## Summary

The MailBox API is designed to be a very simple programming interface for developers of ZigBee applications. It may be used for the rapid application development of ZigBee products with minimal prior knowledge of ZigBee or radio communications.

MailBox is designed for interoperability and ease-of-use. In particular, it offers compatibility with existing MailBox products and avoids the need to apply for ZigBee Alliance membership. The protocol ensures data is received at the destination error-free and in the sequence in which it was transmitted, but does not attempt to define the content of the data. It is the applications responsibility, however, to know what to do with it when it arrives.

The MailBox layer communicates using the MailBox ZigBee profile which is being submitted to the ZigBee organization for registration. Until registration is complete, 'placeholder' profile IDs have been used which may be subject to change.

In addition, the Mailbox API provides generic communications for applications without a dedicated ZigBee profile, for example:

- application wishing to take advantage of existing MailBox devices such as serial and USB adapters
- where communications are to be bridged to non-ZigBee media such as TCP/IP or RS485
- where the market size does not merit development of a dedicated ZigBee profile

A network-wide sleep feature is also provided which allows a network or parts of the network to agree to allow all devices to sleep until a later wakeup time.

The MailBox layer communicates using the MailBox ZigBee profile which has been submitted to the ZigBee organization for registration. Until registration is complete, 'placeholder' profile IDs have been used which may be subject to change. The profile is public. FlexiPanel Ltd retains the right to define the profile, but attaches no conditions or fees for to its adoption and use.

## Compatibility

The API is fully compatible with all MailBox products, including:

- *Pixie Gateway*
- *PixieDARC*
- *UZBee Gateway (when available)*
- *COMdongle (when available)*

## Features

MailBox profile incorporates the following features:

- Integrated Microchip ZigBee stack
- conveyance of arbitrary data payloads in sequence at up to approx 19.2kbit/s
- addressed or broadcast (bus) messaging
- function-specific functional address system independent of underlying ZigBee addresses
- payload acknowledgement and failure management
- network-wide sleep and wakeup
- sleepy end device sleep & wakeup
- application specific messages, *e.g.* for modem status signals and application reflashing

## *Contents*

## *MailBox Overview*

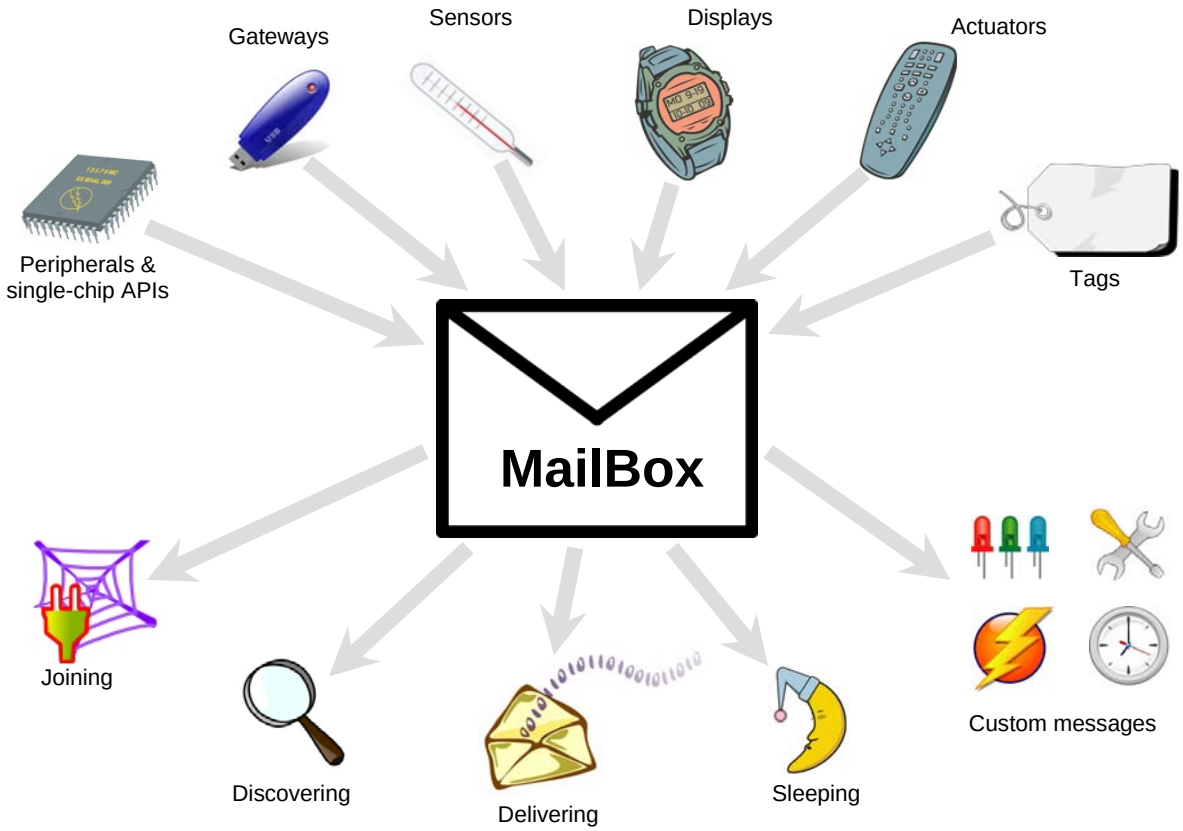
MailBox API is used with the Pixie and Pixie Lite 2.4GHz IEEE 802.15.4 / ZigBee transceiver modules.

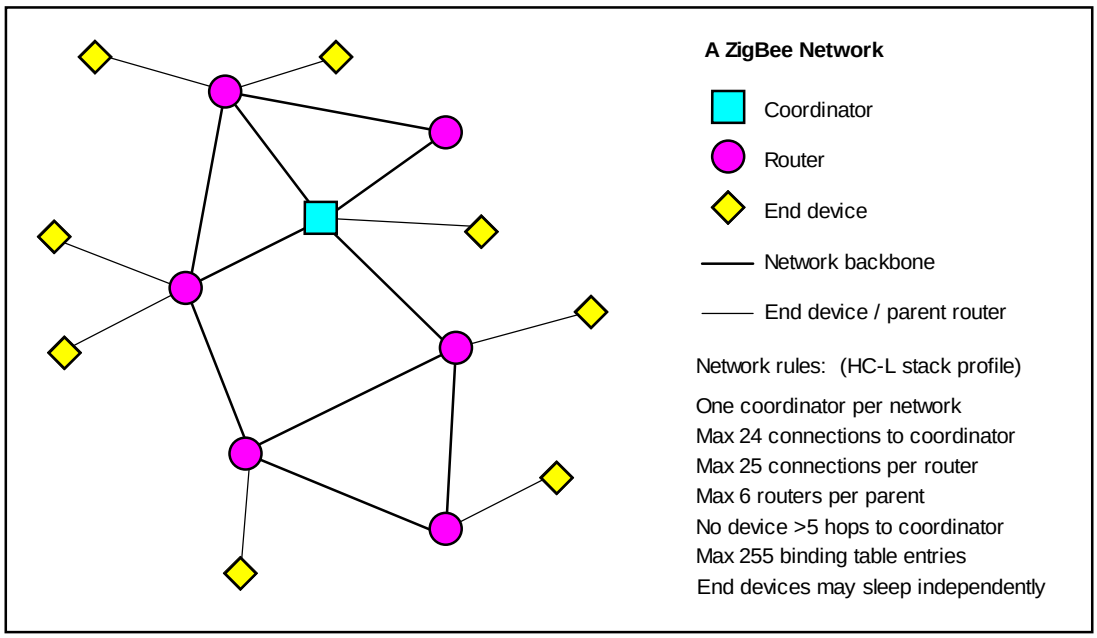
The IEEE 804.15.4 protocol provides services for transceiver devices to discover each other and then exchange packets of data in a reliable, error-free manner. Above it, the ZigBee layer allows multi-hop communications across mesh networks. At the top, the MailBox layer provides generic data communications without the need to understand how the lower layers work.

MailBox is designed for interoperability and ease-of-use. In particular, it offers compatibility with existing MailBox products and avoids the need to apply for ZigBee Alliance membership. The protocol ensures data is received at the destination error-free and in the sequence in which it was transmitted, but does not attempt to define the content of the data. It is the applications responsibility, however, to know what to do with it when it arrives.

The MailBox layer communicates using the MailBox ZigBee profile which is being submitted to the ZigBee organization for registration. Until registration is complete, 'placeholder' profile IDs have been used which may be subject to change.

The MailBox layer adds two important features to the ZigBee communications protocol. The first is packet sequencing, which ensures that packets arrive in sequence, without loss or repetition. The second is functional clusters, which address devices according to the function they are supposed to perform, rather than their MAC or ZigBee short addresses. In addition, it provides guidelines for network-wide sleep and power outage recovery, which are not defined by the lower layers.






## *Device Types*

Several device types are employed in a ZigBee network. The *coordinator* is the device which dictates network-wide rules such as operating frequency. There must be one coordinator in a network and it is the first member of the network. The network is then built by joining new devices on to existing devices in the network.

*Routers* are devices which can forward messages on behalf of other devices. They form the basis of the multi-hop messaging system. Unless the entire network has agreed to sleep simultaneously, routers and the coordinator shall be always on and listening for messages.

*End devices* cannot route messages on behalf of other devices, and they cannot have admit new devices into the network. *Fast end devices* keep their radios on all the time. *Sleepy end devices* can spend most of their time asleep; when they wake, they must check with their parent router to see if there are any messages waiting for them.

With any ZigBee network, there are rules about how many child devices a router may have, how far a device may be from the coordinator, etc. These rules are referred to collectively as the stack profile. Devices with different stack profiles are not compatible. Since MailBox Gateway is intended to piggyback on any type of ZigBee network, it can be applied to any stack profile. The default implementation is the Home Controls profile, whose rules are indicated in the figure above.

## *Functional Clusters*

Each device maintains a list of application-defined functional clusters which indicate the services it provides.

ZigBee devices use an 8-byte globally unique address when joining a network. Once joined, they are assigned a 2-byte address which is unique within the network. Since neither address is under the control of the application, this doesn't help the application know who to talk to. Functional clusters help perform this task.

MailBox devices use two-byte functional clusters to identify themselves according to the functions they perform. All devices support cluster 0x0000 and they may support any number of other clusters. In addition to sending a message to a specific device short address, a message may be broadcast to all devices which support a particular cluster. Function-specific device discovery allows network devices to bind together without user intervention.

The interpretation of cluster values 0x0001-0x00FD is application specific. For example, in a sensor network, all sensors might support clusters 0x0000 and 0x0001. Data gathering gateways might support clusters 0x0000 and 0x0002. A gateway could send a message to all sensors by broadcasting to cluster 0x0001. A sensor can search for a data gateway using device discovery and then direct MailBox packets directly to it.

It is anticipated that 2-byte clusters will be supported in ZigBee 1.1. At that time, the range of application specific clusters will be extended to 0x0001-0xFFFD.

Cluster 0xFFFE shall be the Redirect Address signifier. If 0xFFFE is specified as a destination address, the actual destination shall be that specified by the most recently received Redirect Address message. This allows reduced function devices such as sensors to be instructed remotely as to where to send their data.

Cluster 0xFFFF signifies a null cluster; messages addressed to the null cluster will be discarded without transmission.

## *State Machine Architecture*

Most ZigBee MailBox functions are implemented using the function *MailBoxTasks()*. To perform a specific action such as joining a network, the *MailBoxState* variable is set to a request value such as *MBS_Join_Request*. *MailBoxTasks ()* must then be called repeatedly until the *MailBoxState* changes to the associated confirm value such as *MBS_Join_Confirm*. On occasions where the MailBox must provide the application with unprompted information, for example if it received data, it will set the *MailBoxState* to an indication value such as *MBS_Data_Indication*.

## *Network Joining*

A device can't do much until it has established communication with the network. Therefore after initializing, a device should attempt to join the network using the *MBS_Join_Request* message. If this is the first time it has joined the network, the device which will be its parent should be place in the permit join state using the *MBS_Permit_Join_Request* command.

## *Device and Service Discovery*

A variety of mechanisms are provided for working out what other devices are on the network and who to send messages to. Device discovery (*MBS_Device_Discovery_Request*) is used to search for devices supporting specific functional clusters. Service discovery (*MBS_Device_Discovery_Request*) is used to interrogate a specific device. Present messages (*MBS_Present_Request*) announce to other devices that this device is present on the network. Redirect requests (*MBS_Redirect_Request*) instruct other devices where to send their messages.

## *Data Delivery*

Payloads of up to 64 bytes of data can be sent at a time in a packet using *MBS_Data_Request*. Packets can be broadcast to all devices supporting a functional cluster, or unicast to a specific short address. They can be unacknowledged, or acknowledged.

- *Unacknowledged*. Frame sequence numbers are used to determine if a frame has arrived out of sequence or has been lost. Loss of sequence is reported to the receiving application but no other action is taken. No acknowledge is sent to the source. Broadcast communication is permitted.

- *Acknowledged*. The destination will formally accept a frame before the source transmits the next frame. Broadcast transmissions are not permitted. If a frame does not reach the destination, it can be repeated, thus guaranteeing uninterrupted, sequenced data. Broadcast communication is currently not permitted but it will be in future.

## *Sleep Management*

Two sleep modes are provided. Device sleep (MBS_Device_Sleep_Request) permits sleepy end devices to sleep at any time. Network sleep (MBS_Network_Sleep_Request) allows the entire network to sleep simultaneously.

Sleep is entirely under the control of the host. On receipt of a network sleep message, the application can device when and whether to sleep. A sleepy end device may sleep indefinitely, but if it needs to poll its parent for messages, it must be woken intermittently.

Three device settings affect sleep behavior:

The *PersistenceTime* is the length of time that a router will hold a message for a sleeping child.

The *PollRate* is the frequency with which a sleepy end device will poll its parent for messages while it is awake. (While asleep, it cannot poll at all.) If *PollRate* is set to zero, the parent will only be polled on wakeup.

The *SleepWakeMode* device setting governs the operation of the Sleep/Wake pin. In the default mode (+DSWR=00 command), the device is placed in sleep mode using the +MSMR command and woken by a change of state of the Sleep/Wake pin. In this mode, the Sleep/Wake pin is tied to the RxD input so that sending a character will wake the device. (The character will otherwise be ignored. A null character is recommended.)

In the alternate Sleep/Wake pin mode (+DSWR=01 command), the device will sleep when the Sleep/Wake pin is high and wake when the pin is low.

In both modes, the RTS pin will be high during sleep and during waking. When waking is complete, RTS will go low and a +DRYI message will be generated.

## *Custom Messages*

128 custom messages (MBS_Custom_Request) are available for application specific use.

## *MIB Attributes*

Various API variables may be set to control MailBox behavior. These are collectively called MailBox Information Base (MIB) attributes and may be set with the +DSxR command.

## *Application ID*

The use of functional clusters alone does not allow for the fact that two MailBox applications may need to coexist on a network. Specifically, one manufacturer may associate different meanings to different clusters.

To allow for this, a 4-byte application ID is associated with each manufacturer's interpretation of clusters. This application ID is included with all data transactions and is also available for direct querying to verify a device prior to unicast transmission. It also allows manufacturer specific messages to be transmitted.

In order for application IDs to be unique, FlexiPanel will allocate on request a 3-byte MailBox Unique Identifier (MUI). This is free of charge.

Application IDs shall be allocated as follows:

| | |
|---|---|
| Bytes 0-2: | Manufacturer's MUI number |
| Byte 3: | Assigned by manufacturer |

The 'free-for-all' MUI 00:00:00 may be used by any application that can ensure that it is the only MailBox application on the network.

## *Endpoints*

ZigBee endpoints allow several applications to share one ZigBee radio. The MailBox protocol uses one ZigBee endpoint. By default it is 0x10, but this may be changed for the convenience of other applications. This may be any endpoint in the allowed range 0x01-0xF0. The same endpoint must be used for all devices employing that Application ID.

## *Notation, Byte & Bit order*

All numbers in this documentation are in decimal unless prefixed with 0x, in which case they are hexadecimal. Index counting starts at zero, so the first byte of a message is byte zero.

Multi-byte data is transmitted least-significant byte first ('little-endian'), as is standard in the ZigBee specification.

## *Symbol Periods*

Several time periods are expressed in units of Symbol Periods. A symbol period is 1/62,500 second. 0x10000 symbol periods equates to just over one second.

## *Copy Protection*

To protect against copying, if the MailBox API firmware is run on any hardware except FlexiPanel Pixie and Pixie Lite products, it will cease to function after approximately two minutes.

## *Release notes, version 0B40011510352106 1006pt*

In this release, security is not supported. MailBox profile ID 0xC1EE has been assigned

## *Bibliography*

***IEEE 802.15.4 specification***, downloadable from *www.ieee.org*.

***The MailBox Profile for ZigBee***, downloadable from *www.flexipanel.org*.

***ZigBee for Applications Developers***, white paper downloadable from *www.flexipanel.com*.

***ZigBee Specification***, downloadable from *www.zigbee.org*.

### *Firmware Development Guide*

The Microchip Technology MPLAB development environment and C18 compiler will be needed to develop MailBox API applications. A debugger such as the Microchip Technology ICD2 is recommended. Please refer to Microchip Technology documentation for full details on how to develop applications for PIC microprocessors.

A MailBox application project include the following files:

| | |
|---|---|
| `MailBox.h` | Header file for MailBox library functions and data. |
| `MailBoxAPI-`*`HSSD`*`.lib` | MailBox library. *HSSD* signifies the different library versions: *H* indicates hardware, being Pixie (H) or Pixie Lite (L). *SS* indicates stack profile, for example Home Controls (HC). *D* indicates device type, being Coordinator (C), Router (R), Fast End Device (F) or Sleepy End Device (S). |
| `MbxLink`*`xxxx`*`.lkr` | Required linker script. *XXXX* is 4620 for Pixie and 2520 for Pixie Lite. |

You will also need to provide code for your main application program and also specify the configuration bits you require. The following memory model settings should be specified:

- Small code model
- Large data model
- Single-bank model

The oscillator configuration must be set for a 16MHz clock. If using the internal oscillator block, set the oscillator setting to Internal RC and include the following lines in your startup code:

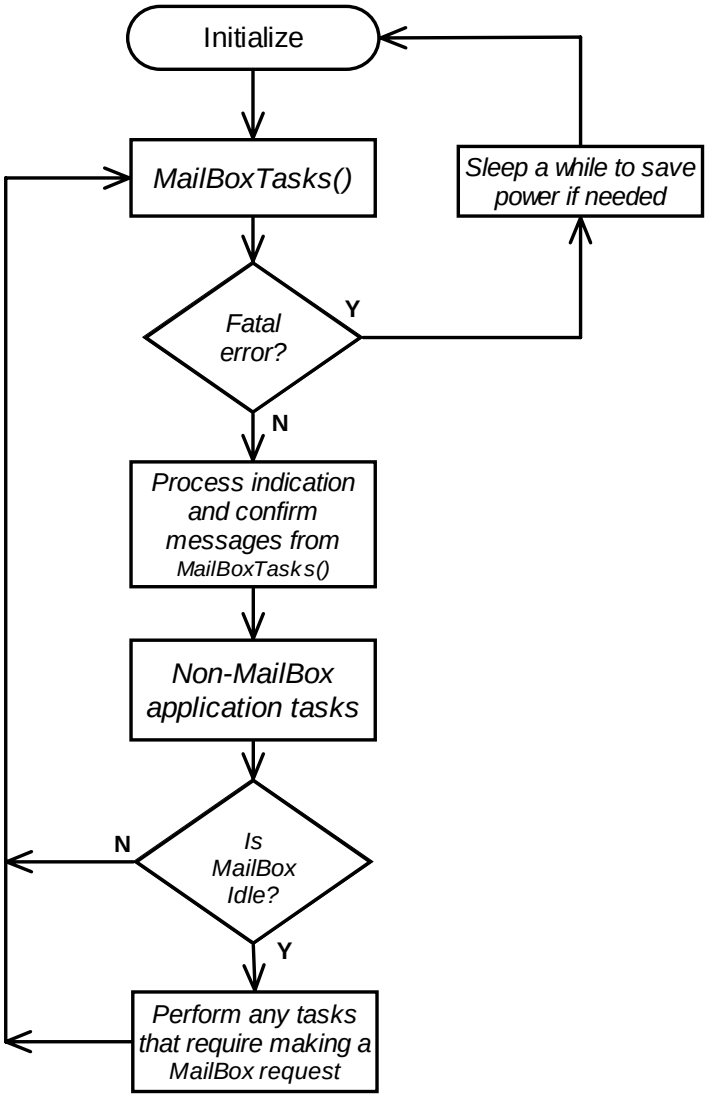
```
OSCCONbits.IRCF1 = 1;      // changes to <IDCF2:IDCF0>=110 = 4MHz
OSCTUNEbits.PLLEN = 1;     // PLL 4MHz -> 16MHz
Delay1KTCYx( 100 );        // allow 25ms for clock to settle
```

### *Application Development Example*

The following example source code is for the PixieDARC data acquisition and remote control device. Refer to its separate documentation if you are unfamiliar with it. It accepts instructions via the MailBox Profile in order to set digital outputs and also to read digital and analog inputs. Six different sets of project files are in the development kit, one for each of the different possible builds:

`PXDC` – Pixie DARC Coordinator
`PXDR` – Pixie DARC Router
`PXDF` – Pixie DARC Fast End Device
`PXDS` – Pixie DARC Sleepy End Device
`PLDF` – Pixie Lite DARC Fast End Device
`PLDS` – Pixie Lite DARC Sleepy End Device

The file `Config.c` specifies the configuration bits required. The file `PixieDARC.c` contains the application program. It has the following basic structure:



The `PixieDARC.c` code is reproduced below with annotations:

```
//********************************************************************
// FileName:        Pixie DARC.c
//                  Sample Mailbox application - Data Acquisition and Remote control
//
//********************************************************************

// Basic process in this application:
// 1. If I'm started up with the join button down, erase network data
// 2. If I'm not joined to anything, attempt to join
// 3. If join fails, or any errors occur, reset
// 4. If join succeeds, send present announcement
// 5. If join button pressed, permit joining (routers & coordinator only)
// 6. On receipt of instruction from another device, process and respond to that device

//********************************************************************

// Compile switches - you can either uncomment here or in the build settings
//#define DARC_COORDINATOR
//#define DARC_ROUTER
//#define DARC_FAST_END
//#define DARC_SLEEPY_END

//#define PIXIE
//#define PIXIE_LITE

#if (!defined(PIXIE) && !defined(PIXIE_LITE))
#error "PIXIE_xxx must be defined"
#endif

#if (!defined(DARC_COORDINATOR) && !defined(DARC_ROUTER) && !defined(DARC_FAST_END) &&
!defined(DARC_SLEEPY_END))
#error "DARC_xxx must be defined"
#endif


//********************************************************************

// includes

#include <p18cxxx.h>

// MailBox definitions
#include "MailBox.h"
#include <delays.h>       // for delay functions
#include <string.h>         // for memcpy-type functions

//********************************************************************

// DARC commands
#define SET_IO_VAL        0x01
#define GET_IO_VAL        0x02
#define GET_AN_VAL        0x03
#define SET_AN_CHANS  0x04
#define COMMAND_STREAM    0x05

#define IO_RA0    0x00
#define IO_RA1    0x01
#define IO_RA2    0x02
#define IO_RA3    0x03
#define IO_RB4    0x14
#define IO_RB5    0x15
#define IO_RB6    0x16
#define IO_RC6    0x26
#define IO_RC7    0x27
#define IO_RE0    0x40
#define IO_RE1    0x41
#define IO_RE2    0x42
#define IO_AN0    0x80
#define IO_AN1    0x81
#define IO_AN2    0x82
#define IO_AN3    0x83
#define IO_AN5    0xC0
#define IO_AN6    0xC1
#define IO_AN7    0xC2

#define DARC_STATUS_SUCCESS                         0x00
#define DARC_SYNTAX_ERROR     0x01
#define DARC_STATUS_OUT_OF_RANGE          0x02
#define DARC_STATUS_IO_NOT_POSSIBLE                 0x03

//*****************************************************************************/
// DARC function prototypes & static variables

void SetIOVal( BYTE IOval );
void GetIOVal( BYTE IOval );
void GetANVal( BYTE IOval );
void SetANChannels( void );
void SetStream( void );

BYTE NumANchannels = 0;
BYTE StreamChannel = 0;
BYTE StreamRate = 0;
```

**Device type settings must match the MailBox library you link to.**

**Definitions relating to the DARC communications messages.**

**Declarations relating to DARC functions.**

```
#ifdef DARC_SLEEPY_END
BYTE StreamCountDown ;
#else
TICK LastStream;
#endif

//******************************************************************************/
// semaphores

typedef union _DARC_FLAGS
{
    WORD Val;
    struct _Bits
    {
         BYTE ParseError : 1;
         BYTE Joined : 1;
         BYTE PresenceAnnounced : 1;
         BYTE GoToSleep : 1;
         BYTE Erase : 1;
         BYTE ProcessInboundMessage : 1;
         BYTE SendStreamValue : 1;
         BYTE StayAwakeToStream : 1;
         BYTE Announce: 1;
    } Bits;
}  DARC_FLAGS;

DARC_FLAGS DARCFlags;

//-----------------------------------------------------------------------------------------------

// Cache for incoming data
#define  MAX_CMD_LEN 8
BYTE        MessageCache[ MAX_CMD_LEN ];
BYTE     MessageLen;
WORD_VAL ReplyAddress;

//-----------------------------------------------------------------------------------------------

// Nonvolatile variables.  Note that writing nonvolatile memory requires erasing a block
// of memory 0x40 bytes long, so we will ensure sure the rest of the block is not program
// memory by declaring filler before and after.  This is very ineccifient, there are better ways
to do it

rom BYTE DARCfiller1[0x40];

// Check MAC address is set.
rom BYTE nvMACset = 0;            // Nonzero if MAC address has been set

rom BYTE DARCfiller2[0x40];

//-----------------------------------------------------------------------------------------------

// UART is used for setting MAC address

#define CLOCK_FREQ 16000000
#define BAUD_RATE 19200
#define USART_USE_BRGH_HIGH
#if defined(USART_USE_BRGH_LOW)
    #define SPBRG_VAL   ( ((CLOCK_FREQ/BAUD_RATE)/64) - 1)
#else
    #define SPBRG_VAL   ( ((CLOCK_FREQ/BAUD_RATE)/16) - 1)
#endif

#if SPBRG_VAL > 255
    #error "Calculated SPBRG value is out of range for currnet CLOCK_FREQ."
#endif

#if defined(USART_USE_BRGH_HIGH)
#define TXSTA_VAL 0x24
#else
#define TXSTA_VAL 0x20
#endif

//-----------------------------------------------------------------------------

// I/O pin definitions

#define STATUS_LED                        PORTAbits.RA4
#define TRIS_STATUS_LED                   TRISAbits.TRISA4

    #define BIND_SWITCH             PORTBbits.RB7
    #define TRIS_BIND_SWITCH        TRISBbits.TRISB7

//******************************************************************************/

void PutROMString(ROM char* str);
void NetworkFailed( void );

// main program

void main(void)
{
    // initialize I/O.
    DARCFlags.Val = 0x0000;
    ADCON1 = 0x0F;            // Make PORTA digital I/O.
    TRIS_BIND_SWITCH = 1; // Bind switch is input
```

**Semaphore values. These record the state of the device and are used to note jobs that require doing when the MailBox state is idle.**

**The UART is used only to initialize the MAC address.**

```
    TRIS_STATUS_LED = 0;  // Status LED is output
    STATUS_LED = 1;           // Status LED is on until join is successful

    // Test to see if Join was presed on power-up, indicating a request for erase
    if (!BIND_SWITCH)
    DARCFlags.Bits.Erase = 1;

    // set up internal oscillator for 16MHz operation
#ifdef OSC_INTERNAL
    OSCCONbits.IRCF1 = 1;     // assuming <IDCF2:IDCF0>=100 on startup, changes to
<IDCF2:IDCF0>=110 = 4MHz
    OSCTUNEbits.PLLEN = 1;  // PLL 4MHz -> 16MHz
    Delay1KTCYx( 100 );  // allow 25ms for clock to settle
#endif

    // enable watchdog timer
    ClrWdt();
    WDTCONbits.SWDTEN = 1;

    // Initialize mailbox
    MailBoxInit();

   // If MAC not set, prompt and ask for it now
    if (!nvMACset)
    {
        BYTE MACAddr[8];
        BYTE* pMACAddr = &MACAddr[8];
        BYTE cTemp;

        // set up uart
        TXSTA = TXSTA_VAL;
        RCSTA = 0x90;
    SPBRG = SPBRG_VAL;

        // ask for MAC address in hex and parse it.  Note there is not very much error correction
        PutROMString( (rom char *) "MAC->" );
        do {
            pMACAddr--;
            while (PIR1bits.RCIF==0) {ClrWdt();}
            cTemp = RCREG - '0';
            if (cTemp > 0x09) cTemp -= 0x07; // A-F
            if (cTemp > 0x0F) cTemp -= 0x20; // a-f
            *pMACAddr = cTemp << 4;
            while (PIR1bits.RCIF==0) {ClrWdt();}
            cTemp = RCREG - '0';
            if (cTemp > 0x09) cTemp -= 0x07; // A-F
            if (cTemp > 0x0F) cTemp -= 0x20; // a-f
            *pMACAddr += cTemp ;
        }
        while (pMACAddr > MACAddr);

        // Write nonvolatile data
        PutMACAddress( pMACAddr );
        cTemp = 1;
        NVMWrite(&nvMACset, &cTemp, 1);

        // Confirm and reset
        PutROMString( (rom char *) "\r\nOK\r\n" );
        Reset();
    }

    while (1)
    {
        // perform mailbox tasks
        ClrWdt();
       MailBoxTasks( );

        // look at all possible return states and decide what to do
        switch (MailBoxState)
        {
            // Some confirms return with an error condition which should be checked for error
            case MBS_Data_Confirm:
                if (pMailBoxParam->MBS_Data_Confirm.Status != 0) NetworkFailed();
#ifdef DARC_SLEEPY_END
                if (DARCFlags.Bits.StayAwakeToStream) DARCFlags.Bits.StayAwakeToStream = 0;
#endif
                break;

            case MBS_Permit_Join_Confirm:
                // check join was success
                if (pMailBoxParam->MBS_Data_Confirm.Status != 0) NetworkFailed();
                break;

            // If an error happens, reset and rejoin
            case MBS_Leave_Indication:
            case MBS_Error:
            case MBS_Sync_Loss_Indication:
                NetworkFailed();                  // does not return

            case MBS_Join_Confirm:
                // check join was success
                if (pMailBoxParam->MBS_Data_Confirm.Status != 0) NetworkFailed();

                // Join succeeded
                DARCFlags.Bits.Joined = 1;
                DARCFlags.Bits.Announce = 1;
```

**Set the oscillator up on initialization.**

**Initialize MailBox.**

**Ensure the MAC address is set.**

**Main program loop, with MailBoxTasks() at the top.**

**Check no errors reported.**

**If we successfully joined, announce our presence to all MailBox devices.**

```
                    break;

                case MBS_Present_Confirm:
                    // check was success
                    if (pMailBoxParam->MBS_Data_Confirm.Status != 0) NetworkFailed();
                    DARCFlags.Bits.PresenceAnnounced = 1;
                    STATUS_LED = 0;                   //      Initialization is complete
                    break;

                case MBS_Permit_Join_Indication:
                    // a device joined, clear the LED to indicate it.
                    if (pMailBoxParam->MBS_Data_Confirm.Status != 0) NetworkFailed();
                    STATUS_LED = 0;
                    break;

                case MBS_Data_Indication:
                    // Data is acceptable unless we have received a m
                    // it came with a serious health warning
                    if ( pMailBoxParam->MBS_Data_Indication.Status!=D
                         pMailBoxParam->MBS_Data_Indication.Status!=D
                         pMailBoxParam->MBS_Data_Indication.DataPayloadLen <= MAX_CMD_LEN )
                         {
                            // we received a message so remember it and who it was from
                            ReplyAddress = pMailBoxParam->MBS_Data_Indication.SourceAddr;
                            MessageLen = pMailBoxParam->MBS_Data_Indication.DataPayloadLen;
                            memcpy( MessageCache, (void*) pMailBoxParam->MBS_Data_Indication.pRxData,
MessageLen );
                            DARCFlags.Bits.ProcessInboundMessage = 1;
                        }
                    break;

#ifdef DARC_SLEEPY_END
                case MBS_Device_Sleep_Confirm:
                    // Mailbox is happy to sleep
                    if (pMailBoxParam->MBS_Data_Confirm.Status == 0)
                    {
                        DARCFlags.Bits.GoToSleep = 1;
                    }
                    break;
#endif

                    // no action required for other exit conditions
                    default: break;
            }

            // Pump the mailbox tasks until idle
            if (MailBoxState!=MBS_Idle)  continue;

            // Any application tasks that do need to modify MailBoxState from here on

            // If I haven't joined yet, then join
            if (!DARCFlags.Bits.Joined)
            {
                pMailBoxParam->MBS_Join_Request.Flags.Erase = ( DARCFlags.Bits.Erase ? 0x01 : 0x00 );
                MailBoxState = MBS_Join_Request;
                continue;
            }

            // If I've joined but not announced presence, do so now
            if (DARCFlags.Bits.Announce)
            {
                DARCFlags.Bits.Announce = 0;
                MailBoxState = MBS_Present_Request;
                pMailBoxParam->MBS_Present_Request.Flags.Broadcast = 1;
                pMailBoxParam->MBS_Present_Request.Destination.Val = 0x0000;  // Broadcast to all
nodes
                continue;
            }

#ifdef DARC_SLEEPY_END
            // If ready to sleep, then sleep
            if (DARCFlags.Bits.GoToSleep)
            {
                DARCFlags.Bits.GoToSleep = 0;

                // Perform any pre-sleep shutdown here - set to wake up on watchdog timeout
            ClrWdt();

                // sleep
                Sleep();
                Nop();

                // Perform any post-wake operations - clear watchdog timer
            ClrWdt();

                // restart MailBox stack

                MailBoxState = MBS_Device_Wake_Request;

                // stream data if needed
                if (StreamRate)
                {
                    StreamCountDown--;
                    if (!StreamCountDown)
                    {
                        DARCFlags.Bits.SendStreamValue = 1;
```

**If a message is received, save it and process it when idle.**

**If the MailBox allows us to sleep, start sleeping.**

**Non-Mailbox application tasks can be executed here, then return to MailBoxTasks() if MailBox is not Idle.**

**Join network**

**Announce presence**

**Sleep**

**Stream Data on wakeup**

```
                    StreamCountDown = StreamRate;
                }
            }
            continue;
        }
#endif

        // If permit join pressed, permit joining
#if (defined(DARC_COORDINATOR) || defined(DARC_COORDINATOR))
        // Note, this command should only be used for routers and
        if (STATUS_LED==0 && BIND_SWITCH==0)  // If LED does not i
joining and switch is on (active low)
        {
            // permit join indefinitely; no cancel permit join
            // code is provided if it needs to be cancelled, reset
            pMailBoxParam->MBS_Permit_Join_Request.PermitDuration = 0xFF;
            MailBoxState = MBS_Permit_Join_Request;
            STATUS_LED = 1;
            continue;
        }
#endif

        // process a command, if one received
        if (DARCFlags.Bits.ProcessInboundMessage)
        {
            DARCFlags.Bits.ProcessInboundMessage = 0;

            // set up default response
            pTxData[0] = DARC_SYNTAX_ERROR;
            pMailBoxParam->MBS_Data_Request.DataPayloadLen = 0x01;

            switch (MessageCache[0])
            {
                case SET_IO_VAL:
                    if (MessageLen==0x03) SetIOVal(MessageCache[1]);
                    break;
                case GET_IO_VAL:
                    if (MessageLen==0x02) GetIOVal(MessageCache[1]
                    break;
                case GET_AN_VAL:
                    if (MessageLen==0x02) GetANVal(MessageCache[1]
                    break;
                case SET_AN_CHANS:
                    if (MessageLen==0x02) SetANChannels();
                    break;
                case COMMAND_STREAM:
                    if (MessageLen==0x03) SetStream();
                    break;
            }

            // compile reply
            pMailBoxParam->MBS_Data_Request.Flags.IsBroadcast = 0;
            pMailBoxParam->MBS_Data_Request.Flags.IsAcknowledge = 0;
            pMailBoxParam->MBS_Data_Request.Flags.IsRetry = 0;
            pMailBoxParam->MBS_Data_Request.Destination.Val = ReplyAddress.Val;
            MailBoxState = MBS_Data_Request;
            continue;
        }


    // Streaming, for non-sleepy devices
#ifndef DARC_SLEEPY_END
    if (StreamRate)
    {
        TICK tNow = TickGet();
        if (TickGetDiff( tNow, LastStream ) > (((DWORD) StreamRate) << 16))
        {
            LastStream = tNow;
            DARCFlags.Bits.SendStreamValue = 1;
        }
    }
#endif

    if (DARCFlags.Bits.SendStreamValue)
    {
        DARCFlags.Bits.SendStreamValue = 0;
        DARCFlags.Bits.StayAwakeToStream = 1;

        // compile reply
        if (StreamChannel>=IO_AN0)
            GetANVal(StreamChannel);
        else
            GetIOVal(StreamChannel);
        pMailBoxParam->MBS_Data_Request.Flags.IsBroadcast = 0;
        pMailBoxParam->MBS_Data_Request.Flags.IsAcknowledge = 0;
        pMailBoxParam->MBS_Data_Request.Flags.IsRetry = 0;
        pMailBoxParam->MBS_Data_Request.Destination.Val = ReplyAddress.Val;
        MailBoxState = MBS_Data_Request;
    }

    // if I got this far and I'm sleepy, joined, PresenceAnnounced
sleeping
#if defined(DARC_SLEEPY_END)
    if (!DARCFlags.Bits.StayAwakeToStream && DARCFlags.Bits.Presen
        MailBoxState = MBS_Device_Sleep_Request;
#endif
```

**Permit joining if Bind pressed and device is Coordinator or Router**

**If a remote device has sent a command, process it and prepare a response**

**Set and Get functions can be seen in the development source code**

**Stream Data if not sleepy**

**If sleepy and all tasks done, ask MailBox if we can sleep**

```
    }  // end of while (1) loop
}


/*********************************************************************************
User Interrupt Handlers

The stack uses some interrupts at low priority for its internal processing.  Once it is done
checking for its interrupts, the stack calls UserInterruptHandler function to allow for any
additional interrupt processing.  Interrupts should not be disabled unless the sleep state
has been entered.

PriorityUserInterruptHandler must be for very fast, defined duration operations only

*********************************************************************************/


#pragma interrupt PriorityUserInterruptHandler
void PriorityUserInterruptHandler(void)
{
    // Only very fast functions here
}


void UserInterruptHandler(void)
{
    // Slower functions here
}


/*********************************************************************************
ZigBeeHook

This function allows a sneak look at the state of the ZigBee stack.
If you care to investigate the ZigBee stack source, you can declare variables as extern
and look at them.  DO NOT modify them - that would invalidate stack compliance

*********************************************************************************/

void ZigBeeHook( void )
{
}


/*********************************************************************************
NetworkFailed

Works out what to do if cannot be sure we're still connected to the network
This implementation just waits a second, then resets.  Consideration should
be given to whether a longer delay should be implemented for power saving

*********************************************************************************/

void NetworkFailed( void )
{
    Sleep();
    Nop();
    Reset();
}
```

**High priority interrupt**

**Low priority interrupt**

**You can peek at the stack, but don't touch!**

**Recovery-on-error function**

## Function Reference

## Data Types

A variety of data structures are declared in `MailBox.h`. Most have rather obvious functions and will not be documented in detail – refer to `MailBox.h` for further information. The *MailBoxParams* structure is extensive and is detailed in the *MailBoxTasks()* section.

## MailBoxInit()

*MailBoxInit()* initializes the MailBox. It will enable high and low priority interrupts. The function should be called once during initialization.

## MailBoxTasks()

This section describes the function of the MailBox state machine based on the *MailBoxState* on entry to and on exit from *MailBoxTasks*.

Related information is stored in the *MailBoxParams* structure. For example, the Erase variable referred to in *MBS_Join_Request* below actually refers to the variable *MailBoxParams.MBSInitialize_request.Erase*.

All *MailBoxParams* fields named *Status* will be 0x00 to indicate success; otherwise the status code will be equal to a NWK layer status code (refer to ZigBee 1.0 specification section 2.1).

The following limitations apply as to when certain primitives may be specified by the application:

- An *MBS_Join_Request* operation must complete successfully before any other request is made.
- No variable used by the MailBox layer may be modified from an interrupt service routine.
- The MailBox state may only be changed by the application if the current state is *MBS_Idle*.

## Transient MailBox States

**MBS_Idle**

*Parameters:* None

*Meaning On Exit*: MailBox is free to accept requests.

*Action On Entry*: No action taken other than to run background ZigBee stack tasks.

**MBS_Wait**

*Parameters:* None

*Meaning On Exit*: MailBox is processing a request.

*Action On Entry*: Background ZigBee stack tasks are run to see if operations have completed.

**MBS_Error**

*Parameters:*

```
BYTE    ErrorNo
```

*Meaning On Exit*: MailBox met with a fatal error. *ErrorNo* indicates the nature of the error as follows:

| *ErrorNo* | *Description* |
|---|---|
| 0x01 | *MailBoxState* was not idle when value was changed. |
| 0x02 | Ran out of memory. |
| 0x03 | Operation not permitted prior to joining / forming network |
| ≥0x80 | Unexpected state; contact FlexiPanel quoting error number. |

*Action On Entry*: MailBox immediately returns without changing *MailBoxState*.

## Joining

**MBS_Join_Request**

*Parameters:*

```
BIT    Flags.Erase
```

*Meaning On Exit*. Cannot occur.

*Action On Entry*: Mailbox will (re)initialize. If *Erase* is nonzero, all ZigBee network information will be erased.

On a coordinator, this command will start a network. If it has never started a network before, or *Erase* was true, it will be a new network. Otherwise it will restart an existing network.

On routers and end devices, if it has never before joined a network or *Erase* was true, it will then attempt to join any network. A router or coordinator on the network it is supposed to join must be in range and in the *PermitJoin* state to allow the new device to join.

Otherwise, if it was previously a member of a network, it will rejoin the network and no special join permission will be required.

*Example usage:*

```
MailBoxState = MBS_Join_Request;
pMailBoxParam->MBS_Join_Request.Erase = 0;
if (bDoFactoryRset)
  pMailBoxParam->MBS_Join_Request.Erase = 1;
```

**MBS_Join_Confirm**

*Parameters:*

```
BYTE          Status
SHORT_ADDR    ShortAddress
```

*Meaning On Exit*. Initialization has completed.

If *Status* is 0x00, initialization was successful. If the device was already a member of a network, it has rejoined the network. If it was not a member, or network information was erased, the device has joined a new network. *ShortAddress* will contain the short address allocated to this device.

If *Status* is nonzero, initialization failed because the device could not join / rejoin a network.

*Action On Entry*: State returned to MBS_Idle or MBS_Wait.

*Example usage:*

```
if (MailBoxState == MBS_Join_Confirm)
{
  // Sleep until a human intervenes to reset
  if (pMailBoxParam->MBS_Join_Confirm.Status)
    Sleep();
}
```

**MBS_Permit_Join_Request**

*Parameters:*

```
BYTE    PermitDuration
```

*Meaning On Exit*. Cannot occur.

*Action On Entry*: Sets permissions for allowing new devices to join the network via this device. This action is only permitted for routers and coordinators.

If *PermitDuration* is 0x00, devices are not allowed to join the network via this device.

If *PermitDuration* is 0xFF, devices will be allowed to join the network via this device until otherwise instructed.

If *PermitDuration* is any other value, devices will be allowed to join the network via this device for then next *PermitDuration* seconds. No indication will be given when this duration has elapsed.

*Example usage:*

```
if (bPermitJoinButtonPressed)
{
  // Device joined; disable permit join
  MailBoxState = MBS_Permit_Join_Request;
  pMailBoxParam->MBS_Permit_Join_Request.
      PermitDuration = 0xFF;
}
```

**MBS_Permit_Join_Confirm**

*Parameters:*

```
BYTE        Status
```

*Meaning On Exit*. Permit join instruction has completed processing.

If *Status* is 0x00, processing was successful.

If *Status* is nonzero, processing was not successful.

*Action On Entry*: State returned to MBS_Idle or MBS_Wait.

**MBS_Permit_Join_Indication**

*Parameters:*

```
SHORT_ADDR    ShortAddress
LONG_ADDR     ExtendedAddress
```

*Meaning On Exit*. A device joined the network via this device. There is no explicit indication as to whether this is a rejoin of an existing member or a new member joining.

*ShortAddress* will contain the short address of the device that joined.

*ExtendedAddress* will contain the long address of the device that joined.

*Action On Entry*: State returned to MBS_Idle or MBS_Wait.

*Example usage:*

```
// Permit joining
MailBoxState = MBS_Permit_Join_Request;
pMailBoxParam->MBS_Permit_Join_Request.
      PermitDuration = 0x00;
```

**MBS_Leave_Indication**

*Parameters:* None

*Meaning On Exit*. This device was instructed to leave the network. This event could only have been initiated by a non-MailBox member of the ZigBee network.

*Action On Entry*: State returned to MBS_Idle or MBS_Wait.

**MBS_Sync_Loss_Indication**

*Parameters:* None

*Meaning On Exit*. This device did not receive a reply from its parent during a poll operation. Polling will have been tried four times; the application should assume that the parent is no longer operating, on this frequency at least. The recommended course of action is to sleep a while to conserve power, then reset and then attempting to rejoin the network.

*Action On Entry*: State returned to MBS_Idle or MBS_Wait.

## *Device Discovery*

**MBS_Device_Discovery_Request**

*Parameters:*

```
WORD_VAL    FuncID
```

*Meaning On Exit*. Cannot occur.

*Action On Entry*: Initiates a scan for other MailBox devices with the same Functional ID specified by *FuncID*.

*Example usage:*

```
MailBoxState = MBS_Device_Discovery_Request;
pMailBoxParam->MBS_Device_Discovery_Request.
    FuncID.Val = 0x0000;  // All devices
```

**MBS_Device_Discovery_Confirm**

*Parameters:*

```
BYTE        Status
```

*Meaning On Exit*. Device discovery has completed processing. Device discovery will complete after a period of two MIB_MailBoxTimeout periods.

If *Status* is 0x00, processing was successful.

*Action On Entry*: State returned to MBS_Idle or MBS_Wait.

**MBS_Device_Discovery_Indication**

*Parameters:*

```
WORD_VAL    SourceAddr
```

*Meaning On Exit*. A device was discovered conforming to the discovery request.

*SourceAddr* will contain the short address of the discovered device. It will support the Functional Address specified. However, it is not guaranteed to have the same Application ID.

*Action On Entry*: State returned to MBS_Idle or MBS_Wait.

*Example usage:*

```
if (MailBoxState==MBS_Device_Discovery_Indication)
{
  WORD_VAL Discovered_Device =
  pMailBoxParam->MBS_Device_Discovery_Indication.
       SourceAddr.Val;
}
```

*Example usage:*

```
if (MailBoxState==MBS_Device_Discovery_Indication)
{
  WORD_VAL Discovered_Device =
  pMailBoxParam->MBS_Device_Discovery_Indication.
       SourceAddr.Val;
}
```

**MBS_Service_Discovery_Request**

*Parameters:*

```
WORD_VAL    Destination
WORD_VAL    InfoType
BYTE        DataPayloadLen
BYTE*       pRxData
```

*Meaning On Exit*. Cannot occur.

*Action On Entry*: Initiates a request for information from the ZigBee device with address *Destination*.

*InfoType* indicates the information required. If equal to 0x0080, the information requested is the device's application ID. *DataPayloadLen* should be zero so that *pDataPayload* is be ignored.

If *InfoType* is anything other than 0x0080, the value shall be interpreted as a ZigBee Device Object (ZDO) cluster and the data will be sent to the ZDO on the remote device. For further information refer to the ZigBee specification. Note that for ZDO requests, *Destination* does not have to be a MailBox device.

If supplementary information is required for a ZDO request, the buffer *pTxZDO* shall be filled with the supplementary information and *DataPayloadLen* shall indicate its length. The buffer is a fixed memory location defined by a macro and memory does not need to be allocated. It must be filled at the time the *MailBoxState* is set to *MBS_Service_Discovery_Request*.

If supplementary information is not required, *DataPayloadLen* shall be zero.

The ZDO services currently supported by MailBox devices are shown below. Refer to the ZigBee specification for full details of their actual function and request / confirm payloads.

| ***Name*** | ***Description*** | ***Info Requested*** |
|---|---|---|
| IEEE_ADDR_req | Get MAC address & child device info | 0x0001 |
| NODE_DESC_req | Get node descriptor | 0x0002 |
| POWER_DESC_req | Get power descriptor | 0x0003 |
| SIMPLE_DESC_req | Get simple descriptor | 0x0004 |
| ACTIVE_EP_req | Get active endpoint list | 0x0005 |

*Example usage (Application ID):*

```
MailBoxState = MBS_Service_Discovery_Request;
pMailBoxParam->MBS_Service_Discovery_Request.
    Destination.Val = 0x0001;
pMailBoxParam->MBS_Service_Discovery_Request.
    InfoType.Val = 0x0080;
```

*Example usage (ZDO request):*

```
MailBoxState = MBS_Service_Discovery_Request;
pMailBoxParam->MBS_Service_Discovery_Request.
      Destination.Val = 0x0001;
pMailBoxParam->MBS_Service_Discovery_Request.
       InfoType.Val = IEEE_ADDR_req;
pMailBoxParam->MBS_Service_Discovery_Request.
    DataPayloadLen = 4;
pTxZDO[0] = 0x01;  // Destination address
pTxZDO[1] = 0x00;
pTxZDO[2] = 0x00;  // Request type
pTxZDO[3] = 0x00;  // Start index
```

**MBS_Service_Discovery_Confirm**

*Parameters (Application ID):*

```
BYTE        Status
WORD_VAL    InfoType
DWORD_VAL   ApplicationID
```

*Parameters (ZDO services):*

```
BYTE        Status
WORD_VAL    InfoType
BYTE        DataPayloadLen
BYTE*       pRxData
```

*Meaning On Exit*. A device information request has completed processing.

If *Status* is 0x00, processing was successful and *InfoType* will indicate the information requested.

If *InfoType* is 0x0080, the *ApplicationID* shall contain the remote device's Application ID.

If *InfoType* is anything other than 0x0080, *pRxData* shall be interpreted as a ZigBee Device Object (ZDO) response. For further information refer to the ZigBee specification. *DataPayloadLen* shall indicate its length. The data in *pRxData* shall only be valid until *MailBoxTasks()* is next called.

*Action On Entry*: State returned to MBS_Idle or MBS_Wait.

*Example usage (Application ID):*

```
if (MailBoxState == MBS_Service_Discovery_Confirm)
{
  if (!pMailBoxParam->
        MBS_Service_Discovery_Confirm_AppId.Status)
  {
    if (pMailBoxParam->
        MBS_Service_Discovery_Confirm_AppId.
        InfoType.Val==0x0080)
    {
       // Application ID is in:
       pMailBoxParam->
           MBS_Service_Discovery_Confirm_AppId.
           ApplicationID
    }
  }
}
```

*Example usage (ZDO):*

```
if (MailBoxState == MBS_Service_Discovery_Confirm)
{
  if (!pMailBoxParam->
        MBS_Service_Discovery_Confirm_AppId.Status)
  {
    if (pMailBoxParam->
        MBS_Service_Discovery_Confirm_AppId.
        InfoType.Val!=0x0080)
    {
      // ZDO response is in:
      pMailBoxParam->
        MBS_Service_Discovery_Confirm_ZDO.pRxData

      // Length is in:
      pMailBoxParam->
              MBS_Service_Discovery_Confirm_ZDO.
              DataPayloadLen
    }
  }
}
```

**MBS_Present_Request**

*Parameters:*

```
BIT         Flags.Broadcast
WORD_VAL    Destination
```

*Meaning On Exit*. Cannot occur.

*Action On Entry*: Sends a message to the device(s) specified by *Destination*, announcing its presence. If *Broadcast* is true, *Destination* is interpreted to be a Functional Cluster and the message is broadcast. If *Broadcast* is false, *Destination* is interpreted to be a short address and the message is unicast.

**MBS_Present_Confirm**

*Parameters:*

```
BYTE        Status
```

*Meaning On Exit*. Confirms that an *MBS_Present_Request* operation has completed.

If *Status* is 0x00, processing was successful.

*Action On Entry*: State returned to MBS_Idle or MBS_Wait.

**MBS_Present_Indication**

*Parameters:*

```
WORD_VAL       SourceAddr
BYTE           NumCluster
WORD_VAL       ClusterList[]
```

*Meaning On Exit*. A device announced it was present.

*SourceAddr* will contain the short address of the originating device. It will have the correct Application ID.

*ClusterList* will list the functional clusters that the device supports. (The mandatory cluster 0x0000 may or may not be in the list.) *NumCluster* will equal the number of clusters in *ClusterList*. Any null clusters (0xFFFF) in the list shall be ignored.

*Action On Entry*: State returned to MBS_Idle or MBS_Wait.

**MBS_Redirect_Request**

*Parameters:*

```
BIT         Flags.Broadcast
WORD_VAL    Destination
BIT         Redirect.IsBroadcast
BIT         Redirect.IsAdknowledge
WORD_VAL    AddrOrClust
```

*Meaning On Exit*. Cannot occur.

*Action On Entry*: Sends a message to the device(s) specified by *Destination*, requesting that they change their redirect address. If *Broadcast* is true, *Destination* is interpreted to be a Functional Cluster and the message is broadcast. If *Broadcast* is false, *Destination* is interpreted to be a short address and the message is unicast.

The message instructs the destination mailbox layer to set its redirect address to *AddrOrClust*, which shall be interpreted as a short address if *Redir.IsBroadcast* is false, or as a broadcast cluster otherwise. Transmissions using the redirect address shall be acknowledged if *Redir.IsAcknowledge* is true.

**MBS_Redirect_Confirm**

*Parameters:*

```
BYTE        Status
```

*Meaning On Exit*. Confirms that an *MBS_Redirect_Request* operation has completed.

If *Status* is 0x00, processing was successful.

*Action On Entry*: State returned to MBS_Idle or MBS_Wait.

**MBS_Redirect_Indication**

*Parameters:* None.

*Meaning On Exit*. The redirect address was changed by another device.

*Action On Entry*: State returned to MBS_Idle or MBS_Wait.

## *Power Saving*

**MBS_Device_Sleep_Request**

*Parameters:* None.

*Meaning On Exit*. Cannot occur.

*Action On Entry*: Mailbox will begin to shut down the ZigBee stack.

Note: Only sleepy end devices should be permitted to sleep unless a network sleep command was received. An end device's parent can cache messages for it during sleep, but it must wake up to retrieve the message. There is no mechanism for the router to wake up the end device when there has a message for it.

**MBS_Device_Sleep_Confirm**

*Parameters:*

```
BYTE        Status
```

*Meaning On Exit*. ZigBee stack has shut down.

If *Status* is 0x00, device may sleep. Currently, it cannot return nonzero.

*Action On Entry*: State returned to MBS_Idle or MBS_Wait.

*Parameters:*

```
BIT         Flags.Broadcast
WORD_VAL    Destination
```

**MBS_Device_Wake_Request**

*Parameters:* None.

*Meaning On Exit*. Cannot occur.

*Action On Entry*: MailBox will start to resume the ZigBee stack.

**MBS_Device_Wake_Confirm**

*Parameters:*

```
BYTE        Status
```

*Meaning On Exit*. The operation to resume the ZigBee stack has completed.

If *Status* is 0x00, the stack resumed successfully and the device may resume normal operation.

*Action On Entry*: Ignored; state returned to MBS_Idle.

**MBS_Network_Sleep_Request**

*Parameters:*

```
BIT             Flags.Broadcast
WORD_VAL        Destination
WORD_VAL        ClosedownPeriod
THREEBYTE_VAL   SleepPeriod
WORD_VAL        WakeupPeriod
```

*Meaning On Exit*. Cannot occur.

*Action On Entry*: Sends a message to the device(s) specified by *Destination*, indicating that they may sleep. If *Broadcast* is true, *Destination* is interpreted to be a Functional Cluster and the message is broadcast. If *Broadcast* is false, *Destination* is interpreted to be a short address and the message is unicast.

Refer to *Network-Sleep.indication* for definitions of *ClosedownPeriod*, *SleepPeriod* and *WakeupPeriod*.

**MBS_Network_Sleep_Confirm**

*Parameters:*

```
BYTE        Status
```

*Meaning On Exit*. Confirms that an *MBS_Network_Sleep_Request* operation has completed.

If *Status* is 0x00, processing was successful.

*Action On Entry*: State returned to MBS_Idle or MBS_Wait.

**MBS_Network_Sleep_Indication**

*Parameters:*

```
WORD_VAL            SourceAddr
WORD_VAL            ClosedownPeriod
THREEBYTE_VAL       SleepPeriod
WORD_VAL            WakeupPeriod
```

*Meaning On Exit*. A network sleep message was received. *SourceAddr* will contain the short address of the new device. It will have the correct Application ID.

*ClosedownPeriod* is the duration, in seconds, from the moment *Network-Sleep.indication* was generated, that the application should continue to keep its MailBox layer active on and respond to messages if required. The application may not initiate any new MailBox operations within the *Closedown* period.

The *SleepPeriod* is the duration, in seconds, from the moment the *Network Sleep* message was received, that the receiving applications may enter a sleep state. Prior to sleeping, an MBS_*Device_Sleep_Request* should be issued to suspend and power down the ZigBee stack.

At the end of the *Sleep period*, the application shall wake up and issue an MBS_*Device_Wake_Request* to power up and resume the ZigBee stack.

The *WakeupPeriod* is the duration, in seconds, immediately following the end of the *Sleep period*. During this period, the application may not initiate any transmissions unless it first receives a transmission from another device.

Sleep is optional on receipt of a *Network-Sleep.indication*. Devices which must remain awake in order to continue to provide services for non-MailBox nodes in the ZigBee network should stay awake.

*Action On Entry*: State returned to MBS_Idle or MBS_Wait.

## *Data Transfer*

### MBS_Data_Request

*Parameters:*

```
BIT             Flags.IsBroadcast
BIT             Flags.IsAcknowledge
BIT             Flags.IsRetry
WORD_VAL        Destination;
BYTE            DataPayloadLen
BYTE*           pRxData
```

*Meaning On Exit*. Cannot occur.

*Action On Entry*: Sends the data in *pTxData* to the the device(s) specified by *Destination*. *DataPayloadLen* shall indicate the number of bytes to be sent.

If Is*Broadcast* is true, *Destination* is interpreted to be a Functional Cluster and the message is broadcast. If Is*Broadcast* is false, *Destination* is interpreted to be a short address and the message is unicast.

If *IsAcknowledge* is false, a *Data.Confirm* will be issued as soon as transmission to nearest neighbors is complete. If *IsAcknowledge* is false, a *Data.Confirm* will be issued only when the destination has replied with an acknowledgement, or the operation times out. Acknowledgement is only permitted for non-broadcast transmissions.

*IsRetry* should be false unless this is a repeated attempt to transmit the previous packet.

### MBS_Data_Confirm

*Parameters:*

```
BYTE            Status
BYTE            Seq
WORD_VAL        RetryDelay
```

*Meaning On Exit*. The data request has completed processing. Possible status return codes are given below.

| ***Name*** | ***Description*** | ***Value*** |
|---|---|---|
| *SUCCESS* | Operation completed successfully | 0x00 |
| *UNKNOWN* | This packet received OK but insufficient information to determine if it is in sequence | 0x01 |
| *SEQUENCE_ERROR* | An error occurred (unicast) | 0x02 |
| *NOT_PERMITTED* | Operation not permitted | 0x03 |
| *TIMED_OUT* | Acknowledge not received | 0x04 |
| *RETRY_LATER* | Destination could not accept packet; try again later | 0x05 |
| *STACK_FAIL* | A ZigBee stack failure occurred | 0x06 |
| *RESET_MISMATCH* | This packet received OK but a device has been reset | 0x09 |
| *CHECKSUM_FAIL* | Packet not received due to checksum failure | 0x0A |

*SUCCESS* shall only confirm that the destination(s) received the packets if the transmission was acknowledged.

If the status is *TIMED_OUT*, *RETRY_LATER*, or *CHECKSUM_FAIL*, the application may retry the data request as many times as desired, provided it does so prior to transmitting any further packets. If it does so, *pTxData* must be reloaded and the *Data.request* flag *Retry* should be set to true. If the status is *RETRY_LATER*, the retry should occur no sooner than $256 \times RetryDelay$ symbol periods later.

*Seq* is provided for informational purposes and indicates the sequence number used to transmit the packet. If the status is *STACK_FAIL*, *Seq* will contain the ZigBee stack error status value. The most likely error status is 0x03, *APS_NO_ACK*, indicating no response from another device.

*Action On Entry*: State returned to MBS_Idle or MBS_Wait.

### MBS_Data_Indication

*Parameters:*

```
BYTE            Status
WORD_VAL        SourceAddr
BYTE            Seq
BYTE            DataPayloadLen
BYTE*           pRxData
```

*Meaning On Exit*. A data frame was received from another device. Possible status return codes are gives below.

| Name | Description | Value |
|---|---|---|
| *SUCCESS* | Operation completed successfully | 0x00 |
| *UNKNOWN* | This packet received OK but insufficient information to determine is in sequence | 0x01 |
| *SEQUENCE_ERROR* | An error occurred (unicast) | 0x02 |
| *STACK_FAIL* | A ZigBee stack failure occurred | 0x06 |
| *FRAMES_LOST* | This packet received OK but earlier packet(s) are missing | 0x07 |
| *LATE_FRAME* | This packet received OK but a later packet has already been the subject of a Data.indication | 0x08 |
| *RESET_MISMATCH* | This packet received OK but a device has been reset | 0x09 |

*SourceAddr* will contain the short address of the sender. If *DataPayloadLen* is nonzero, *pRxData* will contain the data payload and *DataPayloadLen* shall indicate its length. The data in *pRxData* shall only be valid until *MailBoxTasks()* is next called.

*Action On Entry*: State returned to MBS_Idle or MBS_Wait.

**MBS_Custom_Request**

*Parameters:*

```
BYTE          Custom_FID
BIT           Flags.Broadcast
WORD_VAL      Destination
BYTE          DataPayloadLen
BYTE*         pRxData
```

DataPayloadLen *Meaning On Exit*. Cannot occur.

*Action On Entry*: Sends the custom frame *Custom_FID* to the device(s) specified by *Destination*. If *Broadcast* is true, *Destination* is interpreted to be a Functional Cluster and the message is broadcast. If *Broadcast* is false, *Destination* is interpreted to be a short address and the message is unicast.

If a data payload accompanies the request, the buffer *pTxCustom* shall contain it and *DataPayloadLen* shall indicate its length. The buffer is a fixed memory location defined by a macro and memory does not need to be allocated. It must be filled at the time the *MailBoxState* is set to *MBS_Service_Discovery_Request*.

If no payload required, *DataPayloadLen* shall be zero.

*Example usage:*

```
MailBoxState = MBS_Custom_Request;
pMailBoxParam->MBS_Custom_Request.Flags.
    Broadcast = 1;
pMailBoxParam->MBS_Custom_Request.Destination.
    Val = 0x0000;
pMailBoxParam->MBS_Custom_Request.
    Custom_FID = 0x80;
pMailBoxParam->MBS_Custom_Request.
    DataPayloadLen = 7;  // 7-char message
NVMRead( (void *) pTxCustom,
    (void *) "Hello\r\n", 7 );
```

**MBS_Custom_Confirm**

*Parameters:*

```
BYTE        Status
```

*Meaning On Exit*. The custom request has completed processing.

If *Status* is 0x00, processing was successful in terms of transmitting to the nearest neighbor. Custom frames are not acknowledged, so this does not guarantee that the message was received by the destination.

*Action On Entry*: State returned to MBS_Idle or MBS_Wait.

**MBS_Custom_Indication**

*Parameters:*

```
BYTE          Custom_FID
WORD_VAL      SourceAddr
BYTE          DataPayloadLen
BYTE*         pRxData
```

*Meaning On Exit*. A custom frame was received from another device.

*Custom_FID* will contain the ID of the custom message. *SourceAddr* will contain the short address of the sender. If *DataPayloadLen* is nonzero, *pRxData* will contain the data payload and *DataPayloadLen* shall indicate its length. The data in *pRxData* shall only be valid until *MailBoxTasks()* is next called.

*Action On Entry*: State returned to MBS_Idle or MBS_Wait.

*Example usage:*

```
if (MailBoxState == MBS_Custom_Indication )
{
  // Payload is in:
  pMailBoxParam->MBS_Service_Discovery_Confirm_ZDO.
      pRxData

  // Length is in :
  pMailBoxParam->MBS_Service_Discovery_Confirm_ZDO.
      DataPayloadLen
}
```

## *Callback Functions*

The following three callback functions must be provided even if they are not used:

**void PriorityUserInterruptHandler(void)**

Used for very high priority interrupt processing. This takes precedent over all other processing,

including caching incoming ZigBee messages. Anything using the priority interrupt handles must complete very quickly. Typically it will be used to note that an event has occurred, for example UART data has been received, so that it may be processed at a later time.

**void UserInterruptHandler(void)**

Used for normal priority interrupt processing.

**void ZigBeeHook(void)**

ZigBeeHook allows advanced users to sneak look at the state of the ZigBee stack. If you investigate the ZigBee stack source, you can declare variables as extern and inspect them from within this hook function. DO NOT modify any variables - that would invalidate stack compliance.

## *Utility Functions*

The utility functions are parts of the ZigBee stack which have been exposed because the application may also have independent uses for them. *Note: MailBoxInit() must be called before using any of these functions.*

**void NVMWrite(NVM_ADDR *dest, BYTE *src, BYTE count)**

***NVMWrite*** writes up to 255 bytes to nonvolatile memory. Note that the entire 0x40-byte-aligned section of memory is erased and restored during this process, so the entire section of memory should not contain executable code. Data is verified before writing, so writing identical values to the memory does not exhaust it.

**void NVMRead(BYTE *dest, NVM_ADDR *src, BYTE count)**

***NVMRead*** reads up to 255 bytes of nonvolatile memory.

**TICK TickGet(void)**

***TickGet*** provides a clock which counts symbols (1/62,500ths of a second). The timer is 4 bytes, so the clock rolls over very 19 hours.

**TICK TickGetDiff(TICK a, TICK b)**

***TickGetDiff*** calculates the difference between two time values.

**unsigned char * SRAMalloc(unsigned char nBytes)**

***SRAMalloc*** allocates and returns the address of nBytes of RAM from the heap. Zero is returned if the memory could not be allocated.

**void SRAMfree(unsigned char * pSRAM)**

***SRAMfree*** must be used to free memory previously allocated with SRAMalloc.

## *MailBox Information Base (MIB)*

The following mailbox constants are used by the MailBox:

***MIB_ApplicationEndpoint*** Endpoint used by this Application ID. The default value is 0x10*. (1 byte)*

***MIB_ApplicationID*** Four-byte Application ID. If the application can guarantee that it is the only operator of MailBox applications on the network, the three most significant bytes may be 00:00:00; the least significant byte may then be application specific. To obtain a unique allocation of the upper three bytes, contact FlexiPanel Ltd. Default value is 0x00000000*. (4 bytes)*

***MIB_ClusterList*** A list of seven 2-byte functional clusters which are to be supported in addition to the mandatory cluster, 0x0000. If fewer than seven clusters are to be specified, the remainder should be set to the null cluster value, 0xFFFF. Default value is all null clusters*.* With ZigBee 1.0, the maximum non-null cluster value is 0x00FD. *(14 bytes)*

***MIB_MACaddress*** The 8-byte MAC address of the device. This must be set before starting MailBox operations. *(8 bytes)*

***MIB_MailBoxTimeout*** Number of symbol periods expected for a message to propagate to another node in a network and for a reply to be received. If a reply is not received within this time, or a multiple of it if appropriate, an operation will be abandoned. Default value is 0x00010000 – approx one second. *(4 bytes)*

***MIB_PALevel*** Transmit power as device in CC2420 documentation. Default value is 0xFF, maximum power. *(1 byte)*

***MIB_PersistTime*** Symbol periods that a message should reside on a router pending collection by a sleepy end device. Default value is 0x00400000 – approx one minute. *(4 bytes)*

***MIB_PollRate*** Symbol periods between sleepy end device's polls to its parent for messages. Sleepy devices are required to do this while awake in order to pick up messages. Other devices (except the coordinator) may choose to do it simply to confirm that the parent is still there.

Default value is 0x0080 for sleepy devices (approx half a second) and 0x0000 for other devices.

***MIB_RetryDelay*** The period, in multiples of 256 symbol periods, which a remote device should pause before attempting to retransmit if incoming data is to be rejected for flow control reasons. If the data is to be accepted, the value is 0x0000. The default value is 0x0000. Unlike most other MIB values, this value is in RAM and so may be set directly. It is reset to the default value on startup. *(2 bytes)*

***MIB_RedirectAddr*** The destination to send messages to when the Redirect Cluster (0xFFFE) is specified in a Data.request. If the first byte is zero, the second and third bytes shall be interpreted as a short address. If the first byte is nonzero, the second and third bytes shall be interpreted as a broadcast cluster.

The value 0xFFFFFF shall be interpreted as no address being specified and no transmission shall be made and the Data.confirm shall report success. Unlike most other MIB values, this value is in RAM and so may be set directly. It is reset to 0xFFFFFF on startup. *(3 bytes)*

***MIB_SeqBufSize*** Number of sequence records to buffer in order to monitor packet sequence / loss / repetition. Refer to MailBox profile definition for details. Default value is 0x10 for Pixie, 0x04 for Pixie Lite. Maximum value 0x2A. *(1 byte)*

## *Macros*

Note that setting nonvolatile MIB values may not necessarily have any effect until the ZigBee stack is next initialized.

***SetAppID(x)***, ***SetAppID(x)*** set and retrieve ***MIB_ApplicationID***, where ***x*** is a pointer to a 4-byte buffer in RAM.

***SetAppEP(x)***, ***SetAppEP(x)*** set and retrieve ***MIB_ApplicationEndpoint***, where ***x*** is a pointer to a 1-byte buffer in RAM.

***SetClusterList(x)***, ***SetClusterList(x)*** set and retrieve ***MIB_ClusterList***, where ***x*** is a pointer to a 14-byte buffer in RAM.

***SetMACAddress(x)***, ***GetMACAddress(x)*** set and retrieve ***MIB_MACaddress***, where ***x*** is a pointer to an 8-byte buffer in RAM. The device must be reset after setting this value.

***SetMailBoxTimeout(x)***, ***GetMailBoxTimeout(x)*** set and retrieve ***MIB_MailBoxTimeout***, where ***x*** is a pointer to a 4-byte buffer in RAM.

***SetPALevel(x)***, ***SetPALevel(x)*** set and retrieve ***MIB_PALevel***, where ***x*** is a pointer to a 1-byte buffer in RAM.

***SetPersistTime(x)***, ***GetPersistTime(x)*** set and retrieve ***MIB_PersistTime***, where ***x*** is a pointer to a 4-byte buffer in RAM.

***SetSeqBufSiz(x)***, ***SetSeqBufSiz(x)*** set and retrieve ***MIB_SequenceBufferSize***, where ***x*** is a pointer to a 1-byte buffer in RAM.

**extern WORD_VAL MIB_RetryLater** may be read and modified directly by the application at any time.

**extern REDIR_VAL MIB_*RedirectAddr*** may be read and modified directly by the application at any time.

***RANDOM_LSB***, ***RANDOM_MSB*** provide a pseudorandom number generated from the Timer0 clock.

## *Development Kit Inventory*

The ToothPIC Development Kit contains:

1. The files `MailBox.h`, `MailBox-PHCC.lib`, `MailBox-PHCR.lib`, `MailBox-PHCF.lib`, `MailBox-PHCS.lib`, `MailBox-LHCF.lib`, `MailBox-LHCE.lib`, `MbxLink4620.lkr` and `MbxLink2520.lkr`, which are required for MPLAB-based applications development.

2. The files `PixieDARC.c`, `Config.c`, which are required for the example application, and project files for the six different builds PXDC, PXDR, PXDF, PXDE, PLDF, and PLDE.

3. This documentation, `MailBoxAPI.pdf`.

4. The PixieDARC documentation, `PixieDARC.pdf`.

The MPLAB development environment and C18 compiler must be bought separately from Microchip Technology Inc (www.microchip.com) or one of its distributors.

## *Development Support*

FlexiPanel Ltd publishes free Sniffer firmware which parses MailBox frames which may prove useful. Mailbox.lib code snippets can also be provided to partners if a nondisclosure agreement is signed. Please contact us if you have any comments or suggestions.

## *Revision History*

| Version | Date | Major revisions |
|---|---|---|
| 103521200906 | 20-Sep-06 | Initial release |
| 103521061006 | 06-Oct-06 | ZigBee Alliance Profile 0xC1EE assigned |

Please note the following with ToothPIC release 103521200906:

- Security not implemented.

- MailBox Profile may be subject to revision during the registration process with the ZigBee Alliance. Commercial product release with this version may require later firmware upgrades to maintain compatability.

## *Contact details*

The MailBox profile was developed by FlexiPanel Ltd:

*FlexiPanel*

FlexiPanel Ltd
2 Marshall Street, 3rd
London W1F 9BB, United Kingdom
*www.flexipanel.com*
*email: support@flexipanel.com*