Power over Ethernet

PoE Host software communication API

User Guide

Revision 1.1

Catalog Number POE_HOST_SW_UG

Table of Contents

1 Introduction

The PoE Host software communication API simplifies the communication between the Host and Microsemi PoE controller.

PoE communication controller use 15 bytes communication protocol. See document Serial Communication Protocol User Guide (included in the package).

The communication protocol between the Host CPU and the PoE Controller is handled by the PoE Host software communication API eliminating the need to construct 15 byte communication packet format for each PoE command

Communication between Host and PoE software API is based on Various Data Block types arranged according to Commands/Requests fields sharing same memory resources (Union structures).

Using union structure provides the following advantages:

- More flexibility on data structure changes
- Small shared memory size

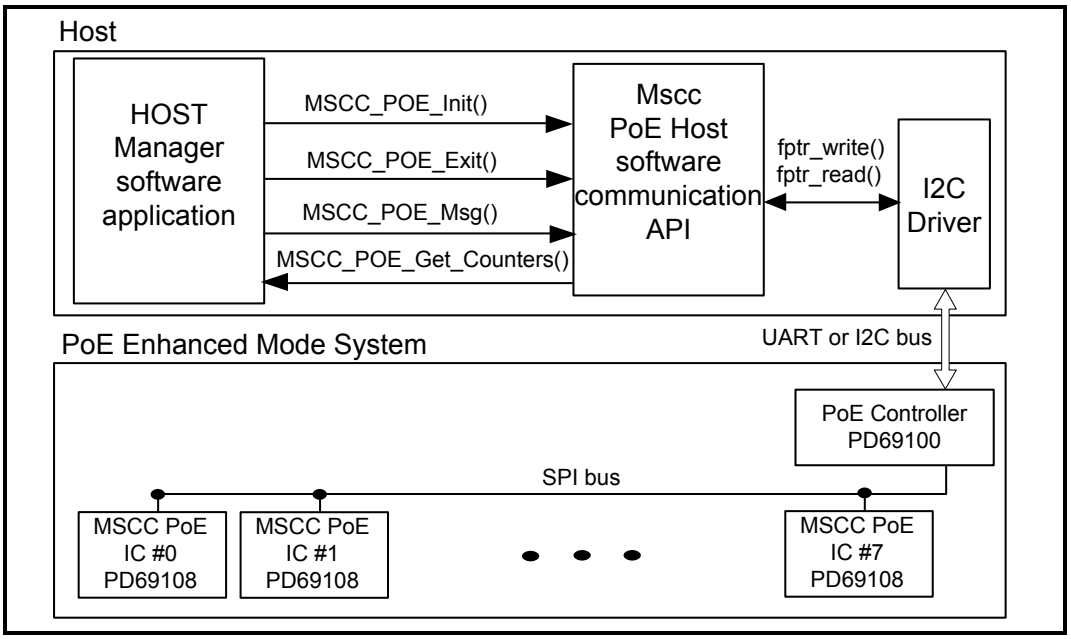

Figure 1: General System Schematic

Note: to simplify user software interface, only mili unit is in use. For example: mV (mili volt), mW (mili Watt), mA (mili Amper), mC (mili Celsius).

1.1 Host PoE API software flow

- **MSCC_POE_Init()** – initializes the PoE Host Communication software package.
- **MSCC_POE_Clear_POE_Device_Buffer()** – call this function as part of startup process to clear PoE Enhanced Mode System device previous PoE messages .
- **MSCC_POE_Msg()** – call this function to exchange information (get/set) between Host and PoE devices.
- **MSCC_POE_Get_Counters()** – call this function to obtain various statistics counters such as transmission counters and errors counters. All counters are 32 bit width.
- **MSCC_POE_Exit()** - call this function to terminate communications with Microsemi PoE devices**.**

By using Microsemi's simplified API, the software programmer has no need of handling the 15bytes communication protocol fields or the communication with the PoE devices.

Note: For detailed description on the following functions, please refer to ***Host PoE API for 15bytes protocol over I2C bus Interface Description*** on page 8.

1.2 Threads

The PoE API driver code does use any thread. The PoE API was designed to be "thread safe" to support multi threading. The PoE API has only a single instance and therefore it should be initialized only once by calling **MSCC_POE_Init().**

Note: when all PoE actions are called from a single thread there is no need of implementing the Mutex functions inside the API.

.

1.3 Software Distribution Structure

Software package is made up of three sections:

- **Examples**: Software distribution contains two Linux based examples. Each example is linked with the *poe_host_comm_api_lib.a* and *mscc_arcitecture_lib.a* libraries. The examples were created and tested on Linux Fedora Core 9 distribution.
- **Mscc_architecture**: This folder contains all the software glue functions (as OS_Sleep_mS()) required by PoE API and software examples in order to function properly over various operating systems such as Linux, etc. The makefile is optimized for GNU GCC. Typing 'make' will compile all files into an Architecture library named *mscc_arcitecture_lib*.a which will be linked by the example code into an executable application.
- **Poe_host_comm_api**: This folder contains PoE host communication 15 byte API software wrapper simplifying communication with PoE devices. All the software code inside this folder is **ANSI-C** compliant. It uses *mscc_arcitecture_lib*.a library for operating system call's functions such as OS_Sleep_mS().Typing 'make' will compile all files into an Architecture library named *poe_host_commI_api_lib.a*, which will be linked by the example code into an executable application.

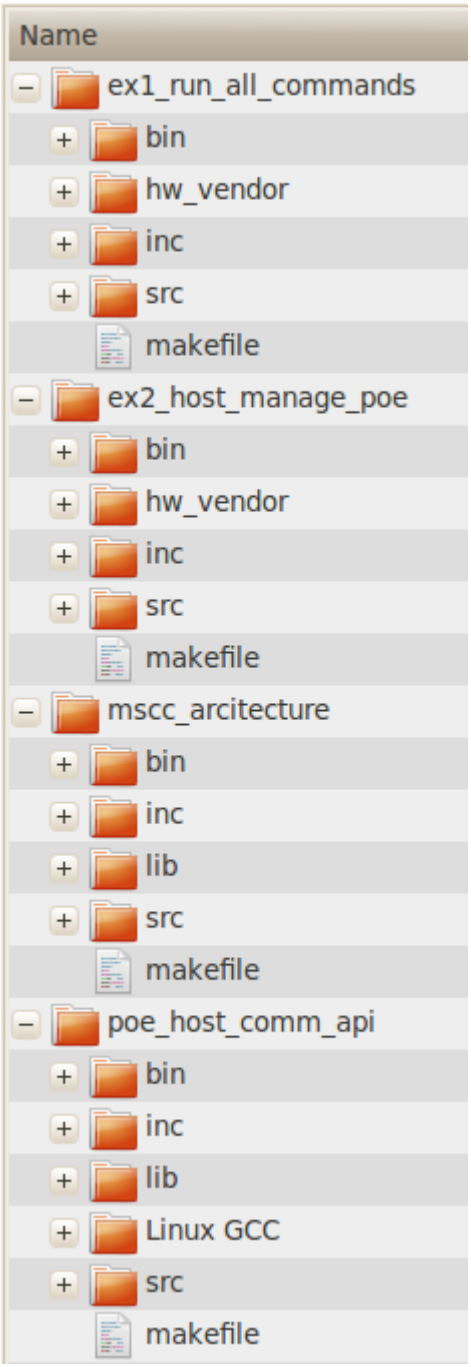

Figure 2: Host PoE API Workspace Tree Structure

2 Code Data Types

To comply with various CPU architectures (16/32 bit CPU's) and various compilers, PoE software API uses the following code conventions for variables used inside the code.

```
typedef char               S8   ;
typedef unsigned char      U8   ;
typedef signed short       S16  ;
typedef unsigned short     U16  ;
typedef long               S32  ;
typedef unsigned long      U32  ;
```

3 Code Functions Return Value

All PoE API software return values can be found in file *MSCC_POE_STATUS_e*.
Zero represents "OK". Negative value represents an error (see the list below).

```
typedef enum
{
    e_POE_STATUS_OK                                                  =  0,
    e_POE_STATUS_ERR_RESET_DETECTED                                  = -1,
    e_POE_STATUS_ERR_I2C_DEVICE_ERROR_OR_POE_UNIT_HARDWARE_RESET     = -2,
    e_POE_STATUS_ERR_COMMUNICATION_TIMEOUT                           = -3,

    e_POE_STATUS_ERR_REPLY_COMMAND_RECEIVED_WRONG_CHECKSUM           = -4,
    e_POE_STATUS_ERR_REPLY_FAILED_EXECUTION_CONFLICT_IN_SUBJECT_BYTES = -5,
    e_POE_STATUS_ERR_REPLY_FAILED_EXECUTION_WRONG_DATA_BYTE_VALUE    = -6,
    e_POE_STATUS_ERR_REPLY_FAILED_EXECUTION_UNDEFINED_KEY_VALUE      = -7,
    e_POE_STATUS_ERR_REPLY_CODE_OUT_OF_RANGE                         = -8,

    e_POE_STATUS_ERR_RX_MSG_CONFLICT_IN_DATA_BYTES                   = -9,
    e_POE_STATUS_ERR_RX_MSG_CONFLICT_IN_KEY_BYTES                    = -10,
    e_POE_STATUS_ERR_RX_MSG_CHECKSUM_ERR                             = -11,
    e_POE_STATUS_ERR_RX_MSG_ECHO_MISMATCH                            = -12,

    e_POE_STATUS_ERR_MSG_TYPE_NOT_EXIST                              = -13,

    e_POE_STATUS_ERR_MUTEX_INIT_ERROR                                = -14,
    e_POE_STATUS_ERR_MUTEX_DESTROY_ERROR                             = -15,
    e_POE_STATUS_ERR_MUTEX_LOCK_ERROR                                = -16,
    e_POE_STATUS_ERR_MUTEX_UNLOCK_ERROR                              = -17,
    e_POE_STATUS_ERR_SLEEP_FUNCTION_ERROR                            = -18,
    e_POE_STATUS_ERR_MUTEX_ALREADY_INITIALIZED                       = -19,
    e_POE_STATUS_ERR_MUTEX_ALREADY_DISPOSED                          = -20,

    e_POE_STATUS_ERR_OPEN_I2C_DEVICE_ERROR                           = -21,
    e_POE_STATUS_ERR_CLOSE_I2C_DEVICE_ERROR                          = -22,
    e_POE_STATUS_ERR_INITIALIZATION_ERROR                            = -23,
}mscc_POE_STATUS_e;
```

4 Host PoE API for 15bytes protocol over I2C bus Interface Description

The following software functions are used to control the PoE API:

- MSCC_POE_Init()
- MSCC_POE_Exit()
- MSCC_POE_Clear_POE_Device_Buffer()
- MSCC_POE_Msg()
- MSCC_POE_Get_Counters()

4.1 MSCC_POE_Init()

```
POE_STATUS_e MSCC_POE_Init(_IN INIT_INFO_t *ptInitInfo,_OUT S32 *plDevice_error);
```

4.1.1 Details

This function initializes *MSCC PoE API* by providing I^2C driver read and write functions call.

4.1.2 Arguments

- INIT_INFO_t *pInitInfo: the structure contains the followed members:
  - FPTR_Write fptr_write: I^2C driver write-function pointer (has to be implemented by the user). The I^2C function pointer has the following format:

    *typedef S32 (*FPTR_Write)(U8 I2C_Address, const U8* Txdata, U16 num_write_length);*

    _IN U8 I2C_Address: PoE device I2C address.

    _IN const U8* pTxdata: Pointer to bytes data array to be transmitted.

    _IN U16 num_write_length: Number of bytes to be transmitted.
  - FPTR_Read fptr_read: I^2C driver read-function pointer (has to be implemented by the user). I^2C function pointer has the following format:

    *typedef S32 (*FPTR_Read)(U8 I2C_Address,U8* Rxdata, U16 number_of_bytes_to_read);*

    _IN U8 I2C_Address: IC's I2C address.

    _IN U8* pRxdata: Pointer to bytes data array where driver should place received I2C data.

    _IN U16 length: number of received bytes.
- OUT S32 *pDevice_error: Pointer where to place user I^2C driver returned error code during init stage.

4.1.3 Return Value

POE_STATUS_e

4.1.4 Example using SUB20 I2C over USB device

Note: In the example bellow the Sub20 I2C over USB device read and write functions are assigned, followed by calling to function MSCC_POE_Init().

```
/*=============================================================================
/ Start of InitInfo struct building
/============================================================================*/
INIT_INFO_t tInitInfo;

/* Type the pointer for the functions which read and write I2C 15 bytes */
tInitInfo.fptr_write = Sub20_Write;/* pointer for Writing driver function */
tInitInfo.fptr_read = Sub20_Read;  /* pointer for Reading driver function */

/*=============================================================================
/ End of InitInfo struct building
/============================================================================*/

/* Init PoE API software */
ePoe_status = MSCC_POE_Init(&tInitInfo,&plDevice_error);
if (ePoe_status != e_POE_STATUS_OK)
{
    MSCC_POE_UTIL_PrintStatus(_IN "Init",_IN ePoe_status,_IN plDevice_error);
    goto FINISH;
}
```

4.2 MSCC_POE_Exit()

```
POE_STATUS_e MSCC_POE_Exit();
```

4.2.1 Details

This function terminates Host PoE API functionality (has to be called only once).

4.2.2 Return Value

POE_STATUS_e

4.2.3 Example

```
POE_STATUS_e ePoe_status = MSCC_POE_Exit();
if(ePoe_status != POE_STATUS_OK)
{
   //Error occurred
}
```

4.3 MSCC_POE_Clear_POE_Device_Buffer ()

```
POE_STATUS_e MSCC_POE_Clear_POE_Device_Buffer (_IN U8 bIC_Address,_OUT S32
*plDevice_error);
```

4.3.1 Details

MSCC_POE_Clear_POE_Device_Buffer API command clears the specific PoE Enhanced Mode System communication Transmit buffer that we are going to interact with.

Upon PoE device power up, a unique message is being sent by the PoE device reporting various powers up status information. The above function analyzes and clears this message in order to allow a proper communication between Host and PoE device.

Note: this function must be called immediately after calling MSCC_POE_Init() for each PoE device.

4.3.2 Arguments

- _IN U8 bIC_Address: the PoE device I2C address.
- OUT S32 * plDevice_error: Pointer where to place user I2C driver returned error code during write operation.

4.3.3 Return Value

POE_STATUS_e

Return status is used by the Host example to detect one of the following states :(see example code below):

1. Buffer is clear and device communicates correctly.
2. Hardware reset detected, buffer cleared and device communicate correctly.
3. There is an I2C communication problem (wrong I2C address or No PoE device connected correctly).

4.3.4 Example

```
eMscc poe status = MSCC POE Clear POE Device Buffer( IN bI2C Device Address, OUT
&ulDevice_error);

if (eMscc_poe_status == ePOE_STATUS_OK)
{
    printf("POE device: 0x%X OK, PoE device communicate correctly\n",
bI2C_Device_Address);
}
else if (eMscc_poe_status == ePOE_STATUS_ERR_POE_DEVICE_RESET_DETECTED)
{
    printf("POE device: 0x%X OK, PoE device reset detected ,buffer cleaned and
communicate correctly\n",bI2C_Device_Address);
}
else
{
    printf("CLEAR_POE_DEVICE_BUFFER operation failed, I2C device error: %ld
message: %s\n",ulDevice_error,fpDevice_status_string(ulDevice_error));
        goto FINISH;
}
```

4.4 MSCC_POE_Msg()

```
POE_STATUS_e MSCC_POE_Msg(_IN PROTOCOL_15BYTES_MSG_TYPE_e eMessageType, _INOUT
POE_MSG_t *ptMessage, _IN U8 bIC_Address,_OUT S32 *plDevice_error);
```

4.4.1 Details

This function should be used to send various Host PoE command/request messages through various Data Block types.

4.4.2 Arguments

- _IN PROTOCOL_15BYTES_MSG_TYPE_e eMessageType: the specific 15bytes communication protocol command/telemetry.
- _INOUT POE_MSG_t *ptMessage: pointer to Data Block type arranged according to eMessageType

- _IN U8 bIC_Address: the PoE device I2C address.
- OUT S32 * plDevice_error: Pointer where to place user I2C driver returned error code during write operation.

4.4.3 Return Value

POE_STATUS_e

4.4.4 Example

```
S32 ulDevice_error;   /* contain I2C device error number   */
POE_STATUS_e ePoe_status; /* poe software status */
POE_MSG_t tMessage;
U8 bI2C_Device_Address = 0x20;

PROTOCOL_15BYTES_MSG_TYPE_e eMsgType = eSave_User_Byte;
memset(&tMessage,0,sizeof(tMessage));

tMessage.HC08_TX_MSG_PRM_e.Save_User_Byte_t.Set.bUserByte = 0x5A;

/* Send Data Block type to PoE API */
ePoe_status= MSCC_POE_Msg(eMsgType,&tMessage,bI2C_Device_Address,&ulDevice_error);
if (ePoe_status != e_POE_STATUS_OK)
{
    //Error occurred
}
```

4.5 MSCC_POE_Get_Counters()

```
Void MSCC_POE_Get_Counters(_OUT COUNTERS_t *ptCounters);
```

4.5.1 Details

This function provides all PoE devices communication statistics:

U32 ulSucceedMsgsCntr – counts the number of successful sent messages.

PoE API Error Counters:

- *U32 ulChecksum* – counts the number of times wrong checksum was received
- *U32 ulCommunicationTimeout* – counts the number of times no reply was received from the PoE device.
- *U32 ulRxRetry* – counts the number of times I2C PoE device return data as zero indicating that command is still being processed by the PoE device.
- *U32 ulEchoMismatch* – counts the number of times received message echo number defers from transmitted echo number.

PoE Device Error Counters:

- *U32 ulCommand_Received_Wrong_Checksum* – counts the number of times wrong checksum was received
- *U32 ulFailed_Execution_Conflict_in_Subject_Bytes* – counts the number of times the subject byte inside the 15 byte communication protocol was incorrect

- *U32 ulFailed_Execution_Wrong_Data_Byte_Value* – counts the number of times the data byte value inside the 15 byte communication protocol was incorrect
- *U32 ulFailed_Execution_Undefined_Key_Value* – counts the number of times the key byte value inside the 15 byte communication protocol was incorrect
- *U32 ulUnknownError* – general error

4.5.2 Arguments

- _ OUT COUNTERS_t *ptCounters: /* pointer to struct of type Counters_t */

4.5.3 Return Value

None

4.5.4 Example

```
COUNTERS_t tCounters;
MSCC_POE_Get_Counters (_OUT &tCounters);
```

5 Software Examples

To execute the included examples, you will need the following:

- **Hardware:**
  - PC running Linux Ubuntu or any other similar Linux distribution.
  - Microsemi PD-IM-7424A PoE evaluation board.
  - DIMAX SUB20 I2C over USB device + Sub20 I2C Cable adapter or
    Aardvark I2C over USB device + Aardvark I2C Cable adapter.
    Note: In case the I2C USB device failed to operate properly p**leas**e try other USB port.

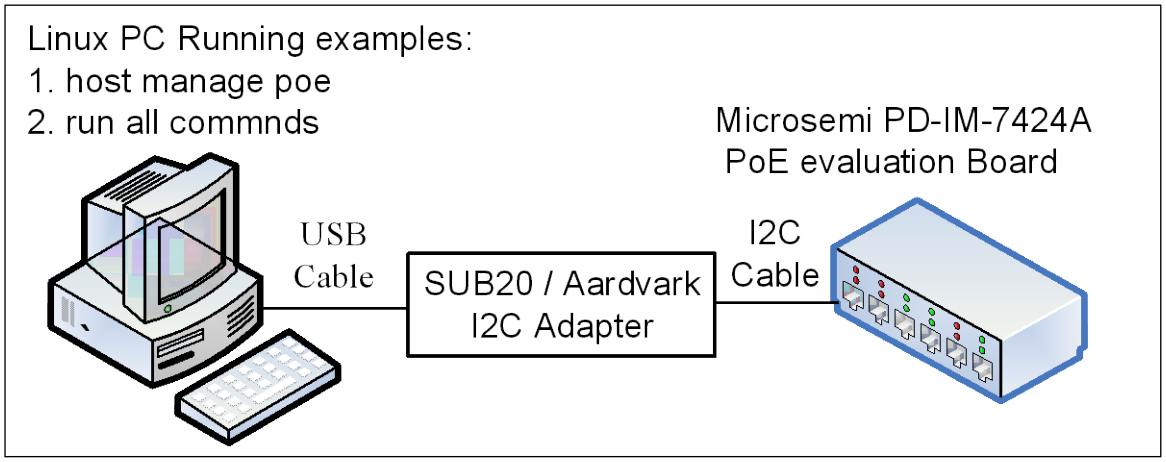

Figure 3: Examples 1 & 2 Setup

- **Software:**
  - Optional: The project was developed under eclipse IDE environment. Download and install Indigo Eclipse IDE from: eclipse-cpp-helios-SR2-linux-gtk for easy viewing editing and running purpose.
  - PoE Host software API that communicates with PoE Devices over an I2C interface. To test PoE API examples, you can use one of the following USB to I2C interfaces:
    - Sub20 : DIMAX SUB20 USB to I2C adapter
    - Aardvark: Total Phase USB to I2C interface

    To operate the examples with Sub20 or with the Aardvark you need to install the libusb-1.0 package which included in the delivered software package. Briefly, the follow shell commands should configure, build, and install the package:
    1. Extract the package: libsub-1.0.0.tar.bz2 to usr folder
    2. Open terminal and type: 'sudo su' to grant administrator access.
    3. Navigate to the extracted folder and type:
       './configure'
       'make'
       'make install'

    Note: the examples use the Sub20 interface as default. To change the defaults to Aardvark interface change the bSub20 variable in the example code to false:
    bSub20 = ePOE_FALSE;

- **Software example's – common resource:**
  Both examples use function named ***TransmitMsg*()** which works on top of *MSCC_POE_Msg()*. The function simplifies communication with PoE device by handling various communication errors as retransmission due to timeout, checksum error; PoE device reset detection, etc.

The return value of this function actually instructs the host what to do next.

eTransmitMsg_OK: message operation succeeds. Host can continue the operation.

eTransmitMsg_RestartManager: PoE device had reset event so PoE device unit needs to be configuring again.

eTransmitMsg_ExitManager: a fatal error was detected in PoE device controller so no communication can be done with the PoE device.

Although this function is out of the Microsemi Host PoE API software code, the software developer may use this function in his own software project in order to simplify communication error handling.

5.1 Example1 – *run all commands*

This example, named ***ex1_run_all_commands***, can be used to verify proper PoE and I2C setup by sending all PoE communication protocol commands while reporting command result.

Running Example 1:

1. Copy and Extract the 'poe_host_comm_workspace' folder to *./usr* folder.
2. Open terminal
3. Type 'sudo su' to grant administrator access
4. Navigate to the example folder: */usr/poe_host_comm_workspace/ex1_run_all_commands/*
5. Type: 'make clean'
6. Type: 'make'
7. Type *'./bin/ex1_run_all_commands'*

5.2 Example2 – *host manage PoE*

This example, named ***ex2_host_manage_poe***, stimulates host software managing PoE device by initializing PoE software, configure PoE device, and then periodically monitor PoE device functionality.

Running Example 2:

1. Copy and Extract the 'poe_host_comm_workspace' folder to *./usr* folder.
2. Open terminal
3. Type 'sudo su' to grant administrator access
4. Navigate to the example folder: */usr/poe_host_comm_workspace/ex2_host_manage_poe/*
5. Type: 'make clean'
6. Type: 'make'
7. Type *'./bin/ ex2_host_manage_poe'*

6 Integrating *poe_host_comm_api* with User Host Software

The following section describes how to integrate PoE API software with the user host software.

6.1 Basic Procedure

1. Add *architecture* and poe*_host_comm_api* software folders to the directory in which the user project is located.

2. Add user architecture dependent functions:

- **OS_Sleep_mS ()**: Sleep function, sleep value in milliseconds, minimum required range is 20 to 50 milliseconds with a resolution of 10 milliseconds.
- **OS_mutex_init ()**: Initializes the mutex.
- **OS_mutex_destroy ()**: Clean up a mutex that is no longer needed.
- **OS_mutex_lock ()**: Locks a mutex.
- **OS_mutex_unlock ()**: Unlocks or releases a mutex.

Notes:

- In cases where the architecture is different than Linux, you have to define the new architecture and implement the above functions by using the "**_NEW_ARCH_**" definition.
- In cases where the architecture has only one thread there is no need to modify the OS_mutex functions. In this case there is also no need to implement the mutex functions (OS_mutex_init (), OS_mutex_lock (), OS_mutex_unlock ()).

3. Open file *mscc_arcitecture/inc/mscc_arch_functions.h*, and modify the text marked in green.

```
//** mscc_arch_functions.h file. **//

/*============================================================================
/ Define here the Architecture
/============================================================================*/
#define _LINUX_PC_ARCH_
/*#define _NEW_ARCH_*/

#ifdef _LINUX_PC_ARCH_

#include <pthread.h>
#include <unistd.h>

static pthread_mutex_t sharedVariableMutex = PTHREAD_MUTEX_INITIALIZER;


/*-----------------------------------------------------------------------
 * description:     Sleep function
 * input :   sleepTime_mS  - sleep value in milliseconds
 *               minimum required range is: 20 milliseconds to 50 milliseconds
 *               with resolution of 10 milliseconds
 * output:   none
 * return:   e_POE_STATUS_OK             - operation succeed
 *           e_POE_STATUS_ERR_SLEEP_FUNCTION_ERROR - operation failed due to
                                                usleep function operation error
 *-----------------------------------------------------------------------*/
```

```
 S32 OS_Sleep_mS(U16 sleepTime_mS)
 {
     S32 status_number = 0;
     status_number = usleep(sleepTime_mS*1000);
     if(status_number != e_POE_STATUS_OK)
         return e_POE_STATUS_ERR_SLEEP_FUNCTION_ERROR;

     return e_POE_STATUS_OK;
 }


/*-------------------------------------------------------------------

 * description:     initialize the mutex
 * input :   none
 * output:   none
 * return:   e_POE_STATUS_OK                   - operation succeed
 *           e_POE_STATUS_ERR_MUTEX_INIT_ERROR - operation failed due to mutex
                                                 initialize operation error
 *-------------------------------------------------------------------*/
S32 OS_mutex_init()
{
    S32 status_number = 0;

    /* initializes the mutex */
    status_number = pthread_mutex_init(&sharedVariableMutex, NULL);
    if(status_number != e_POE_STATUS_OK)
        return e_POE_STATUS_ERR_MUTEX_INIT_ERROR;

    return e_POE_STATUS_OK;
}









/*-------------------------------------------------------------------
 *    description:     destroy the mutex - Clean up a mutex that is no longer
needed
 *
 *    input :   none
 *
 *    output:   none
 *
 *    return:   ePOE_STATUS_OK                    - operation succeed
```

```
 *              ePOE_STATUS_ERR_MUTEX_DESTROY_ERROR - operation failed due to
mutex initialize operation error
 *-------------------------------------------------------------------------*/
POE_STATUS_e OS_mutex_destroy()
{
    S32 lsStatus_number = 0;

    /* initializes the mutex */
    lsStatus_number = pthread_mutex_destroy(&sharedVariableMutex);
    if (lsStatus_number != ePOE_STATUS_OK)
        return ePOE_STATUS_ERR_MUTEX_DESTROY_ERROR;

    return ePOE_STATUS_OK;
}



/*-------------------------------------------------------------------------

 * description:  locking a mutex
 * input :   none
 * output:   none
 * return:   e_POE_STATUS_OK                    - operation succeed
 *           e_POE_STATUS_ERR_MUTEX_LOCK_ERROR - operation failed due to mutex
                                        lock operation error
*-------------------------------------------------------------------------*/
 S32 OS_mutex_lock()
 {
    S32 status_number = 0;
    /* lock the mutex. */
    status_number = pthread_mutex_lock(&sharedVariableMutex);
    if(status_number != e_POE_STATUS_OK)
        return e_POE_STATUS_ERR_MUTEX_LOCK_ERROR;

    return e_POE_STATUS_OK;
 }







/*-------------------------------------------------------------------------

 * description:     Unlocking or releasing a mutex
 * input :   none
 * output:   none
 * return:   e_POE_STATUS_OK                        - operation succeed
 *           e_POE_STATUS_ERR_MUTEX_UNLOCK_ERROR    - operation failed due to mutex
                                                      unlock operation error
 *-------------------------------------------------------------------------*/
 S32 OS_mutex_unlock()
```

```
 {
    S32 status_number = 0;
    /* Release the mutex. */
    status_number = pthread_mutex_unlock(&sharedVariableMutex);
    if(status_number != e_POE_STATUS_OK)
      return e_POE_STATUS_ERR_MUTEX_UNLOCK_ERROR;

    return e_POE_STATUS_OK;
 }

#elif defined NEW_ARCH

/*-------------------------------------------------------------------------
 * description:     Sleep function
 * input :   sleepTime_mS  - sleep value in milliseconds
 *               minimum required range is: 20 milliseconds to 50 milliseconds
 *               with resolution of 10 milliseconds
 * output:   none
 * return:   e_POE_STATUS_OK             - operation succeed
 *           e_POE_STATUS_ERR_SLEEP_FUNCTION_ERROR - operation failed due to
                                          usleep function operation error
 *-------------------------------------------------------------------------*/
 S32 OS_Sleep_mS(U16 sleepTime_mS)
 {
    S32 status_number = 0;

    /* TODO - implement here the function depending your architecture */

    return e_POE_STATUS_OK;
 }




/*-------------------------------------------------------------------------
 * description:     initialize the mutex
 * input :   none
 * output:   none
 * return:   e_POE_STATUS_OK                   - operation succeed
 *           e_POE_STATUS_ERR_MUTEX_INIT_ERROR - operation failed due to mutex
                                                 initialize operation error
 *-------------------------------------------------------------------------*/
 S32 OS_mutex_init()
 {
    S32 status_number = 0;

    /* TODO - implement here the function depending your architecture */

    return e_POE_STATUS_OK;
 }
```

```
/*-------------------------------------------------------------------
 *    description:     destroy the mutex - Clean up a mutex that is no longer
needed
 *
 *    input :   none
 *
 *    output:   none
 *
 *    return:   ePOE_STATUS_OK                      - operation succeed
 *              ePOE_STATUS_ERR_MUTEX_DESTROY_ERROR - operation failed due to
mutex initialize operation error
 *-------------------------------------------------------------------*/
POE_STATUS_e OS_mutex_destroy()
{
       S32 lsStatus_number = 0;

          /* TODO - implement here the function depending your architecture */
      #error OS_mutex_destroy function should be Implement.

          return ePOE_STATUS_OK;
}




/*-------------------------------------------------------------------
 * description:  locking a mutex
 * input :   none
 * output:   none
 * return: e_POE_STATUS_OK                   - operation succeed
 *         e_POE_STATUS_ERR_MUTEX_LOCK_ERROR - operation failed due to mutex lock
                                               operation error
 *         e_POE_STATUS_ERR_SLEEP_FUNCTION_ERROR - operation failed due to usleep
                                                   function operation error
 *-------------------------------------------------------------------*/
 S32 OS_mutex_lock()
 {
    S32 status_number = 0;

    /* TODO - implement here the function depending your architecture */

    return e_POE_STATUS_OK;
 }

/*-------------------------------------------------------------------
 * description:   Unlocking or releasing a mutex
 * input :   none
 * output:   none
 * return:   e_POE_STATUS_OK                     - operation succeed
 *           e_POE_STATUS_ERR_MUTEX_UNLOCK_ERROR   - operation failed due to mutex
                                                     unlock operation error
 *-------------------------------------------------------------------*/
```

```
 S32 OS_mutex_unlock()
 {
    S32 status_number = 0;

    /* TODO - implement here the function depending your architecture */

    return e_POE_STATUS_OK;
 }
#else
      #error UNSUPPORTED PLATFORM
#endif

/*============================================================================
/ End of Architecture Definition
/============================================================================*/
```

a. Add #define to the new architecture and unmark the existing #define Linux architecture

/ #define _LINUX_PC_ARCH_ */*

*#define _YOUR_NEW_ARCH_*

b. Rename the **_YOUR_NEW_ARCH_** with your architecture name and implement the function:

OS_Sleep_mS (), OS_mutex_init (),OS_mutex_lock(),OS_mutex_unlock()

depending on your architecture.

2. Customize the makefile to your own compiler and linker by modifying the above makefiles.

4. Execute *makefile* from within each project, which should create library files *mscc_arcitecture_lib.a*, *poe_host_comm_api/makefile*

- *mscc_arcitecture/makefile* creates lib file named *mscc_arcitecture_lib.a*.
- *poe_host_comm_api/makefile* creates lib file named *poe_host_comm_api_lib.a*.

6.2 Implementing I²C Read/Write Function Calls

PoE API software communicates with the PoE devices through the I²C interface. The software programmer has to Implement I²C read/write functions per the following functions prototypes.

Note: I2C read and write functions should return 0 for successful operation, and negative values for various potential errors.

6.2.1 Pointer for Driver Write Function

This function writes a stream of bytes to the I²C PoE device. The return value of the function is a status code

```
typedef S32 (*mscc_FPTR_Write)(U8 I2C_Address, const U8* Txdata, U16
num_write_length);
```

- **_IN U8 I2C_Address**: IC's I²C address.
- **_IN const U8* pTxdata**: The data bytes array to be transmitted.
- **_IN U16 num_write_length**: Number of data bytes to be transmitted.

Below is an example of Sub20 Device I²C Write function usage:

```
S32 Sub20_Write(U8 I2C_Address,  const U8* Txdata, U16 num_write_length)
{
    int i;

    for (i=0;i<num_write_length;i++)
        sn_buf[i] = Txdata[i];

    return sub_i2c_write( fd, I2C_Address,0,0, sn_buf, num_write_length );
}
```

6.2.2 Pointer for Driver Read Function

This function reads a stream of bytes from the I²C PoE device. This function returns the data bytes read into the **pRxdata** variable. The return value of the function is a status code.

```
typedef S32 (*mscc_FPTR_Read)(U8 I2C_Address,U8* pRxdata, U16
number_of_bytes_to_read);
```

- **_IN U8 I2C_Address**: IC's I²C address.
- **_OUT U8* pRxdata:** pointer to bytes array where to fill the received data.
- **_IN U16 number_of_bytes_to_read**: Number of bytes to read

Below is an example of the I²C Read function usage:

```
S32 Sub20_Read (U8 I2C_Address,U8* Rxdata, U16 number_of_bytes_to_read)
{
    memset(sn_buf,250,15*sizeof(char));
    S32 ulStatusCode = sub_i2c_read( fd, I2C_Address,0,0, sn_buf,
number_of_bytes_to_read);

    if (ulStatusCode == eSub20Status_OK)
    {
        int i;
        for (i=0;i<number_of_bytes_to_read;i++)
            Rxdata[i] = sn_buf[i];
    }

    return ulStatusCode;
}
```

6.3 Simplifying the API Implementation

To simplify the API implementation, include the "**mscc_poe_host_comm_api.h**" file in your project. This H file contains the necessary POE globals and types described at *poe_host_comm_api\mscc_poe_global_types.h*.

Revision History

Revision Level / Date	Para. Affected	Description
1.0 / Sep 2012	-	Initial release
1.1 December 2012		Adding MSCC_POE_Clear_POE_Device_Buffer API command

For support contact: sales_AMSG@microsemi.com

Visit our web site at: www.microsemi.com **Catalog Number:** POE_HOST_SW_UG