MAC Sample Application
Software Design

Document Number: SWRA200

Texas Instruments, Inc.
San Diego, California USA

Version	Description	Date
0.3	Initial release.	07/28/2006
1.0	Fixes to make MSA more robust	11/16/2006
1.1	Updated for changes to MSA	03/28/2007
1.2	Updated with new OSAL task priority	10/18/2007
1.3	Updated for enhanced beacon request	09/19/2011
1.4	Updated for MAC security	09/22/2011
1.5	Added MAC_CbackQueryRetransmit()	02/26/2015
1.6	Added MSA_NvRestore and MSA_NvReset	03/06/2015

TABLE OF CONTENTS

1. Introduction

1.1 Scope

This document describes the software design of the MAC Sample Application.

1.2 Acronyms

API	Application Programming Interface.
BO	Beacon Order
Coordinator	A full function device that accepts associations and transmits beacons.
CSMA	Carrier senses multiple accesses.
FFD	Full Function Device.
HAL	Hardware Abstraction Layer.
LQI	Link Quality Indication.
MAC	Medium Access Control.
MSA	Mac Sample Application
OSAL	Operating System Abstraction Layer.
PAN	Personal Area Network.
PAN Coordinator	A coordinator that is the principal coordinator of a PAN.
PIB	PAN Information Base.
RFD	Reduced Function Device.
SO	Superframe Order

2. System Overview

2.1 System Context

This section gives a brief description of the system, describing its general operation and interactions.

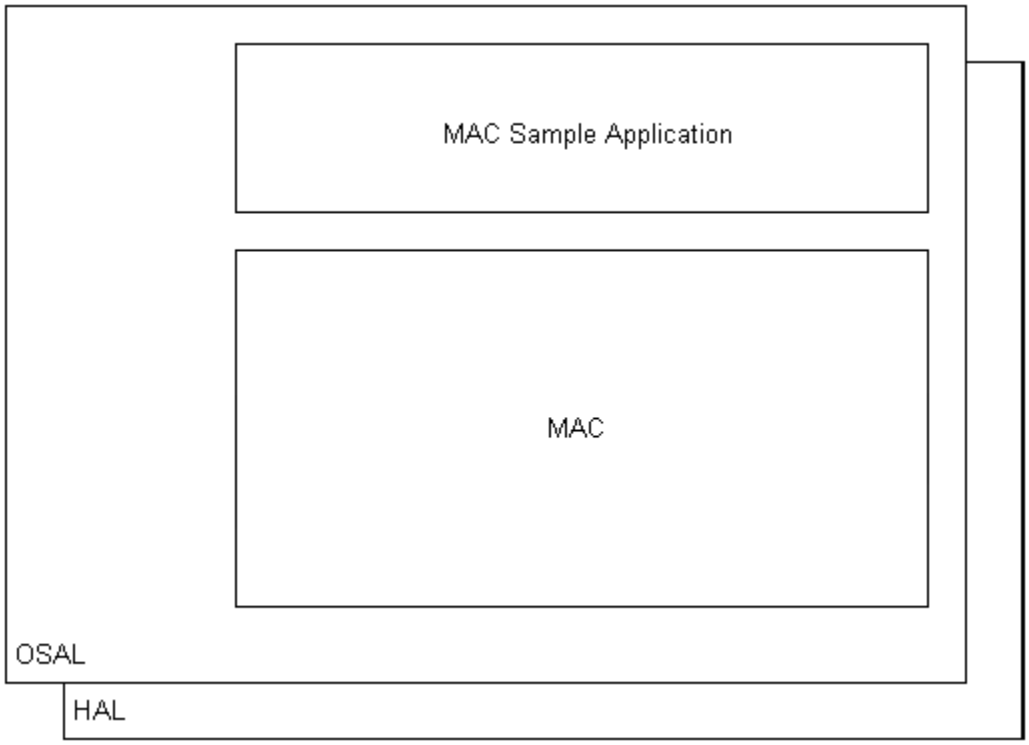

Figure 1 – Overview of the system

2.1.1 MAC Sample Application

MAC Sample Application contains basic protocols and functions for the user to test and verify the 802.15.4 MAC operation. The application receives commands from either a keypress and performs basic procedure such as:

- Powering up and setting device type: Coordinator or Device (with or without beacon enabled)
- Joining / forming the network
- Association / Disassociation
- Scanning
- Transmitting / Receiving secured or unsecured data
- Beacon support

2.1.2 MAC

The MAC performs the following procedures:

- Processing of OSAL events and messages from API functions, received command frames, transmit frame status, and timers.
- Action functions for MAC procedures such as scan, association, disassociation, network start, pan id conflict, indirect data, polling, and portions of beacon processing.
- PIB management.
- Building and parsing frames.
- Radio management.
- MAC timer management.
- Frame reception and transmission including CSMA.
- Frame acknowledgement handling
- Encryption and decryption.
- Power management.

2.1.3 OSAL

OSAL is an operating system abstraction layer. It provides event handling, message passing, timers, memory allocation, and other services.

2.1.4 HAL

HAL is a hardware abstraction layer. It provides a platform-independent interface to hardware services such as GPIOs, timers, and UARTs.

3. Detailed Design

MAC Sample Application contains 4 major components: main, initialization, system event processing and callback processing.

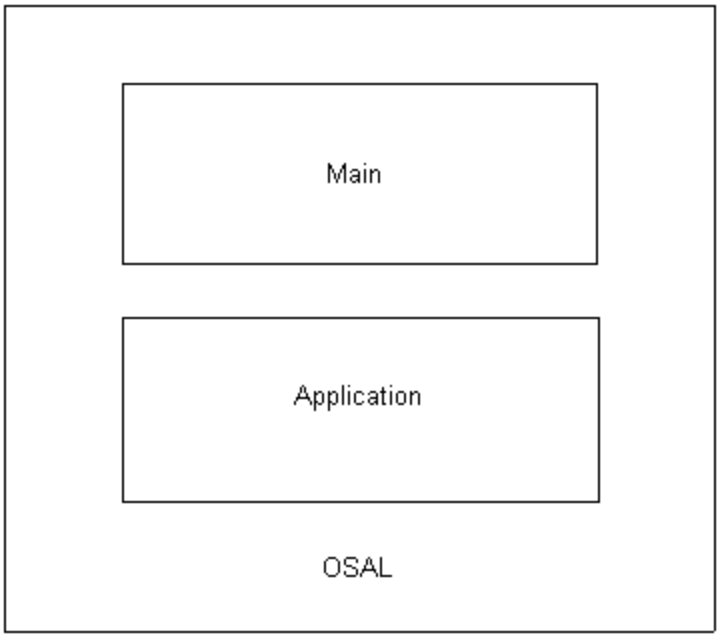

Figure 2 - Sample Application Detailed Design.

3.1 Main

The Main module contains files that initialize and start up the system.

3.1.1 msa_Main.c

3.1.1.1 main ()

Starting point of the system. This is where hardware, MAC and HAL initialization are performed. This is also where OSAL is initialized and tasks are added in the scheduler. OSAL timer and HAL keys are configured and started. HAL drivers must be initialized at this stage for the system to work properly. Other initializations such as radio, interrupt and MAC also need to be performed here. After all the system initializations, osal_start_system() is called to start the tasks.

3.1.1.1.1 Prototype

```
void main ( void );
```


3.1.1.1.2 Parameter Details

None

3.1.1.1.3 Return

None

3.2 Application

The Application module contains files that perform tasks which the application is designed for.

3.2.1 msa_Osal.c

3.2.1.1 osalInitTasks ()

This is where the tasks can be initialized and added. For MSA, there are three main tasks: MAC, HAL and Application task. During the initializing process, init function and event handling function from each task will be initialized and added in the OSAL task scheduler. The order of initializing the task specifies the priority of the task. Task that is initialized first has the highest priority.

3.2.1.2 Prototype

```
void osalInitTasks( void )
```

3.2.1.3 Parameter Details

None

3.2.1.4 Return

None

3.2.2 msa.c

3.2.2.1 MSA_Init ()

This is where the initialization of the application happens. MAC is initialized as Device or Coordinator. The MAC is also reset here. Application registered with the hal_keys so the key callbacks can be properly handled. Task_ID is assigned a value by the osalTaskAdd();

3.2.2.1.1 Prototype

```
void MSA_Init ( uint8 taskId );
```

3.2.2.1.2 Parameter Details

`taskId` – Task ID of the application. OSAL uses this ID to send messages to the application

3.2.2.1.3 Return

None

3.2.2.2 MSA_SecurityInit ()

This routine initializes the MAC security PIBs and enables MAC security for incoming and outgoing data frames.

3.2.2.2.1 Prototype

```
void MSA_SecutityInit ( void );
```

3.2.2.2.2 Parameter Details

None

3.2.2.2.3 Return

None

3.2.2.3 MSA_ProcessEvent ()

This is where the application messages are handled. Application events are distributed to the application through OSAL will be parsed and process in two categories. The SYS_EVENT_MSG and all others that are not SYS_EVENT_MSG. When SYS_EVENT_MSG event is received, all the messages will be retrieved one by one until there is nothing left then the routine will return the remain events to the task manager. All other events will be handled one by one.

3.2.2.3.1 Prototype

```
uint16 MSA_ProcessEvent ( uint8 taskId, uint16 events );
```

3.2.2.3.2 Parameter Details

`taskId` – ID of the application task when it registered with the OSAL

`events` – Events for this task

3.2.2.3.3 Return

16 bit unprocessed events.

3.2.2.4 MAC_CbackEvent ()

This callback function sends MAC events to the application. The application must implement this function. A typical implementation of this function would allocate an OSAL message, copy the event parameters to the message, and send the message to the application's OSAL event handler. This function may be executed from task or interrupt context and therefore must be reentrant.

3.2.2.4.1 Prototype


```
void MAC_CbackEvent ( macCbackEvent_t *pData );
```

3.2.2.4.2 Parameter Details

`pData` – union of callback structures

3.2.2.4.3 Return

None

3.2.2.5 MAC_CbackCheckPending ()

This callback function returns the number of indirect messages pending in the application. When Autopend is not enabled, the higher level application shall implement this call back function to calculate how many indirect data packets are pending for end devices to poll.

3.2.2.5.1 Prototype

```
void MAC_CbackCheckPending ( void );
```

3.2.2.5.2 Parameter Details

`None`

3.2.2.5.3 Return

None

3.2.2.6 MAC_CbackQueryRetransmit ()

This callback function returns whether or not to continue MAC retransmission. This callback function shall be used to stop continuing retransmission for RF4CE. MAC shall call this callback function whenever it finishes transmitting a packet for *macMaxFrameRetries* times.

3.2.2.6.1 Prototype

```
void MAC_CbackQueryRetransmit ( void );
```

3.2.2.6.2 Parameter Details

`None`

3.2.2.6.3 Return

A return value '0x00' will indicate no continuation of retry and a return value '0x01' will indicate to continue retransmission.

3.2.2.7 MSA_CoordinatorStartup ()

This routine will setup the device as a Pan Coordinator. As the Coordinator, MAC_RX_ON_WHEN_IDLE is TRUE in order for the Coordinator to receive incoming messages. Depends on the Beacon Order and Super Frame Order, coordinator will be setup with beacon enabled or not.

3.2.2.7.1 Prototype

```
void MSA_CoordinatorStartup ();
```

3.2.2.7.2 Parameter Details

None

3.2.2.7.3 Return

None

3.2.2.8 MSA_DeviceStartup ()

This routine will setup the device as a Device. Depends on the Beacon Order and Super Frame Order, coordinator will be setup with beacon enabled or not.

3.2.2.8.1 Prototype

```
void MSA_DeviceStartup ();
```

3.2.2.8.2 Parameter Details

None

3.2.2.8.3 Return

None

3.2.2.9 MSA_AssociateReq ()

This routine will fill in the data and call MAC_MlmeAssociateReq().

3.2.2.9.1 Prototype

```
void MSA_AssociateReq (void)
```

3.2.2.9.2 Parameter Details

None

3.2.2.9.3 Return

None

3.2.2.10 MSA_AssociateRsp ()

This routine will fill in the data and call MAC_MlmeAssociateRsp(). The Coordinator will assign the short address to each of the device associates with it. The device then receives this short address and set its short address correspondingly.

3.2.2.10.1 Prototype

```
void MSA_AssociateRsp (macCbackEvent_t* pMsg)
```

3.2.2.10.2 Parameter Details

`pMsg` – Pointer to the macAssociateInd_t structure

3.2.2.10.3 Return

None

3.2.2.11 MSA_McpsDataReq ()

This routine will call MAC_McpsDataReq(). Based on the input parameters, the routine determine if the message is direct or indirect, thus correct parameters are set and the data will be sent out.

3.2.2.11.1 Prototype

```
void  MSA_McpsDataReq ( uint8* data,
                        uint8 dataLength,
                        bool directMsg,
                        uint16 dstShortAddr );
```

3.2.2.11.2 Parameter Details

`data` – pointer to the buffer contains data to be sent

`datalength` – length of the data

`directMsg` – TRUE if data is direct, FALSE if it is indirect

`dstShortAddr` – Short Address of the destination

3.2.2.11.3 Return

None

3.2.2.12 MSA_McpsPollReq ()

This routine will call MAC_MlmePollReq(). If the device is setup as a in-direct messaging device, MAC_MlmePollReq() will have to be call every MSA_WAIT_PERIOD to retrieve the message out of the Coordinator.

3.2.2.12.1 Prototype


```
void MSA_McpsPollReq ( void );
```

3.2.2.12.2 Parameter Details

None

3.2.2.12.3 Return

None

3.2.2.13 MSA_ScanReq ()

This routine will call MAC_MlmeScanReq() to perform an active scan.

3.2.2.13.1 Prototype

```
void MSA_ScanReq ( uint8 scanType, uint8 scanDuration );
```

3.2.2.13.2 Parameter Details

scanType:

Value	Description
0x00	Energy scan
0x01	Active scan
0x02	Passive scan
0x03	Orphan scan
0x05	Enhanced scan

scanDuration: 0-14 – Duration of the scan

3.2.2.13.3 Return

None

3.2.2.14 MSA_SyncReq ()

This routine will call MAC_MlmeSyncReq () to perform a sync request.

3.2.2.14.1 Prototype

```
void MSA_SyncReq ( void );
```

3.2.2.14.2 Parameter Details

None

3.2.2.14.3 Return

None

3.2.2.15 MSA_SecuredDeviceTableUpdate ()

This routine updates the secured device table and key device descriptor handle.

3.2.2.15.1 Prototype

```
void MSA_SecureDeviceTableUpdate ( uint16 panID,
                                   uint16 shortAddr,
                                   sAddrExt_t extAddr,
                                   uint8 *pNumOfSecuredDevices );
```

3.2.2.15.2 Parameter Details

```
panID - Secured network PAN ID
shortAddr - Other device's short address
extAddr - Other device's extended address
pNumOfSecuredDevices - pointer to number of secured devices
```

3.2.2.15.3 Return

None

3.2.2.16 MSA_BeaconPayLoadCheck()

This routine will check the beacon payload for pattern that identifies the device as MSA device. It will return TRUE if the device is MSA device and FALSE otherwise.

3.2.2.16.1 Prototype

```
bool MSA_BeaconPayLoadCheck ( uint8* pSdu );
```

3.2.2.16.2 Parameter Details

`pSdu` – pointer to the buffer contains the received beacon payload

3.2.2.16.3 Return

Return TRUE if the beacon payload is the same as the predefined one. Otherwise, return FALSE.

3.2.2.17 MSA_DataCheck ()

This routine will check the received data with a pre-defined data. It will return TRUE if the data matched and FALSE otherwise.

3.2.2.17.1 Prototype

```
bool MSA_DataCheck ( uint8* data, uint8 dataLength );
```

3.2.2.17.2 Parameter Details

`data` – pointer to the buffer that contains the received data

`dataLength` – length of the received data

3.2.2.17.3 Return

TRUE if the data matches, FALSE otherwise.

3.2.2.18 MSA_HandleKeys ()

This routine receives and process key events for the application.

3.2.2.18.1 Prototype

```
void MSA_HandleKeys ( uint8 keys, uint8 shift );
```

3.2.2.18.2 Parameter Details

`keys` – bit mask of keys received

`shift` – keys are pressed in combination with shift key or not.

3.2.2.18.3 Return

None.

3.2.2.19 MSA_NvRestore ()

NV_RESTORE flag needs to be defined to include this functionality. Immediately after startup, the device checks if the device was on the network previously and if the settings were saved in Non Volatile memory. If so, it reads the settings from Non Volatile memory and restores the device to those settings. For example, when power is suddenly lost and comes back, if the feature is enabled by defining the compiler flag NV_RESTORE, the device will get restored to the settings before power loss.

3.2.2.19.1 Prototype

```
void MSA_NvRestore(void);
```

3.2.2.19.2 Parameter Details

None.

3.2.2.19.3 Return

None.

3.2.2.20 MSA_NvReset ()

NV_RESTORE flag needs to be defined to include this functionality. When this flag is defined the device stores its network settings in Non Volatile memory and restores it immediately after startup using the MSA_NvRestore() API described above. However, if you want to reset the device and not use the previously networking settings stored in NV, push SW1 before powering the device to use this functionality. MSA_NvReset() API will be called when the SW1 is pushed before powering on the device, and it resets the device to out of box settings and clears data stored in Non Volatile memory.

3.2.2.20.1 Prototype

```
void MSA_NvReset(void);
```

3.2.2.20.2 Parameter Details

None.

3.2.2.20.3 Return

None.

4. Operations

4.1 Setting channel and Transmit rate

- The channel can be changed using the following define in msa.h:

  #define MSA_MAC_CHANNEL MAC_CHAN_11

Name
MAC_CHAN_11
MAC_CHAN_12
MAC_CHAN_13
MAC_CHAN_14
MAC_CHAN_15
MAC_CHAN_16
MAC_CHAN_17
MAC_CHAN_18
MAC_CHAN_19
MAC_CHAN_20
MAC_CHAN_21
MAC_CHAN_22
MAC_CHAN_23
MAC_CHAN_24
MAC_CHAN_25
MAC_CHAN_26
MAC_CHAN_27
MAC_CHAN_28

- The period between packets in milliseconds can be changed using the following define in msa.h:

  #define MSA_WAIT_PERIOD 100

4.2 Setup Beacon Order and Superframe Order

- Beacon order and superframe order can be changed using the following defines in msa.h:

  #define MSA_MAC_BEACON_ORDER 15

  #define MSA_MAC_SUPERFRAME_ORDER 15

4.3 Setup Polling or Direct Messaging

- Polling or Direct Messaging can be changed using the following define in msa.h. Set to True if direct messaging is used, False if polling is used.

  #define MSA_DIRECT_MSG_ENABLED TRUE

4.4 Setup MAC data frame security

- Select the “Secure” or “Secure-Banked” project from the IAR workspace to enable MAC_SECURITY services.

4.5 Start the device as a non beacon network Pan Coordinator

- Set MSA_MAC_BEACON_ORDER = 15

- Set MSA_MAC_SUPERFRAME_ORDER = 15
- Turn on the device after downloading the MSA.
- Press SW1 (joystick up).
- The device will scan the channel and if there is no other Pan Coordinator out there, it will become the Pan Coordinator itself.

4.6 Start the device as a beacon network Pan Coordinator

- Set beacon order such that 0 <= MSA_MAC_BEACON_ORDER < 15
- Set superframe order such that 0 <= MSA_MAC_SUPERFRAME_ORDER < 15
- 0 <= MSA_MAC_SUPERFRAME_ORDER <= MAS_MAC_BEACON_ORDER < 15
- Turn on the device after downloading the MSA.
- Press SW1 (joystick up).
- The device will scan the channel and if there is no other Pan Coordinator out there, it will become the Pan Coordinator itself with beacon network enabled. Beacons can be seen on the sniffer at specified interval.

4.7 Start the device as a non beacon network End Device with in-direct messaging support

- Set MSA_MAC_BEACON_ORDER = 15
- Set MSA_MAC_SUPERFRAME_ORDER = 15
- Set MSA_DIRECT_MSG_ENABLED = FALSE
- Turn on the device after downloading the MSA.
- Start a non-beacon network Pan Coordinator (see above).
- Press SW1 (joystick up).
- The device will scan the channel, locates the Pan Coordinator and associates itself to it.
- Also the device will setup itself as an in-direct messaging device and start polling every fixed period.

4.8 Start the device as a non beacon network End Device with direct messaging support

- Set MSA_MAC_BEACON_ORDER = 15
- Set MSA_MAC_SUPERFRAME_ORDER = 15
- Set MSA_DIRECT_MSG_ENABLED = TRUE
- Turn on the device after downloading the MSA.
- Start a non-beacon network Pan Coordinator (see above).
- Press SW1 (joystick up).
- The device will scan the channel, locates the Pan Coordinator and associates itself to it.
- Also the device will setup itself as a direct messaging device.

4.9 Start the device as a beacon network End Device with direct messaging support

- Set beacon order such that 0 <= MSA_MAC_BEACON_ORDER < 15
- Set superframe order such that 0 <= MSA_MAC_SUPERFRAME_ORDER < 15
- 0 <= MSA_MAC_SUPERFRAME_ORDER <= MAS_MAC_BEACON_ORDER < 15
- Set MSA_DIRECT_MSG_ENABLED = TRUE
- Turn on the device after downloading the MSA.
- Start a beacon network Pan Coordinator (see above).
- Press SW1 (joystick up).
- The device will scan the channel, locates the Pan Coordinator and associates itself to it.
- Also the device will setup itself as a direct messaging device with beacon enabled network.

4.10 Transmit packets from the Pan Coordinator to the associated devices

- Press SW2 (joystick right) on the Pan Coordinator.
- The Pan Coordinator starts transmitting a predefined packet to each of the devices in Round Robin manner. Each time a packet is sent out, the Pan Coordinator will blink LED1. Upon receiving the packet, the Device will verify if the received packet is correct or not. If everything is verified, LED3 on the Device will blink. A respond packet will be sent back to the Pan Coordinator.
- Press the SW2 (joystick right) again will stop the transmitting.

4.11 Transmit packets from the Device to the Pan Coordinator

- Press SW2 (joystick right) on the Device
- The Device starts transmitting a predefined packet to the Pan Coordinator. Each time a packet is sent out, the Device will blink LED1. Upon receiving the packet, the Pan Coordinator will verify if the received packet is correct or not. If everything is verified, LED3 on the Pan Coordinator will blink.
- Press the SW2 (joystick right) again will stop the transmitting.