshCfmData.eStatus==NET_ENUM_SUCCESS))
0945             {
0946                 //Remove the device form the leaving procedure
0947                 #ifdef DEBUG
0948                 WRITEBITMAP(sMeshData.sBitMapLeave);
0949                 #endif
0950 
0951                 UNPLACEBITMAP(sMeshData.sBitMapLeave,i);
0952 
0953                 #ifdef DEBUG
0954                 WRITEBITMAP(sMeshData.sBitMapLeave);
0955                 #endif
0956 
0957                 sMeshData.u8NbOfLeavingCommands--;
0958             }
0959             break;
0960         }
0961 
0962     //If there are no more leave commands to manage, remove the leave timer
0963     if((sMeshData.u8NbOfLeavingCommands==0)&&(sMeshData.i8LeaveTimer!=-1))
0964     {
0965         VirtualTimer_bStop(sMeshData.i8LeaveTimer);
0966         VirtualTimer_bDelete(sMeshData.i8LeaveTimer);
0967         sMeshData.i8LeaveTimer=-1;
0968     }
0969 
0970     //Sending a defered confrim to the higher layer
0971     if(u8MhmeFlag==MHME_LEAVING)
0972     {
0973         if(sMeshData.u8NbOfLeavingCommands==0)
0974         {
0975             NET_MhmeReqLeave_s sMhmeReqLeave;
0976             NET_MhmeCfmLeave_s sMhmeCfmLeave;
0977 
0978             //We can proceed with a normal leave process
0979             sMhmeReqLeave.u8RemoveSelf=TRUE;
0980             sMhmeReqLeave.u8RemoveChildren=FALSE;
0981             vHdlMhmeReqLeave(&sMhmeReqLeave,&sMhmeCfmLeave);
0982             psMhmeDcfmInd->u8Type=NET_MHME_DCFM_LEAVE;
0983             psMhmeDcfmInd->uParam.sCfmLeave.u8Status=sMhmeCfmLeave.u8Status;
0984             memcpy(&psMhmeDcfmInd-
>uParam.sCfmLeave.sDeviceAddress,&sMhmeCfmLeave.sDeviceAddress,sizeof(MAC_ExtAddr_s));
0985             u16ReturnValue=1;
0986         }
0987     }
0988     //If the leaving device is another device (not this device)
0989     else
0990     {
0991         psMhmeDcfmInd->u8Type=NET_MHME_DCFM_LEAVE;
0992         psMhmeDcfmInd->uParam.sCfmLeave.u8Status=SUCCESS;
0993         memcpy(&psMhmeDcfmInd-
>uParam.sCfmLeave.sDeviceAddress,&sMeshInfo.psNeighborList[i].sExt,sizeof(MAC_ExtAddr_s));
0994         u16ReturnValue=1;
0995     }
```

```
0996
0997     //Deregister this function off the debugger
0998     #ifdef DEBUG
0999     vStackPopIdentifier();
1000     #endif
1001
1002     return u16ReturnValue;
1003 }
1004
1005
1006
1007 inline uint16 u16HdlMhmeIndChildrenNumberReport(NET_MhmeDcfmInd_s* psMhmeDcfmInd);
1008 inline uint16 u16HdlMhmeIndAddressAssignment(NET_MhmeDcfmInd_s* psMhmeDcfmInd);
1009 inline uint16 u16HdlMhmeIndHello(NET_MhmeDcfmInd_s* psMhmeDcfmInd);
1010 inline uint16 u16HdlMhmeIndLeave(NET_MhmeDcfmInd_s* psMhmeDcfmInd,NET_CommDcfmInd_s* psCommDcfmInd);
1011
1012 inline uint16 u16HdlMhmeIndData(NET_MhmeDcfmInd_s* psMhmeDcfmInd,NET_MeshDcfmInd_s* psMeshDcfmInd)
1013 {
1014     NET_CommDcfmInd_s   sCommDcfmInd;
1015     uint16              u16ReturnValue=0;
1016
1017     //Register this function on the debugger
1018     #ifdef DEBUG
1019     u8StackPushIdentifier("u16HdlMhmeIndData",strlen("u16HdlMhmeIndData"),FALSE);
1020     #endif
1021
1022     //Processes the received command frame
1023     vNetApiCommTranslate(&sCommDcfmInd,psMeshDcfmInd);
1024
1025     switch(sCommDcfmInd.u8Type)
1026     {
1027         case FRAME_CHILDREN_NUMBER_REPORT:
1028         {
1029             #ifdef DEBUG
1030             vStackPrintf(__FILE__,__LINE__,"Received FRAME_CHILDREN_NUMBER_REPORT.");
1031             #endif
1032
1033             u16ReturnValue=u16HdlMhmeIndChildrenNumberReport(psMhmeDcfmInd);
1034         }break;
1035         case FRAME_ADDRESS_ASSIGNMENT:
1036         {
1037             #ifdef DEBUG
1038             vStackPrintf(__FILE__,__LINE__,"Received FRAME_ADDRESS_ASSIGNMENT.");
1039             #endif
1040
1041             u16ReturnValue=u16HdlMhmeIndAddressAssignment(psMhmeDcfmInd);
1042         }break;
1043         case FRAME_HELLO:
1044         {
1045             #ifdef DEBUG
1046             vStackPrintf(__FILE__,__LINE__,"Received FRAME_HELLO.");
1047             #endif
1048
1049             u16ReturnValue=u16HdlMhmeIndHello(psMhmeDcfmInd);
1050         }break;
1051         case FRAME_LEAVE:
1052         {
1053             #ifdef DEBUG
1054             vStackPrintf(__FILE__,__LINE__,"Received FRAME_LEAVE.");
1055             #endif
1056
1057             u16ReturnValue=u16HdlMhmeIndLeave(psMhmeDcfmInd,&sCommDcfmInd);
1058         }break;
1059         case FRAME_INVALID:
1060         {
```

```
1061         }break;
1062         default:
1063         {
1064             #ifdef DEBUG
1065             vStackPrintf(__FILE__,__LINE__,"Unhandeled indication data.");
1066             #endif
1067         }
1068     }
1069
1070     //Deregister this function off the debugger
1071     #ifdef DEBUG
1072     vStackPopIdentifier();
1073     #endif
1074
1075     return u16ReturnValue;
1076 }
1077
1078 inline uint16 u16HdlMhmeIndChildrenNumberReport(NET_MhmeDcfmInd_s* psMhmeDcfmInd)
1079 {
1080     uint16              u16ReturnValue=0;
1081
1082     //Register this function on the debugger
1083     #ifdef DEBUG
1084
u8StackPushIdentifier("u16HdlMhmeIndChildrenNumberReport",strlen("u16HdlMhmeIndChildrenNumberReport"),FALSE);
1085     #endif
1086
1087     //Deregister this function off the debugger
1088     #ifdef DEBUG
1089     vStackPopIdentifier();
1090     #endif
1091
1092     return u16ReturnValue;
1093 }
1094
1095
1096 inline uint16 u16HdlMhmeIndAddressAssignment(NET_MhmeDcfmInd_s* psMhmeDcfmInd)
1097 {
1098     uint16              u16ReturnValue=0;
1099     NET_MhmeCfmJoin_s   sMhmeCfmJoin;
1100
1101     //Register this function on the debugger
1102     #ifdef DEBUG
1103
u8StackPushIdentifier("u16HdlMhmeIndAddressAssignment",strlen("u16HdlMhmeIndAddressAssignment"),FALSE);
1104     #endif
1105
1106     if(sMeshInfo.u16NetworkAddress==0xfffe)
1107     {
1108         NET_MhmeReqLeave_s  sMhmeReqLeave;
1109         NET_MhmeCfmLeave_s  sMhmeCfmLeave;
1110
1111         //We can proceed with a normal leave process
1112         sMhmeReqLeave.u8RemoveSelf=TRUE;
1113         sMhmeReqLeave.u8RemoveChildren=TRUE;
1114         vHdlMhmeReqLeave(&sMhmeReqLeave,&sMhmeCfmLeave);
1115         psMhmeDcfmInd->u8Type=NET_MHME_IND_LEAVE;
1116         psMhmeDcfmInd->uParam.sCfmLeave.u8Status=sMhmeCfmLeave.u8Status;
1117         memcpy(&psMhmeDcfmInd-
>uParam.sCfmLeave.sDeviceAddress,&sMhmeCfmLeave.sDeviceAddress,sizeof(MAC_ExtAddr_s));
1118         u16ReturnValue=1;
1119     }
1120     else
1121     {
1122         //Sends join confirm
```

```
1123         sMhmeCfmJoin.u16NetworkAddress = sMeshInfo.u16NetworkAddress;
1124         sMhmeCfmJoin.u16PanId = sMeshInfo.u16PanId;
1125         sMhmeCfmJoin.u8ChannelPage = 1;
1126         sMhmeCfmJoin.u8ActiveChannel = CHANNEL;
1127         sMhmeCfmJoin.u8Status=SUCCESS;
1128         psMhmeDcfmInd->u8Type=NET_MHME_DCFM_JOIN;
1129         memcpy(&psMhmeDcfmInd->uParam.sCfmJoin,&sMhmeCfmJoin,sizeof(NET_MhmeCfmJoin_s));
1130         u16ReturnValue=1;
1131     }
1132
1133     //Deregister this function off the debugger
1134     #ifdef DEBUG
1135     vStackPopIdentifier();
1136     #endif
1137
1138     return u16ReturnValue;
1139 }
1140
1141 inline uint16 u16HdlMhmeIndHello(NET_MhmeDcfmInd_s* psMhmeDcfmInd)
1142 {
1143     uint16  u16ReturnValue=0;
1144
1145     //Register this function on the debugger
1146     #ifdef DEBUG
1147     u8StackPushIdentifier("u16HdlMhmeIndHello",strlen("u16HdlMhmeIndHello"),FALSE);
1148     #endif
1149
1150     //Deregister this function off the debugger
1151     #ifdef DEBUG
1152     vStackPopIdentifier();
1153     #endif
1154
1155     return u16ReturnValue;
1156 }
1157
1158 inline uint16 u16HdlMhmeIndLeave(NET_MhmeDcfmInd_s* psMhmeDcfmInd,NET_CommDcfmInd_s* psCommDcfmInd)
1159 {
1160     uint16  u16ReturnValue=0;
1161
1162     //Register this function on the debugger
1163     #ifdef DEBUG
1164     u8StackPushIdentifier("u16HdlMhmeIndLeave",strlen("u16HdlMhmeIndLeave"),FALSE);
1165     #endif
1166
1167     //Sending to the application an indiction that it has left the network
1168     psMhmeDcfmInd->u8Type=(psCommDcfmInd-
>u8Status==CFM_STATUS_SYNC_SUCCESS)?NET_MHME_IND_LEFT:NET_MHME_IND_LEAVE;
1169     memset(&psMhmeDcfmInd->uParam.sIndLeave.sDeviceAddress,0x00,sizeof(MAC_ExtAddr_s));
1170     u16ReturnValue=1;
1171
1172     //Deregister this function off the debugger
1173     #ifdef DEBUG
1174     vStackPopIdentifier();
1175     #endif
1176
1177     return u16ReturnValue;
1178 }
1179
```

XIX. MhmeControl.h

```
0001 #ifndef MHME_CONTROL
0002 #define MHME_CONTROL
0003
```

```
0004 #include "Mhme.h"
0005
0006 //To translate messages to mhme messages and to perform the process needed on those messages
0007 uint16 u16MhmeTranslateMessage(NET_MhmeDcfmInd_s* psMhmeDcfmInd,NET_MhmeEvent_s* psMhmeEvent);
0008 inline uint16 u16MhmeTranslateCommMessage(NET_MhmeDcfmInd_s* psMhmeDcfmInd,NET_MeshDcfmInd_s*
psMeshDcfmInd);
0009 inline uint16 u16MhmeTranslateMlmeMessage(NET_MhmeDcfmInd_s* psMhmeDcfmInd,MAC_MlmeDcfmInd_s*
psMlmeDcfmInd);
0010 inline uint16 u16MhmeTranslateTmerMessage(NET_MhmeDcfmInd_s* psMhmeDcfmInd,NET_MhmeTmerInd_s*
psMhmeTmerInd);
0011
0012 //Individual processing of each message
0013 inline uint16 u16HdlMhmeAddressAssignment(NET_MhmeDcfmInd_s* psMhmeDcfmInd,NET_MhmeTmerInd_s*
psMhmeTmerInd);
0014 inline uint16 u16HdlMhmeCfmDiscover(NET_MhmeDcfmInd_s* psMhmeDcfmInd,MAC_MlmeDcfmInd_s* psMlmeDcfmInd);
0015 inline uint16 u16HdlMhmeCfmJoin(NET_MhmeDcfmInd_s* psMhmeDcfmInd,MAC_MlmeDcfmInd_s* psMlmeDcfmInd);
0016 inline uint16 u16HdlMhmeIndJoin(NET_MhmeDcfmInd_s* psMhmeDcfmInd,MAC_MlmeDcfmInd_s* psMlmeDcfmInd);
0017 inline uint16 u16HdlMhmeIndCommStatus(NET_MhmeDcfmInd_s* psMhmeDcfmInd,MAC_MlmeDcfmInd_s*
psMlmeDcfmInd);
0018 inline uint16 u16HdlMhmeCfmData(NET_MhmeDcfmInd_s* psMhmeDcfmInd,NET_MeshDcfmInd_s* psMeshDcfmInd);
0019 inline uint16 u16HdlMhmeIndData(NET_MhmeDcfmInd_s* psMhmeDcfmInd,NET_MeshDcfmInd_s* psMeshDcfmInd);
0020 inline uint16 u16HdlMhmeChildrenNumberReport(NET_MhmeDcfmInd_s* psMhmeDcfmInd,NET_MhmeTmerInd_s*
psMhmeTmerInd);
0021 inline uint16 u16HdlMhmeHello(NET_MhmeDcfmInd_s* psMhmeDcfmInd,NET_MhmeTmerInd_s* psMhmeTmerInd);
0022 inline uint16 u16HdlMhmeAppTimerExpired(NET_MhmeDcfmInd_s* psMhmeDcfmInd,NET_MhmeTmerInd_s*
psMhmeTmerInd);
0023
0024 #endif //MHME_CONTROL
```

XX. MhmeServices.c

```
0001 #include "Mhme.h"
0002 #include "MeshServices.h"
0003 #include "CommandFrames.h"
0004
0005 #include "string.h"
0006
0007 #include "VirtualTimer.h"
0008
0009 #ifdef DEBUG
0010 #include "Debugger.h"
0011 #endif
0012
0013 #define FLAG_DEVICE_TYPE 0x02
0014 #define FLAG_POWER_SOURCE 0x04
0015 #define FLAG_RECEIVER_ON_IDLE 0x08
0016
0017 #define FLAG_SECURITY 0x040
0018 #define FLAG_ADDR 0x080
0019
0020 #define MIN_LINK_QUALITY 1
0021
0022 #define SAME64ADDR(a,b) ((a.u32H==b.u32H)&&(a.u32L==b.u32L))
0023
0024 extern MAC_MlmeReqRsp_s sMlmeReqRsp;
0025 extern MAC_MlmeSyncCfm_s sMlmeSyncCfm;
0026 extern NET_MeshInfo_s sMeshInfo;
0027 extern NET_MeshInfo_s sMeshInfoCopy;
0028 extern NET_MeshData_s sMeshData;
0029 extern NET_MhmeMhsduHandleQueue_s sMhmeMhsduHandleQueue;
0030 extern NET_meshMsgQueue_s   sMeshQueue;
0031
0032 extern MAC_McpsBuffer_s psMcpsBuffers[N_MCPS_BUFFERS];
0033 extern MAC_MlmeBuffer_s psMlmeBuffers[N_MLME_BUFFERS];
```

```
0034
0035 /**AY*/
0036 _Bool REALIGNMENT=FALSE;/*False just in first attempt. 802.15.5*/
0037
0038 extern bool bJoinNetworkStatus;/*Not specified in 802.15.5*/
0039
0040
0041 #define OUR_MAC_H       0x00158d00
0042 #define OUR_MAC_L       0x000ade21
0043
0044
0045 void vMhmeInitializeMhsduHandleQueue(void)
0046 {
0047     memset(&sMhmeMhsduHandleQueue.u64MhsduHandleStatus,0,sizeof(uint64));
0048     sMhmeMhsduHandleQueue.u8IndexOfFirstFreeSpot=0;
0049     sMhmeMhsduHandleQueue.u8HndInQueue=0;
0050 }
0051
0052
0053 int16 i16MhmeInsertMhsduHandle(uint8 u8Type)
0054 {
0055     #ifdef DEBUG
0056     vPrintf("\n\n\nINSERTING HANDLE\n\tType:%d\n",u8Type);
0057     vPrintf("\tHANDLE in Queue:%d\n",sMhmeMhsduHandleQueue.u8HndInQueue);
0058     #endif
0059     if (sMhmeMhsduHandleQueue.u8HndInQueue<MAX_HND_IN_QUEUE)
0060     {
0061         uint64 u64Status=sMhmeMhsduHandleQueue.u64MhsduHandleStatus;
0062         uint16  ret=sMhmeMhsduHandleQueue.u8IndexOfFirstFreeSpot;
0063         uint64 u64StatusHp=1;
0064         uint8   i;
0065
0066
0067         sMhmeMhsduHandleQueue.asMhsduHandle[ret].u8Type=u8Type;
0068         u64StatusHp=u64StatusHp<<((MAX_HND_IN_QUEUE-1)-ret);
0069         u64Status|=u64StatusHp;
0070         #ifdef DEBUG
0071         vPrintf("\tStatus1bin:%x\n",u64Status);
0072         vPrintf("\tStatus1hex:%x\n",u64Status);
0073         vPrintf("\tFree SPOT1:%x\n",ret);
0074         #endif
0075         sMhmeMhsduHandleQueue.u8HndInQueue++;
0076         for (i=sMhmeMhsduHandleQueue.u8IndexOfFirstFreeSpot;i<MAX_HND_IN_QUEUE;i++)
0077         {
0078             if (!(u64StatusHp&u64Status))
0079             {
0080                 sMhmeMhsduHandleQueue.u8IndexOfFirstFreeSpot=i;
0081                 #ifdef DEBUG
0082                 vPrintf("\tFree SPOT2:%x\n",i);
0083                 #endif
0084                 break;
0085             }
0086             u64StatusHp=u64StatusHp>>1;
0087             #ifdef DEBUG
0088             vPrintf("\tStatusMEIObin:%x\n",u64Status);
0089             #endif
0090         }
0091         sMhmeMhsduHandleQueue.u64MhsduHandleStatus=u64Status;
0092         #ifdef DEBUG
0093         vPrintf("\tStatus2bin:%x\n",u64Status);
0094         vPrintf("\tStatus2hex:%x\n",u64Status);
0095         #endif
0096         return ret;
0097     }
0098
```

```
0099     #ifdef DEBUG
0100     vPrintf("\n\n\nBOOOM\n",u8Type);
0101     vPrintf("\tHANDLE in Queue:%d\n",sMhmeMhsduHandleQueue.u8HndInQueue);
0102     #endif
0103     return -1;
0104 }
0105
0106
0107 bool bMhmeValidateMhsduHandle(uint8 u8MhsduHandle)
0108 {
0109     uint64 u64Status=sMhmeMhsduHandleQueue.u64MhsduHandleStatus;
0110     uint64 u64StatusHp=1;
0111
0112     #ifdef DEBUG
0113     vPrintf("The status is %x\n",u64Status);
0114     #endif
0115     if (u8MhsduHandle>=MAX_HND_IN_QUEUE) return FALSE;
0116     u64StatusHp=u64StatusHp<<((MAX_HND_IN_QUEUE-1)-u8MhsduHandle);
0117     if (!(u64StatusHp&u64Status)) return FALSE;
0118     return TRUE;
0119 }
0120
0121 void vMhmeEraseMhsduHandle(uint8 u8MhsduHandle)
0122 {
0123     if (u8MhsduHandle<MAX_HND_IN_QUEUE)
0124     {
0125         uint64 u64Status=sMhmeMhsduHandleQueue.u64MhsduHandleStatus;
0126         uint64 u64StatusHp=1;
0127
0128         u64StatusHp=u64StatusHp<<((MAX_HND_IN_QUEUE-1)-u8MhsduHandle);
0129         u64StatusHp^=u64Status;
0130         u64Status&=u64StatusHp;
0131         sMhmeMhsduHandleQueue.u64MhsduHandleStatus=u64Status;
0132         sMhmeMhsduHandleQueue.u8HndInQueue--;
0133
sMhmeMhsduHandleQueue.u8IndexOfFirstFreeSpot=(u8MhsduHandle<sMhmeMhsduHandleQueue.u8IndexOfFirstFreeSpot)?\
0134                 u8MhsduHandle:\
0135                 sMhmeMhsduHandleQueue.u8IndexOfFirstFreeSpot;
0136     }
0137 }
0138
0139
0140
0141
0142
0143
0144 void vSetMeshInfoDefaultValues(void)
0145 {
0146     int i,j;
0147     uint32* pu32Mac=(uint32*)pvAppApiGetMacAddrLocation();
0148
0149     memset(&sMeshInfo,0,sizeof(NET_MeshInfo_s));
0150
0151     sMeshInfo.u8NbOfChildren = 0;
0152     sMeshInfo.u8CapabilityInformation = 0;
0153     sMeshInfo.u8TtlOfHello = 2;
0154     sMeshInfo.u8TreeLevel = 0;
0155     sMeshInfo.u16PanId = 0xffff;
0156     for(i=0;i<MAX_NEIGHBORS;i++)
0157     {
0158         sMeshInfo.psNeighborList[i].u16BeginningAddress=0xfffe;
0159         sMeshInfo.psNeighborList[i].u16EndingAddress=0xfffe;
0160         sMeshInfo.psNeighborList[i].u8TreeLevel=0xff;
0161         sMeshInfo.psNeighborList[i].u8Relationship=NO_RELATIONSHIP;
0162         sMeshInfo.psNeighborList[i].u8NumberOfOps=0xff;
```

```
0163         sMeshInfo.psNeighborList[i].u8NbOfHello=0;
0164         for(j=0;j<MAX_NEIGHBORS;j++)
0165             sMeshData.p2bConnectivityMatrix[i][j]=FALSE;
0166         sMeshInfo.psNeighborList[i].i8RejoinTimer=-1;
0167         sMeshInfo.psNeighborList[i].i8MsduHandleLeave=-1;
0168     }
0169     sMeshInfo.u8DeviceType = END_DEVICE;
0170     sMeshInfo.u8SequenceNumber = 45; //TODO: RANDOM VALUE
0171     sMeshInfo.u16NetworkAddress = 0xffff;
0172     sMeshInfo.psGroupCommTable = NULL;
0173     sMeshInfo.sAddressMapping.sExtAddr.u32H=pu32Mac[0];
0174     sMeshInfo.sAddressMapping.sExtAddr.u32L=pu32Mac[1];
0175     sMeshInfo.u8AcceptMeshDevice = FALSE;
0176     sMeshInfo.u8AcceptEndDevice = FALSE;
0177     sMeshInfo.u16ChildReportTime = 30;     //TODO: OPTIMIZE THIS VALUE
0178     sMeshInfo.u16ProbeInterval = 0x10;
0179     sMeshInfo.u8MaxProbeNum =  0xff;
0180     sMeshInfo.u16MaxProbeInterval = 0xffff;
0181     sMeshInfo.u8MaxMulticastJoinAttempts = 0x07;
0182     sMeshInfo.u16RbCastTxTimer = 1; //TODO: OPTIMIZE THIS VALUE
0183     sMeshInfo.u16RbCastRxTimer = 1; //TODO: OPTIMIZE THIS VALUE
0184     sMeshInfo.u8MaxRbCastTrials = 5; //TODO: OPTIMIZE THIS VALUE
0185     sMeshInfo.u8AsesOn = FALSE;
0186     sMeshInfo.u8AsesExpected = FALSE;
0187     sMeshInfo.u8WakeupOrder = 15;
0188     sMeshInfo.u8ActiveOrder = 15;
0189     sMeshInfo.u8DestActiveOrder =  0;
0190     sMeshInfo.u8EreqTime = 30;
0191     sMeshInfo.u8ErepTime = 15;
0192     sMeshInfo.u8DataTime = 15;
0193     sMeshInfo.u8MaxNumAsesRetries = 2;
0194     sMeshInfo.u8SesOn = FALSE;
0195     sMeshInfo.u8SesExpected = FALSE;
0196     sMeshInfo.u8SyncInterval = 0x0a;
0197     sMeshInfo.u8MaxSyncRequestAttempts = 3;
0198     sMeshInfo.u8SyncReplyWaitTime = 50;
0199     sMeshInfo.u32FirstTxSyncTime = 0;
0200     sMeshInfo.u32FirstRxSyncTime = 0;
0201     sMeshInfo.u32SecondRxSyncTime = 0;
0202     sMeshInfo.u8RegionSynchronizaerOn = FALSE;
0203     sMeshInfo.u8ExtendedNeighborHopDistance = 0x03;
0204     sMeshInfo.u16RejoinTimer = 5;
0205     sMeshInfo.sBitMapReadOnly.u32A=0x00000000;
0206     sMeshInfo.sBitMapReadOnly.u32B=0x00000000;
0207     memset(&sMeshInfo.sBitMapReadOnly,0x00,sizeof(NET_BitMap_s));
0208     PLACEBITMAP(sMeshInfo.sBitMapReadOnly,(MESH_SEQUENCE_NUMBER-MESH_PIB_MIN));
0209
0210     sMeshData.bIsCoordinator=FALSE;
0211     sMeshData.u16AddressAssigner=0;
0212     sMeshData.u8NbOfChildReportTimers=0;
0213     sMeshData.u8NbOfChildrenReports=0;
0214     sMeshData.i8ChilderReportTimer=-1;
0215     sMeshData.i8HelloTimer=-1;
0216     sMeshData.i8LeaveTimer=-1;
0217     sMeshData.i8AddressAssignmentTimer=-1;
0218     sMeshData.u8NbOfHello=0;
0219     sMeshData.u8MhmeHandle=0;
0220     sMeshData.u8MeshHandle=0;
0221     sMeshData.u8NbOfLeavingCommands=0;
0222     memset(&sMeshData.sBitMapLeave,0x00,sizeof(NET_BitMap_s));
0223     memset(&sMeshData.sBitMapNew,0x00,sizeof(NET_BitMap_s));
0224     memset(&sMeshData.sBitMapUpdate,0x00,sizeof(NET_BitMap_s));
0225     memset(&sMeshData.sBitMapRejoin,0x00,sizeof(NET_BitMap_s));
0226     memset(&sMeshData.sBitMapWaitingReport,0x00,sizeof(NET_BitMap_s));
0227     memset(&sMeshData.sBitMapAddressAssignment,0x00,sizeof(NET_BitMap_s));
```

```
0228     memset(&sMeshData.sBitMapNoNew,0x00,sizeof(NET_BitMap_s));
0229
0230     #ifdef DEBUG
0231     WRITEBITMAP(sMeshData.sBitMapLeave);
0232     #endif
0233
0234 }
0235
0236 PUBLIC void vInitMeshStack(void)
0237 {
0238     uint8 i;
0239
0240     //Register this function on the debugger
0241     #ifdef DEBUG
0242     u8StackPushIdentifier("MHME",strlen("MHME"),TRUE);
0243     u8StackPushIdentifier("vInitMeshStack",strlen("vInitMeshStack"),FALSE);
0244     #endif
0245
0246     #ifdef DEBUG
0247     vStackPrintf(__FILE__,__LINE__,"Initializing mesh stack.");
0248     #endif
0249
0250     u8MhmeFlag=MHME_BEGINNING;
0251
0252     InitializeMhsduHandleQueue();
0253
0254     bJoinNetworkStatus=FALSE;
0255
0256     for(i=0; i<N_MCPS_BUFFERS; i++)
0257     {
0258         psMcpsBuffers[i].u8Used = FALSE;
0259     }
0260
0261     for(i=0; i<N_MLME_BUFFERS; i++)
0262     {
0263         psMlmeBuffers[i].u8Used = FALSE;
0264     }
0265
0266     vSetMeshInfoDefaultValues();
0267
0268     VirtualTimer_vInit(1600000,&vTickTimerISR); //100 ms
0269
0270     memset(&sMhmeQueue,0x00,sizeof(NET_msgQueue_s));
0271     memset(&sMeshQueue,0x00,sizeof(NET_meshMsgQueue_s));
0272
0273     u32AppApiInit(psMlmeDcfmIndGetBuf, vMlmeDcfmIndPost, NULL, psMcpsDcfmIndGetBuf, vMcpsDcfmIndPost,
NULL);
0274
0275     //Deregister this function off the debugger
0276     #ifdef DEBUG
0277     vStackPopIdentifier(); //For the vInitMeshStack
0278     vStackPopIdentifier(); //For the MHME
0279     #endif
0280 }
0281
0282 PUBLIC void vResetMeshStack(void)
0283 {
0284     //Reseting the mhme flag
0285     u8MhmeFlag=MHME_BEGINNING;
0286
0287     //Not joined in the network anymore
0288     bJoinNetworkStatus=FALSE;
0289
0290     //Reseting mesh data timers
0291     if(sMeshData.i8ChilderReportTimer!=-1)
```

```
0292         VirtualTimer_bDelete(sMeshData.i8ChilderReportTimer);
0293     if(sMeshData.i8HelloTimer!=-1)
0294         VirtualTimer_bDelete(sMeshData.i8HelloTimer);
0295
0296     //Reset the mesh and mhme queues
0297     memset(&sMhmeQueue,0x00,sizeof(NET_msgQueue_s));
0298     memset(&sMeshQueue,0x00,sizeof(NET_meshMsgQueue_s));
0299
0300     //Reseting mesh info and mesh data values
0301     vSetMeshInfoDefaultValues();
0302 }
0303
0304 inline void vHdlMhmeReqStartNetwork(NET_MhmeReqStartNetwork_s *psMhmeReqStartNetwork,
NET_MhmeCfmStartNetwork_s *psMhmeCfmStartNetwork)
0305 {
0306     //Register this function on the debugger
0307     #ifdef DEBUG
0308     u8StackPushIdentifier("vHdlMhmeReqStartNetwork",strlen("vHdlMhmeReqStartNetwork"),FALSE);
0309     #endif
0310
0311     #if CONST_COORDINATOR_CAPABILITY == FALSE
0312
0313         psMhmeCfmStartNetwork->u8Status = CFM_STATUS_INVALID_REQUEST;
0314         return;
0315
0316     #endif
0317
0318     void        *pvMac;
0319     MAC_Pib_s *psPib;
0320     pvMac = pvAppApiGetMacHandle();
0321     psPib = MAC_psPibGetHandle(pvMac);
0322     MAC_vPibSetShortAddr(pvMac, 0xfffe);
0323     psPib->bAssociationPermit = 1;
0324     MAC_vPibSetRxOnWhenIdle(pvMac, 1, FALSE);
0325     vSetMeshInfoDefaultValues();
0326
0327     sMeshInfo.u16PanId = PAN_ID;
0328     sMeshInfo.u8DeviceType = MESH_DEVICE;
0329     sMeshInfo.u16NetworkAddress = 0xfffe;
0330
0331     sMeshData.bIsCoordinator=TRUE;
0332
0333     sMlmeReqRsp.u8Type = MAC_MLME_REQ_START;
0334     sMlmeReqRsp.u8ParamLength = sizeof(MAC_MlmeReqStart_s);
0335     sMlmeReqRsp.uParam.sReqStart.u16PanId = PAN_ID;
0336     sMlmeReqRsp.uParam.sReqStart.u8Channel = CHANNEL;
0337
0338     sMlmeReqRsp.uParam.sReqStart.u8BeaconOrder = psMhmeReqStartNetwork->u8BeaconOrder;
0339     sMlmeReqRsp.uParam.sReqStart.u8SuperframeOrder = psMhmeReqStartNetwork->u8SuperFrameOrder;
0340
0341     sMlmeReqRsp.uParam.sReqStart.u8PanCoordinator = TRUE;
0342     sMlmeReqRsp.uParam.sReqStart.u8Realignment = FALSE;
0343
0344     /**DP*/
0345     sMlmeReqRsp.uParam.sReqStart.u8BatteryLifeExt = FALSE;
0346     sMlmeReqRsp.uParam.sReqStart.u8SecurityEnable = FALSE;
0347     /**/
0348
0349     //Initializing the coordinator routing variables
0350     sMeshInfo.u16NetworkAddress=0x00;
0351     sMeshInfo.u8TreeLevel=0;
0352     sMeshData.u16AddressAssigner=0;
0353     sMeshData.u16EndingAddress=0xfffd;
0354
0355     //calculates block size
```

```
0356      sMeshData.u16BlockSize=0xfffd/MAX_NEIGHBORS;
0357      sMeshData.u8LeftAddr=0xfffd%MAX_NEIGHBORS;
0358      sMeshData.u16FreeBlocks=0xfffd/sMeshData.u16BlockSize;
0359      sMeshData.sFreeBlock[0].u16BeginningBlock=0;
0360      sMeshData.sFreeBlock[0].u16NumberOfBlocks=sMeshData.u16FreeBlocks;
0361      sMeshData.u16NumberOfFreeBlocksInList=1;
0362
0363      #ifdef DEBUG
0364      vStackPrintf(__FILE__,__LINE__,"Sending MlmeReqStart.");
0365      #endif
0366      vAppApiMlmeRequest(&sMlmeReqRsp, &sMlmeSyncCfm);
0367
0368      if(sMlmeSyncCfm.u8Status != MAC_MLME_CFM_OK)
0369      {
0370          #ifdef DEBUG
0371          vStackPrintf(__FILE__,__LINE__,"Error Starting network.");
0372          #endif
0373      }
0374      else
0375      {
0376          #ifdef DEBUG
0377          vStackPrintf(__FILE__,__LINE__,"Received MAC_MLME_CFM_OK.");
0378          #endif
0379      }
0380
0381      psMhmeCfmStartNetwork->u8Status = sMlmeSyncCfm.u8Status;
0382
0383      //Deregister this function off the debugger
0384      #ifdef DEBUG
0385      vStackPopIdentifier();
0386      #endif
0387  }
0388
0389
0390
0391
0392
0393  inline void vHdlMhmeReqDiscover(NET_MhmeReqDiscover_s *psMhmeReqDiscover, NET_MhmeCfmDiscover_s
*psMhmeCfmDiscover)
0394  {
0395      //Register this function on the debugger
0396      #ifdef DEBUG
0397      u8StackPushIdentifier("vHdlMhmeReqDiscover",strlen("vHdlMhmeReqDiscover"),FALSE);
0398      #endif
0399
0400      //If the report criteria is on tree level
0401      if(psMhmeReqDiscover->eReportCriteria==TREE_LEVEL)
0402      {
0403          psMhmeCfmDiscover->u8Status=UNSUPPORTED_ATTRIBUTE;
0404      }
0405
0406      //Filling the mlme request structure for an active scan request and sending the request
0407      else
0408      {
0409          #ifdef DEBUG
0410          vPrintf("%s.Reseting MAC Layer. . . %s:%d\n",__FUNCTION__,__FILE__,__LINE__);
0411          #endif
0412
0413          sMlmeReqRsp.u8Type = MAC_MLME_REQ_RESET;
0414          sMlmeReqRsp.u8ParamLength = sizeof(MAC_MlmeReqReset_s);
0415          sMlmeReqRsp.uParam.sReqReset.u8SetDefaultPib = TRUE;
0416
0417          vAppApiMlmeRequest(&sMlmeReqRsp, &sMlmeSyncCfm);
0418
0419          if(sMlmeSyncCfm.u8Status != MAC_MLME_CFM_OK)
```

```
0420        {
0421            #ifdef DEBUG
0422            vPrintf("%s.Error: Syncronous Confirm Received %s:%d\n",__FUNCTION__,__FILE__,__LINE__);
0423            #endif
0424        }
0425
0426        sMlmeReqRsp.u8Type = MAC_MLME_REQ_SCAN;
0427        sMlmeReqRsp.u8ParamLength = sizeof(MAC_MlmeReqScan_s);
0428        sMlmeReqRsp.uParam.sReqScan.u8ScanType = MAC_MLME_SCAN_TYPE_ACTIVE;
0429        sMlmeReqRsp.uParam.sReqScan.u32ScanChannels = psMhmeReqDiscover->u32ScanChannels;
0430        sMlmeReqRsp.uParam.sReqScan.u8ScanDuration = psMhmeReqDiscover->u8ScanDuration;
0431        vAppApiMlmeRequest(&sMlmeReqRsp, &sMlmeSyncCfm);
0432
0433        //Expecting a defered confirmation (i.e. scan in progress)
0434        if(sMlmeSyncCfm.u8Status!=MAC_MLME_CFM_DEFERRED)
0435        {
0436            #ifdef DEBUG
0437            vStackPrintf(__FILE__,__LINE__,"Unhandle scan exception\n");
0438            #endif
0439        }
0440        else
0441        {
0442            psMhmeCfmDiscover->u8Status=CFM_STATUS_DEFERED;
0443            #ifdef DEBUG
0444            vStackPrintf(__FILE__,__LINE__,"Scan in progress. . .");
0445            #endif
0446        }
0447    }
0448
0449    //Changing the mhme flag
0450    u8MhmeFlag=MHME_DISCOVERING;
0451
0452    //Deregister this function off the debugger
0453    #ifdef DEBUG
0454    vStackPopIdentifier();
0455    #endif
0456 }
0457
0458
0459
0460
0461 inline void vHdlMhmeReqJoin(NET_MhmeReqJoin_s *psMhmeReqJoin, NET_MhmeCfmJoin_s *psMhmeCfmJoin)
0462 {
0463    bool bMdtNeighbour=FALSE;
0464    bool bDeviceDiscovered=FALSE;
0465    uint8 u8NbNumber=0;
0466    uint8 i;
0467
0468    //validates join request
0469    if(psMhmeReqJoin->u8RejoinNetwork==0x00 && bJoinNetworkStatus )
0470    {
0471        #ifdef DEBUG
0472        vPrintf("%s.Already joined to a network. Expected Rejoin.
(%s:%d)\n",__FUNCTION__,__FILE__,__LINE__);
0473        #endif
0474
0475        psMhmeCfmJoin->u8Status=CFM_STATUS_INVALID_REQUEST;
0476        return;
0477    }
0478
0479    //multiplex join request
0480    if (psMhmeReqJoin->u8DirectJoin == TRUE)
0481    {
0482        //Search for the neighbour in the MDT
0483        if(psMhmeReqJoin->u8AddrMode==0x02)
```

```
0484        {
0485            for(i=0;i<MAC_MAX_SCAN_PAN_DESCRS;i++)
0486            {
0487                if(sMdt.asMeshDescriptors[i].uAddr.u16Short==psMhmeReqJoin->uParentDevAddr.u16Short)
0488                {
0489                    bDeviceDiscovered=TRUE;
0490                    break;
0491                }
0492            }
0493        }
0494        else
0495        {
0496            for(i=0;i<MAC_MAX_SCAN_PAN_DESCRS;i++)
0497            {
0498                if(!memcmp(&sMdt.asMeshDescriptors[i].uAddr.sExt,&psMhmeReqJoin-
>uParentDevAddr.sExt,sizeof(MAC_ExtAddr_s)))
0499                {
0500                    bDeviceDiscovered=TRUE;
0501                    break;
0502                }
0503            }
0504        }
0505
0506        if(!bDeviceDiscovered)
0507        {
0508            //Parent specified not found!
0509            psMhmeCfmJoin->u8Status=CFM_STATUS_NOT_PERMITTED;
0510            return;
0511        }
0512
0513         //Saves NbNumber
0514        u8NbNumber=i;
0515    }
0516    else
0517    {
0518        uint8 u8LinkQuality=0;
0519        //Search for neighbour with best linkquality
0520        for(i=0;i<MAC_MAX_SCAN_PAN_DESCRS;i++)
0521        {
0522            if(sMdt.asMeshDescriptors[i].u16PanId==psMhmeReqJoin->u16PanId)
0523            {
0524                if(sMdt.asMeshDescriptors[i].bAcceptMeshDevice)
0525                {
0526                    if(sMdt.asMeshDescriptors[i].u8LinkQuality >= MIN_LINK_QUALITY)
0527                    {
0528                        if(sMdt.asMeshDescriptors[i].u8LinkQuality>u8LinkQuality)
0529                        {
0530                            u8LinkQuality=sMdt.asMeshDescriptors[i].u8LinkQuality;
0531                            u8NbNumber=i;
0532                        }
0533                        bMdtNeighbour=TRUE;
0534                    }
0535                }
0536            }
0537        }
0538
0539        if(!bMdtNeighbour)
0540        {
0541            //No match was found
0542            psMhmeCfmJoin->u8Status=CFM_STATUS_NOT_PERMITTED;
0543            #ifdef DEBUG
0544            vPrintf("%s.Neighbour not found!(%s:%d)\n",__FUNCTION__,__FILE__,__LINE__);
0545            #endif
0546
0547            return;
```

```
0548        }
0549
0550    }
0551
0552    //Test if the device is ready to accept the join request
0553    if((sMdt.asMeshDescriptors[u8NbNumber].u16PanId!=psMhmeReqJoin->u16PanId)||
0554        (psMhmeReqJoin->u8JoinAsMeshDevice && !sMdt.asMeshDescriptors[u8NbNumber].bAcceptMeshDevice) ||
0555        (!psMhmeReqJoin->u8JoinAsMeshDevice && !sMdt.asMeshDescriptors[u8NbNumber].bAcceptEndDevice) )
0556    {
0557        psMhmeCfmJoin->u8Status=CFM_STATUS_NOT_PERMITTED;
0558        return;
0559    }
0560
0561    if(!bMdtNeighbour && !bDeviceDiscovered)
0562    {
0563        psMhmeCfmJoin->u8Status=CFM_STATUS_NOT_PERMITTED;
0564        return;
0565    }
0566
0567    #ifdef DEBUG
0568    vPrintf("%s.Join successful (Link Quality:
%d)!(%s:%d)\n",__FUNCTION__,sMdt.asMeshDescriptors[u8NbNumber].u8LinkQuality,__FILE__,__LINE__);
0569    #endif
0570
0571    #ifdef DEBUG
0572    if(psMhmeReqJoin->u8AddrMode==0x02)
0573        vPrintf("\t\tParent Address: %d\n",__FUNCTION__, psMhmeReqJoin->uParentDevAddr.u16Short);
0574    else
0575        vPrintf("\t\tParent Address: %x %x\n",psMhmeReqJoin->uParentDevAddr.sExt.u32H, psMhmeReqJoin-
>uParentDevAddr.sExt.u32L);
0576    #endif
0577
0578    //updates mesh info
0579    sMeshInfo.u16PanId=sMdt.asMeshDescriptors[u8NbNumber].u16PanId;
0580    if(psMhmeReqJoin->u8DeviceType)
0581        sMeshInfo.u8DeviceType=MESH_DEVICE;
0582    else
0583        sMeshInfo.u8DeviceType=END_DEVICE;
0584    for(i=0;i<MAX_NEIGHBORS;i++)
0585        if(sMeshInfo.psNeighborList[i].u8Relationship==NO_RELATIONSHIP) break;
0586    if(sMdt.asMeshDescriptors[u8NbNumber].u8AddrMode==0x03)
0587
memcpy(&sMeshInfo.psNeighborList[i].sExt,&sMdt.asMeshDescriptors[u8NbNumber].uAddr.sExt,sizeof(MAC_ExtAddr_s)
);
0588    else
0589
sMeshInfo.psNeighborList[i].u16BeginningAddress=sMdt.asMeshDescriptors[u8NbNumber].uAddr.u16Short;
0590    sMeshInfo.psNeighborList[i].u8LinkQuality=sMdt.asMeshDescriptors[u8NbNumber].u8LinkQuality;
0591    sMeshInfo.psNeighborList[i].u8Relationship=PARENT;
0592    if(psMhmeReqJoin->u8DeviceType) sMeshInfo.u8CapabilityInformation|=FLAG_DEVICE_TYPE;
0593    if(psMhmeReqJoin->u8PowerSource) sMeshInfo.u8CapabilityInformation|=FLAG_POWER_SOURCE;
0594    if(psMhmeReqJoin->u8ReceiverOnWhenIdle) sMeshInfo.u8CapabilityInformation|=FLAG_RECEIVER_ON_IDLE;
0595    if(psMhmeReqJoin->u8AllocateAddress) sMeshInfo.u8CapabilityInformation|=FLAG_ADDR;
0596
0597    //filling the mlme structure
0598    sMlmeReqRsp.u8Type = MAC_MLME_REQ_ASSOCIATE;
0599    sMlmeReqRsp.u8ParamLength = sizeof(MAC_MlmeReqAssociate_s);
0600    sMlmeReqRsp.uParam.sReqAssociate.sCoord.u16PanId=sMdt.asMeshDescriptors[u8NbNumber].u16PanId;
0601    sMlmeReqRsp.uParam.sReqAssociate.u8LogicalChan=sMdt.asMeshDescriptors[u8NbNumber].u8LogicalChannel;
0602    sMlmeReqRsp.uParam.sReqAssociate.u8Capability=sMeshInfo.u8CapabilityInformation;
0603    sMlmeReqRsp.uParam.sReqAssociate.u8SecurityEnable = FALSE;
0604    sMlmeReqRsp.uParam.sReqAssociate.sCoord.u8AddrMode=sMdt.asMeshDescriptors[u8NbNumber].u8AddrMode;
0605    if(psMhmeReqJoin->u8AddrMode==0x02)
0606
sMlmeReqRsp.uParam.sReqAssociate.sCoord.uAddr.u16Short=sMdt.asMeshDescriptors[u8NbNumber].uAddr.u16Short;
```

```
0607     else
0608
memcpy(&sMlmeReqRsp.uParam.sReqAssociate.sCoord.uAddr.sExt,&sMdt.asMeshDescriptors[u8NbNumber].uAddr.sExt,siz
eof(MAC_ExtAddr_s));
0609
0610     //Request association
0611     vAppApiMlmeRequest(&sMlmeReqRsp,&sMlmeSyncCfm);
0612
0613     //confirms success
0614     psMhmeCfmJoin->u8Status=sMlmeSyncCfm.u8Status==MAC_MLME_CFM_DEFERRED?SUCCESS:INVALID_REQUEST;
0615     #ifdef DEBUG
0616     sMlmeSyncCfm.u8Status==MAC_MLME_CFM_DEFERRED?
0617         vPrintf("SUCCESS in association"):
0618         vPrintf("FAILED  in association");
0619     #endif
0620     u8MhmeFlag=MHME_JOINING;
0621 }
0622
0623
0624
0625
0626
0627
0628
0629 inline void vHdlMhmeReqStartDevice(NET_MhmeReqStartDevice_s
*psMhmeReqStartDevice,NET_MhmeCfmStartDevice_s *psMhmeCfmStartDevice){
0630
0631     //Register this function on the debugger
0632     #ifdef DEBUG
0633     u8StackPushIdentifier("vHdlMhmeReqStartDevice",strlen("vHdlMhmeReqStartDevice"),FALSE);
0634     #endif
0635
0636     #ifdef DEBUG
0637     vPrintf("MHME: Proceeding MhmeReqStartDevice. BeaconOrder=%05x
SuperFrameOrder=%05x\n",psMhmeReqStartDevice->u8BeaconOrder, psMhmeReqStartDevice->u8SuperFrameOrder);
0638     #endif
0639
0640     if(sMeshInfo.u8DeviceType==MESH_DEVICE)
0641     {
0642         #ifdef DEBUG
0643        vPrintf("I'm a MESH DEVICE!!!\n");
0644        #endif
0645
0646         if(!bJoinNetworkStatus)
0647        {
0648            #ifdef DEBUG
0649             vStackPrintf(__FILE__,__LINE__,"Syncronous Start Device Status: INVALID REQUEST");
0650             #endif
0651
0652             psMhmeCfmStartDevice->u8Status=CFM_STATUS_INVALID_REQUEST;
0653             return;
0654        }
0655        else
0656        {
0657            void        *pvMac;
0658             MAC_Pib_s *psPib;
0659             pvMac = pvAppApiGetMacHandle();
0660             psPib = MAC_psPibGetHandle(pvMac);
0661             MAC_vPibSetShortAddr(pvMac, 0xfffe);
0662             psPib->bAssociationPermit = 1;
0663             MAC_vPibSetRxOnWhenIdle(pvMac, 1, FALSE);
0664
0665            sMlmeReqRsp.u8Type = MAC_MLME_REQ_START;
0666            sMlmeReqRsp.u8ParamLength = sizeof(MAC_MlmeReqStart_s);
0667            //Using parameters of vHdlMhmeReqStartDevice
```

```
0668            sMlmeReqRsp.uParam.sReqStart.u8BeaconOrder=psMhmeReqStartDevice->u8BeaconOrder;
0669            sMlmeReqRsp.uParam.sReqStart.u8SuperframeOrder = psMhmeReqStartDevice->u8SuperFrameOrder;
0670            //Other fields
0671            sMlmeReqRsp.uParam.sReqStart.u16PanId=PAN_ID;
0672            sMlmeReqRsp.uParam.sReqStart.u8Channel=CHANNEL;
0673            sMlmeReqRsp.uParam.sReqStart.u8PanCoordinator=FALSE;
0674            sMlmeReqRsp.uParam.sReqStart.u8BatteryLifeExt=FALSE;
0675            sMlmeReqRsp.uParam.sReqStart.u8Realignment=REALIGNMENT;
0676            REALIGNMENT=TRUE;
0677            sMlmeReqRsp.uParam.sReqStart.u8SecurityEnable=FALSE;
0678            vAppApiMlmeRequest(&sMlmeReqRsp, &sMlmeSyncCfm);
0679        }
0680
0681        /* Handle synchronous confirm */
0682            if (sMlmeSyncCfm.u8Status != MAC_MLME_CFM_OK)
0683            {
0684                #ifdef DEBUG
0685                 vStackPrintf(__FILE__,__LINE__,"Syncronous Start Device Status: ERROR");
0686                 #endif
0687
0688                psMhmeCfmStartDevice->u8Status=CFM_STATUS_INVALID_REQUEST;
0689                REALIGNMENT=FALSE;
0690                return;
0691            }else
0692            {
0693
0694                 psMhmeCfmStartDevice->u8Status=CFM_STATUS_SYNC_SUCCESS;
0695
0696                 #ifdef DEBUG
0697                 vStackPrintf(__FILE__,__LINE__,"Device Started!");
0698                 vPrintf("Device Type:%d\n",sMeshInfo.u8DeviceType);
0699                 #endif
0700
0701                 return;
0702             }
0703
0704
0705            // if (sMlmeSyncCfm.u8Status ==MAC_ENUM_UNAVAILABLE_KEY)
0706                // {
0707
0708                // }
0709
0710            // if (sMlmeSyncCfm.u8Status ==MAC_ENUM_FRAME_TOO_LONG)
0711                // {
0712
0713                // }
0714
0715            // if (sMlmeSyncCfm.u8Status ==MAC_ENUM_FAILED_SECURITY_CHECK)
0716                // {
0717
0718                // }
0719
0720            // if (sMlmeSyncCfm.u8Status ==MAC_ENUM_INVALID_PARAMETER)
0721                // {
0722
0723                // }
0724
0725            // if (sMlmeSyncCfm.u8Status ==MAC_ENUM_SUCCESS)
0726                // {
0727
0728                // }
0729    }
0730    else
0731         psMhmeCfmStartDevice->u8Status=CFM_STATUS_NOT_PERMITTED;
0732
```

```
0733     #ifdef DEBUG
0734     vStackPopIdentifier();
0735     #endif
0736
0737 }
0738
0739
0740
0741
0742
0743 inline void vHdlMhmeReqGet(NET_MhmeReqGet_s* psMhmeReqLeave,NET_MhmeCfmGet_s* psMhmeCfmGet)
0744 {
0745     //Register this function on the debugger
0746     #ifdef DEBUG
0747     u8StackPushIdentifier("vHdlMhmeReqGet",strlen("vHdlMhmeReqGet"),FALSE);
0748     #endif
0749
0750     //Find the attribute to return to the upper layer
0751     switch(psMhmeReqLeave->u8MeshIBAttribute)
0752     {
0753         case MESH_NEIGHBOR_LIST:
0754         {
0755             psMhmeCfmGet->eStatus=psMhmeReqLeave->u8MeshIBAttribute;
0756             psMhmeCfmGet->eStatus=SUCCESS;
0757             psMhmeCfmGet->u16MibAttributeLength=sizeof(sMeshInfoCopy.psNeighborList);
0758
memcpy(sMeshInfoCopy.psNeighborList,sMeshInfo.psNeighborList,sizeof(sMeshInfoCopy.psNeighborList));
0759             psMhmeCfmGet->psMibAttributeValue=(uint8*)sMeshInfoCopy.psNeighborList;
0760         }break;
0761         case MESH_NETWORK_ADDRESS:
0762         {
0763             psMhmeCfmGet->eStatus=psMhmeReqLeave->u8MeshIBAttribute;
0764             psMhmeCfmGet->eStatus=SUCCESS;
0765             psMhmeCfmGet->u16MibAttributeLength=sizeof(sMeshInfoCopy.u16NetworkAddress);
0766
memcpy(&sMeshInfoCopy.u16NetworkAddress,&sMeshInfo.u16NetworkAddress,sizeof(sMeshInfoCopy.u16NetworkAddress))
;
0767             psMhmeCfmGet->psMibAttributeValue=(uint8*)&sMeshInfoCopy.u16NetworkAddress;
0768         }break;
0769         case MESH_ADDRESS_MAPPING:
0770         {
0771             psMhmeCfmGet->eStatus=psMhmeReqLeave->u8MeshIBAttribute;
0772             psMhmeCfmGet->eStatus=SUCCESS;
0773             psMhmeCfmGet->u16MibAttributeLength=sizeof(sMeshInfoCopy.sAddressMapping);
0774
memcpy(&sMeshInfoCopy.sAddressMapping,&sMeshInfo.sAddressMapping,sizeof(sMeshInfoCopy.sAddressMapping));
0775             psMhmeCfmGet->psMibAttributeValue=(uint8*)&sMeshInfoCopy.sAddressMapping;
0776         }break;
0777         case MESH_SEQUENCE_NUMBER:
0778         {
0779             psMhmeCfmGet->eStatus=psMhmeReqLeave->u8MeshIBAttribute;
0780             psMhmeCfmGet->eStatus=SUCCESS;
0781             psMhmeCfmGet->u16MibAttributeLength=sizeof(sMeshInfoCopy.u8SequenceNumber);
0782
memcpy(&sMeshInfoCopy.u8SequenceNumber,&sMeshInfo.u8SequenceNumber,sizeof(sMeshInfoCopy.u8SequenceNumber));
0783             psMhmeCfmGet->psMibAttributeValue=(uint8*)&sMeshInfoCopy.u8SequenceNumber;
0784         }
0785
0786         //Case no parameter was found
0787         default:
0788         {
0789             //We should first see if it was a MAC/PHY valid parameter
0790             psMhmeCfmGet->eStatus=psMhmeReqLeave->u8MeshIBAttribute;
0791             psMhmeCfmGet->eStatus=UNSUPPORTED_ATTRIBUTE;
0792             psMhmeCfmGet->u16MibAttributeLength=0;
```

```
0793            psMhmeCfmGet->psMibAttributeValue=(uint8*)NULL;
0794        }
0795    }
0796
0797    //Deregister this function off the debugger
0798    #ifdef DEBUG
0799    vStackPopIdentifier();
0800    #endif
0801 }
0802
0803
0804
0805
0806
0807 inline void vHdlMhmeReqLeave(NET_MhmeReqLeave_s* psMhmeReqLeave,NET_MhmeCfmLeave_s* psMhmeCfmLeave)
0808 {
0809    NET_CommSyncCfm_s sCommSyncCfm;
0810
0811    #ifdef DEBUG
0812    vPrintf("I am here at leaving.\n");
0813    #endif
0814
0815    //Register this function on the debugger
0816    #ifdef DEBUG
0817    u8StackPushIdentifier("vHdlMhmeReqLeave",strlen("vHdlMhmeReqLeave"),FALSE);
0818    #endif
0819
0820    //If it has to remove itself from the network or it is another device to be removed
0821    if(psMhmeReqLeave->u8RemoveSelf)
0822    {
0823        uint8   i;
0824
0825        //Change the network state to leaving
0826        u8MhmeFlag=MHME_LEAVING;
0827
0828        #ifdef DEBUG
0829        vPrintf("I am leaving!!!!!\n");
0830        #endif
0831
0832        //Find if it has to remove its children
0833        if((psMhmeReqLeave->u8RemoveChildren)&&(sMeshInfo.u8NbOfChildren>0))
0834        {
0835            //Remove the children
0836            for(i=0;i<MAX_NEIGHBORS;i++)
0837                if(sMeshInfo.psNeighborList[i].u8Relationship==CHILD)
0838                {
0839                    PLACEBITMAP(sMeshData.sBitMapLeave,i);
0840                    sMeshData.u8NbOfLeavingCommands++;
0841                }
0842
0843            vNetApiCommRequest(FRAME_LEAVE,&sCommSyncCfm);
0844
0845            //Start timer for leaving commands
0846            if(sMeshData.i8LeaveTimer==-1)
0847            {
0848                #ifdef DEBUG
0849                vPrintf("Leave timer set and ready to go.\n");
0850                #endif
0851
0852                sMeshData.i8LeaveTimer=VirtualTimer_i8New(NET_MESH_LEAVE_TIMER*10); //x * (10 * 100)ms
0853                VirtualTimer_bReset(sMeshData.i8LeaveTimer);
0854                VirtualTimer_bCount(sMeshData.i8LeaveTimer);
0855
0856                psMhmeCfmLeave->u8Status=CFM_STATUS_DEFERED;
0857            }
```

```
0858        }
0859        else
0860        {
0861            //Sending hello command frame to leave the network
0862            vNetApiCommRequest(FRAME_HELLO,&sCommSyncCfm);
0863
0864            //Issuing an mlme reset
0865            sMlmeReqRsp.u8Type=MAC_MLME_REQ_RESET;
0866            sMlmeReqRsp.u8ParamLength=sizeof(MAC_MlmeReqReset_s);
0867            sMlmeReqRsp.uParam.sReqReset.u8SetDefaultPib=TRUE;
0868            vAppApiMlmeRequest(&sMlmeReqRsp,&sMlmeSyncCfm);
0869
0870            //Clear references - reset meshInfo and meshData
0871            vResetMeshStack();
0872
0873            //Issuing a sincronous comfirm
0874            psMhmeCfmLeave->u8Status=CFM_STATUS_SYNC_SUCCESS;
0875            memset(&psMhmeCfmLeave->sDeviceAddress,0,sizeof(MAC_ExtAddr_s));
0876        }
0877    }
0878    else
0879    {
0880        uint8   i;
0881        bool    bFoundChild=FALSE;
0882
0883        //Find the children it has to remove
0884        for(i=0;i<MAX_NEIGHBORS;i++)
0885        {
0886            if( (sMeshInfo.psNeighborList[i].u8Relationship==CHILD)&&
0887                (SAME64ADDR(sMeshInfo.psNeighborList[i].sExt,psMhmeReqLeave->sDeviceAddress)))
0888            {
0889                bFoundChild=TRUE;
0890
0891                #ifdef DEBUG
0892                vPrintf("I am here at leaving.\n");
0893                #endif
0894
0895                //Set child for removal
0896                if(!READBITMAP(sMeshData.sBitMapLeave,i))
0897                {
0898                    #ifdef DEBUG
0899                    WRITEBITMAP(sMeshData.sBitMapLeave);
0900                    #endif
0901                    PLACEBITMAP(sMeshData.sBitMapLeave,i);
0902                    #ifdef DEBUG
0903                    WRITEBITMAP(sMeshData.sBitMapLeave);
0904                    #endif
0905                    sMeshInfo.psNeighborList[i].u8NbOfLeaveRetries=10;
0906                    sMeshInfo.psNeighborList[i].bRemoveChildren=psMhmeReqLeave->u8RemoveChildren;
0907                    sMeshData.u8NbOfLeavingCommands++;
0908
0909                    #ifdef DEBUG
0910                    vPrintf("Sending the child to leave.\n");
0911                    #endif
0912
0913                    //Send the leave commands
0914                    vNetApiCommRequest(FRAME_LEAVE,&sCommSyncCfm);
0915                }
0916
0917                //Start timer for leaving commands
0918                if((sMeshData.u8NbOfLeavingCommands>0)&&(sMeshData.i8LeaveTimer==-1))
0919                {
0920                    #ifdef DEBUG
0921                    vPrintf("Leave timer set and ready to go.\n");
0922                    #endif
```

```
0923
0924                    sMeshData.i8LeaveTimer=VirtualTimer_i8New(NET_MESH_LEAVE_TIMER*10); //x * (10 *
100)ms
0925                    VirtualTimer_bReset(sMeshData.i8LeaveTimer);
0926                    VirtualTimer_bCount(sMeshData.i8LeaveTimer);
0927                }
0928
0929                //Now the device will be wating for the confirms of the leave command frames
0930                psMhmeCfmLeave->u8Status=CFM_STATUS_DEFERED;
0931                memcpy(&psMhmeCfmLeave->sDeviceAddress,&psMhmeReqLeave-
>sDeviceAddress,sizeof(MAC_ExtAddr_s));
0932                break;
0933            }
0934        }
0935
0936        //If it has not found a child
0937        if(!bFoundChild)
0938        {
0939            psMhmeCfmLeave->u8Status=UNKNOWN_CHILD_DEVICE;
0940            memset(&psMhmeCfmLeave->sDeviceAddress,0,sizeof(MAC_ExtAddr_s));
0941        }
0942    }
0943
0944    //Deregister this function off the debugger
0945    #ifdef DEBUG
0946    vStackPopIdentifier();
0947    #endif
0948 }
0949
0950
0951
0952 inline void vHdlMhmeReqReset(NET_MhmeCfmReset_s* psMhmeCfmReset)
0953 {
0954
0955    //Send leave command frame if joined
0956    if(bJoinNetworkStatus)
0957    {
0958        NET_MhmeReqLeave_s sMhmeReqLeave;
0959        NET_MhmeCfmLeave_s sMhmeCfmLeave;
0960
0961
0962        sMhmeReqLeave.u8RemoveSelf=TRUE;
0963        sMhmeReqLeave.u8RemoveChildren=FALSE;
0964
0965        //Request to leave network
0966        vHdlMhmeReqLeave(&sMhmeReqLeave, &sMhmeCfmLeave);
0967
0968        //Issues a confirm
0969        psMhmeCfmReset->u8Status = sMhmeCfmLeave.u8Status;
0970    }
0971    else
0972    {
0973
0974        //Issuing an mlme reset
0975        sMlmeReqRsp.u8Type=MAC_MLME_REQ_RESET;
0976        sMlmeReqRsp.u8ParamLength=sizeof(MAC_MlmeReqReset_s);
0977        sMlmeReqRsp.uParam.sReqReset.u8SetDefaultPib=TRUE;
0978        vAppApiMlmeRequest(&sMlmeReqRsp,&sMlmeSyncCfm);
0979
0980        //handles synchronous confirm
0981        vResetMeshStack();
0982
0983        //Issues a confirm
0984        psMhmeCfmReset->u8Status = sMlmeSyncCfm.u8Status;
0985    }
```

```
0986
0987 }
0988
0989
0990
0991 inline void vHdlMhmeReqSet(NET_MhmeReqSet_s* psMhmeReqSet,NET_MhmeCfmSet_s* psMhmeCfmSet)
0992 {
0993     //fills the confirm with the requested atrtibute
0994     psMhmeCfmSet->u8MibAttribute = psMhmeReqSet->u8MibAttribute;
0995
0996     //Understands if it is on a valid range
0997     if((psMhmeReqSet->u8MibAttribute<MESH_PIB_MIN)||(psMhmeReqSet->u8MibAttribute>MESH_PIB_MAX))
0998         psMhmeCfmSet->u8Status = INVALID_PARAMETER;
0999
1000     //Understands if it is read only
1001     else if(!READBITMAP(sMeshInfo.sBitMapReadOnly,(psMhmeReqSet->u8MibAttribute-MESH_PIB_MIN)))
1002         psMhmeCfmSet->u8Status = UNSUPPORTED_ATTRIBUTE;
1003     else
1004     {
1005         //sets a value on the mesh pib
1006         switch(psMhmeReqSet->u8MibAttribute)
1007         {
1008             case MESH_SEQUENCE_NUMBER:
1009             {
1010                 //Understand if the size is appropriate for the request
1011                 if(sizeof(sMeshInfo.u8SequenceNumber)!=psMhmeReqSet->u16MibAttributeLength)
1012                     psMhmeCfmSet->u8Status = INVALID_REQUEST;
1013                 else
1014                 {
1015                     //Perform the set of the variable
1016                     memcpy(&sMeshInfo.u8SequenceNumber,psMhmeReqSet->psMibAttributeValue,psMhmeReqSet-
>u16MibAttributeLength);
1017                     psMhmeCfmSet->u8Status = SUCCESS;
1018                 }
1019             }
1020             break;
1021
1022             default:
1023             {
1024                 psMhmeCfmSet->u8Status = (psMhmeReqSet->u8MibAttribute > MAC_PIB_MAX || psMhmeReqSet-
>u8MibAttribute < MAC_PIB_MIN) ||
1025                     (psMhmeReqSet->u8MibAttribute > MESH_PIB_MAX || psMhmeReqSet->u8MibAttribute <
MESH_PIB_MIN)?
1026                     INVALID_PARAMETER : UNSUPPORTED_ATTRIBUTE;
1027             }
1028             break;
1029         }
1030     }
1031 }
1032
```

XXI. MhmeServices.h

```
0001 #ifndef MHME_SERVICES
0002 #define MHME_SERVICES
0003
0004
0005 #include "mhme.h"
0006
0007 //Msdu handle que processing functions
0008
0009
0010 //Initialize handle queue
0011 void vMhmeInitializeMhsduHandleQueue(void);
```

```
0012
0013 //Insert a handle
0014 int16 i16MhmeInsertMhsduHandle(uint8 u8Type);
0015
0016 //Validate a handle
0017 bool bMhmeValidateMhsduHandle(uint8 u8MhsduHandle);
0018
0019 //Erase a handle
0020 void vMhmeEraseMhsduHandle(uint8 u8MhsduHandle);
0021
0022
0023 //Suported services for the mhme
0024
0025
0026 //Initializing mesh stack
0027 void vSetMeshInfoDefaultValues(void);
0028
0029 //Starting the network
0030 inline void vHdlMhmeReqStartNetwork(NET_MhmeReqStartNetwork_s *psMhmeReqStartNetwork,
NET_MhmeCfmStartNetwork_s *psMhmeCfmStartNetwork);
0031
0032 //Desovering the network
0033 inline void vHdlMhmeReqDiscover(NET_MhmeReqDiscover_s *psMhmeReqDiscover, NET_MhmeCfmDiscover_s
*psMhmeCfmDiscover);
0034
0035 //Joining the network
0036 inline void vHdlMhmeReqJoin(NET_MhmeReqJoin_s *psMhmeReqJoin, NET_MhmeCfmJoin_s *psMhmeCfmJoin);
0037
0038 //Initializing mesh device
0039 inline void vHdlMhmeReqStartDevice(NET_MhmeReqStartDevice_s
*psMhmeReqStartDevice,NET_MhmeCfmStartDevice_s *psMhmeCfmStartDevice);
0040
0041 //Get a mesh parameter
0042 inline void vHdlMhmeReqGet(NET_MhmeReqGet_s* psMhmeReqLeave,NET_MhmeCfmGet_s* psMhmeCfmGet);
0043
0044 //Set a mesh parameter
0045 inline void vHdlMhmeReqSet(NET_MhmeReqSet_s* psMhmeReqSet,NET_MhmeCfmSet_s* psMhmeCfmSet);
0046
0047 //Reset mesh network
0048 inline void vHdlMhmeReqReset(NET_MhmeCfmReset_s* psMhmeCfmReset);
0049
0050 //Leaving the network
0051 inline void vHdlMhmeReqLeave(NET_MhmeReqLeave_s* psMhmeReqLeave,NET_MhmeCfmLeave_s* psMhmeCfmLeave);
0052
0053 //For mlme responses of command frames
0054 inline uint16 u16HdlMhmeIndCommStatus(NET_MhmeDcfmInd_s* psMhmeDcfmInd,MAC_MlmeDcfmInd_s*
psMlmeDcfmInd);
0055
0056 #endif //MHME_SERVICES
```

XXII. VirtualTimer.c

```
0001 #include <jendefs.h>
0002 #include <AppHardwareApi.h>
0003
0004 #ifdef DEBUG
0005 #include "Debugger.h"
0006 #endif
0007
0008 //Some defeintions
0009 #define MAX_HANDLES     32
0010
0011 bool bDebug2=FALSE;
0012
```

```
0013 /* DEFINE HANDLE DATA HERE */
0014
0015 typedef struct
0016 {
0017     uint8   bCounting;
0018     uint8   u8Pad;
0019     uint16  u16Pad;
0020     uint32  u32Count;
0021     uint32  u32End;
0022 }VirtualTimer_s;
0023
0024 typedef struct
0025 {
0026     VirtualTimer_s      asHandles[MAX_HANDLES];
0027     uint32              u32HandleStatus;
0028     uint8               u8FirstFreedHandle;
0029     uint8               u8ExistingHandles;
0030     uint16              u16Pad;
0031 }VirtualTimer_List_s;
0032
0033 VirtualTimer_List_s sVirtualTimer_List;
0034
0035 void   VirtualTimer_vInitialize(void)
0036 {
0037     sVirtualTimer_List.u32HandleStatus=0L;
0038     sVirtualTimer_List.u8FirstFreedHandle=0;
0039     sVirtualTimer_List.u8ExistingHandles=0;
0040 }
0041 bool   VirtualTimer_bValidateHandle(uint8 u8Handle)
0042 {
0043     uint32 u32Status=sVirtualTimer_List.u32HandleStatus;
0044     uint32 u32StatusHp=1;
0045
0046     if(u8Handle>=MAX_HANDLES) return FALSE;
0047     u32StatusHp=u32StatusHp<<((MAX_HANDLES-1)-u8Handle);
0048     return (u32StatusHp&u32Status)!=0L;
0049 }
0050 int8   VirtualTimer_i8InsertHandle(void)
0051 {
0052     #ifdef DEBUG
0053     vPrintf("\tHANDLE in Queue:%d\n",sVirtualTimer_List.u8ExistingHandles);
0054     #endif
0055     if(sVirtualTimer_List.u8ExistingHandles<MAX_HANDLES)
0056     {
0057         uint32 u32Status=sVirtualTimer_List.u32HandleStatus;
0058         uint16 ret=sVirtualTimer_List.u8FirstFreedHandle;
0059         uint32 u32StatusHp=1;
0060         uint8  i;
0061
0062         u32StatusHp=u32StatusHp<<((MAX_HANDLES-1)-ret);
0063         u32Status|=u32StatusHp;
0064         #ifdef DEBUG
0065         vPrintf("\tStatus1bin:%x\n",u32Status);
0066         vPrintf("\tStatus1hex:%x\n",u32Status);
0067         vPrintf("\tFree SPOT1:%x\n",ret);
0068         #endif
0069         sVirtualTimer_List.u8ExistingHandles++;
0070         for(i=ret;i<MAX_HANDLES;i++)
0071         {
0072             if(!(u32StatusHp&u32Status))
0073             {
0074                 sVirtualTimer_List.u8FirstFreedHandle=i;
0075                 #ifdef DEBUG
0076                 vPrintf("\tFree SPOT2:%x\n",i);
0077                 #endif
```

```
0078                break;
0079            }
0080            u32StatusHp=u32StatusHp>>1;
0081            #ifdef DEBUG
0082            vPrintf("\tStatusMEIObin:%x\n",u32Status);
0083            #endif
0084        }
0085        #ifdef DEBUG
0086        sVirtualTimer_List.u32HandleStatus=u32Status;
0087        vPrintf("\tStatus2bin:%x\n",u32Status);
0088        vPrintf("\tStatus2hex:%x\n",u32Status);
0089        #endif
0090        return ret;
0091    }
0092    return -1;
0093 }
0094 bool   VirtualTimer_bDeleteHandle(uint8 u8Handle)
0095 {
0096    if(u8Handle<MAX_HANDLES)
0097    {
0098        uint32 u32Status=sVirtualTimer_List.u32HandleStatus;
0099        uint32 u32StatusHp=1;
0100
0101        u32StatusHp=u32StatusHp<<((MAX_HANDLES-1)-u8Handle);
0102        if(!(u32StatusHp&u32Status)) return FALSE;
0103        u32StatusHp^=u32Status;
0104        u32Status&=u32StatusHp;
0105        sVirtualTimer_List.u32HandleStatus=u32Status;
0106        sVirtualTimer_List.u8ExistingHandles--;
0107
sVirtualTimer_List.u8FirstFreedHandle=(u8Handle<sVirtualTimer_List.u8FirstFreedHandle)?u8Handle:sVirtualTimer
_List.u8FirstFreedHandle;
0108        return TRUE;
0109     }
0110     return FALSE;
0111 }
0112
0113
0114 /* VIRTUAL TIMER API */
0115
0116 uint8                  u8NumberOfCountingTimers;
0117 uint32                 u32TickTimerTimeUnit;
0118 void (*pvISRFunctionLocation)(uint32,uint32);
0119
0120 void VirtualTimer_vTickTimerISR(uint32 u32Device, uint32 u32ItemBitmap);
0121
0122 void   VirtualTimer_vInit(uint32 u32TimeUnit,void* pvISRFunction)
0123 {
0124    u32TickTimerTimeUnit=u32TimeUnit;
0125    u8NumberOfCountingTimers=0;
0126    vAHI_TickTimerInterval(u32TimeUnit);
0127    vAHI_TickTimerIntEnable(TRUE);
0128    vAHI_TickTimerInit(&VirtualTimer_vTickTimerISR);
0129    vAHI_TickTimerConfigure(E_AHI_TICK_TIMER_DISABLE);
0130    pvISRFunctionLocation=pvISRFunction;
0131    VirtualTimer_vInitialize();
0132 }
0133 int8   VirtualTimer_i8New(uint32 u32Time)
0134 {
0135    int8 ret=VirtualTimer_i8InsertHandle();
0136    if(ret!=-1)
0137    {
0138        sVirtualTimer_List.asHandles[ret].bCounting=FALSE;
0139        sVirtualTimer_List.asHandles[ret].u32Count=0;
0140        sVirtualTimer_List.asHandles[ret].u32End=u32Time*u32TickTimerTimeUnit;
```

```
0141     }
0142     return ret;
0143 }
0144 bool   VirtualTimer_bCount(uint8 u8Handle)
0145 {
0146
0147     if(VirtualTimer_bValidateHandle(u8Handle))
0148     {
0149         if(!sVirtualTimer_List.asHandles[u8Handle].bCounting)
0150         {
0151             sVirtualTimer_List.asHandles[u8Handle].bCounting=TRUE;
0152             if(u8NumberOfCountingTimers==0)
0153                 vAHI_TickTimerConfigure(E_AHI_TICK_TIMER_RESTART);
0154             u8NumberOfCountingTimers++;
0155         }
0156         return TRUE;
0157     }
0158     return FALSE;
0159 }
0160 bool   VirtualTimer_bStop(uint8 u8Handle)
0161 {
0162     if(VirtualTimer_bValidateHandle(u8Handle))
0163     {
0164         if(sVirtualTimer_List.asHandles[u8Handle].bCounting)
0165         {
0166             sVirtualTimer_List.asHandles[u8Handle].bCounting=FALSE;
0167             u8NumberOfCountingTimers--;
0168             if(u8NumberOfCountingTimers==0)
0169                 vAHI_TickTimerConfigure(E_AHI_TICK_TIMER_DISABLE);
0170         }
0171         return TRUE;
0172     }
0173     return FALSE;
0174 }
0175 bool   VirtualTimer_bReset(uint8 u8Handle)
0176 {
0177     if(VirtualTimer_bValidateHandle(u8Handle))
0178     {
0179         sVirtualTimer_List.asHandles[u8Handle].u32Count=0;
0180         return TRUE;
0181     }
0182     return FALSE;
0183 }
0184 bool   VirtualTimer_bSet(uint8 u8Handle,uint32 u32Value)
0185 {
0186     if(VirtualTimer_bValidateHandle(u8Handle))
0187     {
0188         sVirtualTimer_List.asHandles[u8Handle].u32Count=u32Value*u32TickTimerTimeUnit;
0189         return TRUE;
0190     }
0191     return FALSE;
0192 }
0193 bool   VirtualTimer_bChange(uint8 u8Handle,uint32 u32Value)
0194 {
0195     if(VirtualTimer_bValidateHandle(u8Handle))
0196     {
0197         sVirtualTimer_List.asHandles[u8Handle].u32End=u32Value*u32TickTimerTimeUnit;
0198         return TRUE;
0199     }
0200     return FALSE;
0201 }
0202
0203 bool   VirtualTimer_bDelete(uint8 u8Handle) {return VirtualTimer_bDeleteHandle(u8Handle);}
0204
0205 void   VirtualTimer_vUpdate(uint32 u32Count,uint32* pu32TimerBitMap)
```

```
0206 {
0207     uint32 bitMap=0L;
0208     uint32 filter=1;
0209     int8   i;
0210 
0211     for(i=0;i<MAX_HANDLES;i++)
0212     {
0213         if(VirtualTimer_bValidateHandle(i))
0214         {
0215             if(sVirtualTimer_List.asHandles[i].bCounting)
0216             {
0217 
if(sVirtualTimer_List.asHandles[i].u32Count+u32Count>=sVirtualTimer_List.asHandles[i].u32End)
0218                     bitMap|=filter;
0219 
sVirtualTimer_List.asHandles[i].u32Count=(sVirtualTimer_List.asHandles[i].u32Count+u32Count)%sVirtualTimer_Li
st.asHandles[i].u32End;
0220             }
0221         }
0222         filter=filter<<1;
0223     }
0224     *pu32TimerBitMap=bitMap;
0225 }
0226 
0227 void VirtualTimer_vTickTimerISR(uint32 u32Device, uint32 u32ItemBitmap)
0228 {
0229     uint32  u32TimerBitMap=0L;
0230 
0231     if(u8NumberOfCountingTimers>0)
0232     {
0233         VirtualTimer_vUpdate(u32TickTimerTimeUnit,&u32TimerBitMap);
0234         if(u32TimerBitMap!=0L)
0235             if(pvISRFunctionLocation!=NULL)
0236                 pvISRFunctionLocation(0,u32TimerBitMap);
0237     }
0238 }
```

XXIII. VirtualTimer.h

```
0001 #ifndef VIRTUAL_TIMER
0002 #define VIRTUAL_TIMER
0003 
0004 #include <jendefs.h>
0005 
0006 void   VirtualTimer_vInit(uint32 u32TimeUnit,void* pvISRFunction);
0007 int8   VirtualTimer_i8New(uint32 u32Time);
0008 bool   VirtualTimer_bCount(uint8 u8Handle);
0009 bool   VirtualTimer_bStop(uint8 u8Handle);
0010 bool   VirtualTimer_bReset(uint8 u8Handle);
0011 bool   VirtualTimer_bSet(uint8 u8Handle,uint32 u32Value);
0012 bool   VirtualTimer_bChange(uint8 u8Handle,uint32 u32Value);
0013 bool   VirtualTimer_bDelete(uint8 u8Handle);
0014 void   VirtualTimer_vUpdate(uint32 u32Count,uint32* pu32TimerBitMap);
0015 
0016 #endif //VIRTUAL_TIMER
```