Power over Ethernet

Microsemi PoE Application Program Interface (API)

User Guide

Revision 1.1

Catalog Number 06-0054-056

Table of Contents

1 Introduction

Microsemi's® PoE Application Program Interface (API) simplifies integration and migration to Microsemi PoE PD690xx family PoE ICs operating in Auto Mode. Microsemi customers already using the 15 byte PoE communication protocol as per PD69000/G release 2.0.x should find it easy to migrate from the PD630xx ICs in Enhanced Mode to PD690xx PoE ICs in Auto Mode.

The PoE API software provides Enhanced Mode features (not available in Auto Mode) such as LLDP and Layer-2 Power management and port matrix.

The PoE API is compliant with the 15 byte communication protocol for PD69000/G release 2.0.x (refer to *User Guide - PD63000 & PD69000/G Serial Communication Protocol*, Catalog Number 06-0032-056).

The PoE API communicates with PD690xx ICs in the Auto Mode configuration and reflects enhanced mode operation as described in the Serial Communication Protocol.

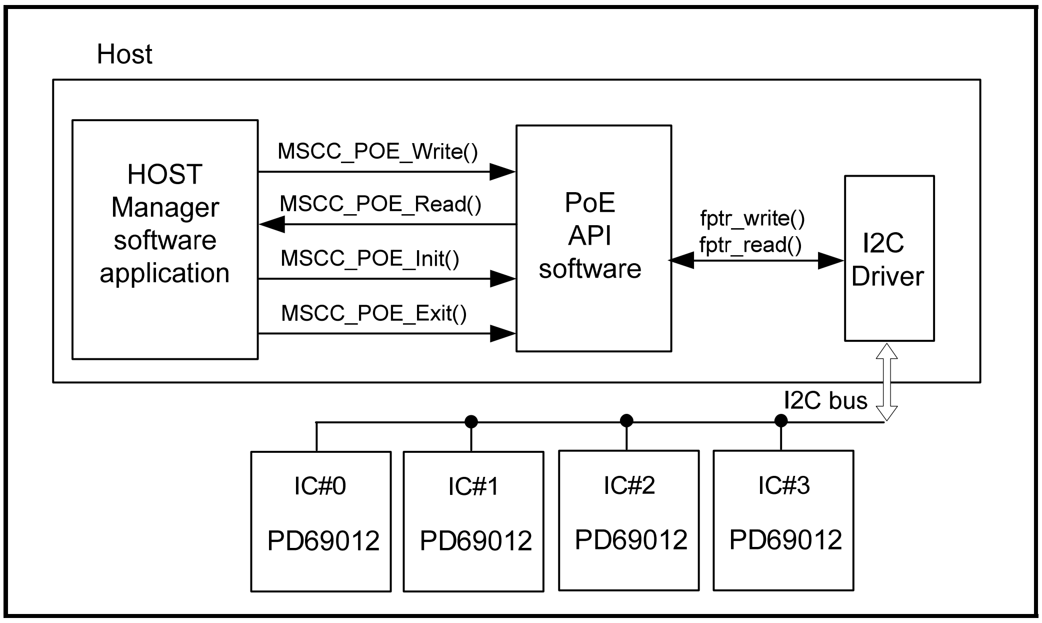


Figure 1: General System

After initializing the API by calling MSCC_POE_Init(), the user has to:

- Build 15 byte command message
- Call the function *MSCC_POE_Write()*, which will process user requests by accessing various PD690xx ICs internal registers.
- Call *MSCC_POE_Read()* to obtain 15 byte returned data as per the Serial Communication Protocol user guide.

By using the Microsemi simplified API, the software programmer has no need to learn all the PD690xx internal registers.

Function *MSCC_POE_Exit()* should be called whenever the user software wishes to terminate communications with Microsemi PoE API.

The PoE API software supports the **DEVICE DISCOVERY PROTOCOL – LLDP** (Link Layer Discovery Protocol).

Power discovery enables switches and phones to convey power information, an especially important capability when Power over Ethernet (PoE) is used.

LLDP provides information related regarding how the device is powered (from the line, from a backup source, from external power source, etc.), power priority (how important is it that this device receives power?), and how much power the device needs.

The LLDP implementation is designed to store the relevant changes that enable combining this information with the current power management capabilities. The host is only required to add additional communication (the host does not require power management logic).

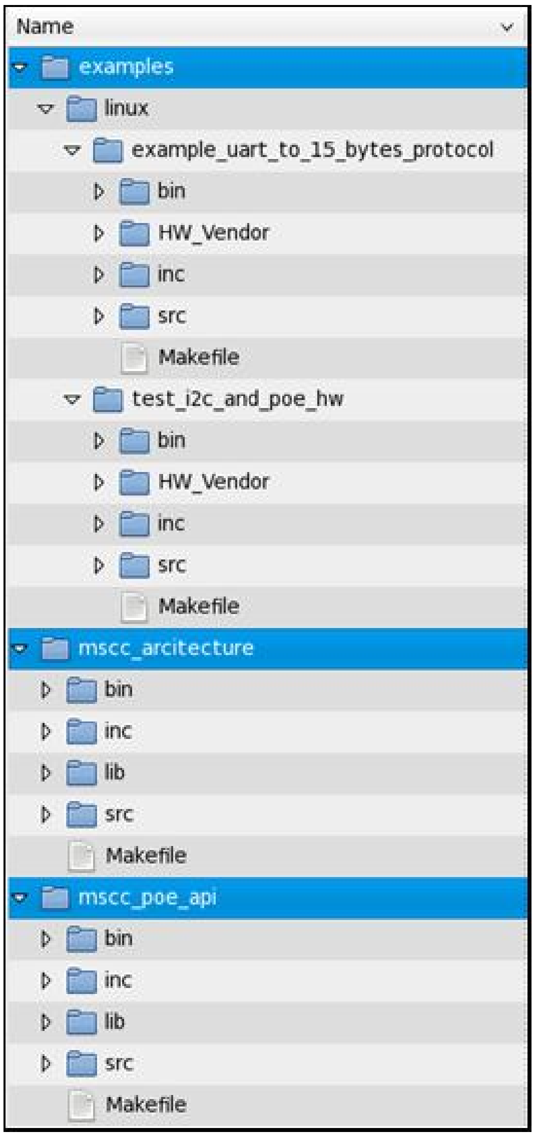
Notes for Layer 2:

- LLDP and Layer2 power management features are activated only when the calculation method of power management is static (according to class or PPL) and the Port Power Limit (Icut) level set to the maximum.
- Configuration mask to enable / disable Layer 2 operation is enabled by default.
- Configuration mask to enable / disable priority definition by PD is disabled by default.

2 Software Distribution Structure

The software package is made up of three sections:

- **PoE API**: A (mscc_poe_api) software package which is **ANSI-C** compliant without any operating system correlation or special requirements (excluding the delay function which should be implemented under the architecture section). The make file is optimized for GNU GCC. Typing ‘make’ will compile all files into a PoE library named *mscc_poe_api.a*, which will be linked by the example code into an executable application.
- **Architecture**: (mscc_arcitecture) This folder contains all the functions that PoE API and examples may require to operate over a specific operating system (as system delay). The make file 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.
- **Examples**: Software distribution contains two Linux based examples. Each example is linked with *mscc_poe_api_lib.a* and *mscc_arcitecture_lib.a* libraries. The examples were created and tested on Linux Fedora Core 9 distribution.



```
Name
examples
  linux
    example_uart_to_15_bytes_protocol
      bin
      HW_Vendor
      inc
      src
      Makefile
    test_i2c_and_poe_hw
      bin
      HW_Vendor
      inc
      src
      Makefile
mscc_arcitecture
  bin
  inc
  lib
  src
  Makefile
mscc_poe_api
  bin
  inc
  lib
  src
  Makefile
```

Figure 2: PoE API tree structure

3 Code Data Types

PoE software API uses the following code conventions.

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

4 Code Functions Return Value

All PoE API user interface functions return the same enum from type *MSCC_POE_STATUS_e*. Zero is used as "OK", and all negative values represent various errors (see the list bellow).

```
typedef enum
{
    e_POE_STATUS_OK                                        =  0,
    e_POE_STATUS_ERR_POE_API_SW_INTERNAL                   = -1,
    e_POE_STATUS_ERR_COMMUNICATION_DRIVER_ERROR            = -2,
    e_POE_STATUS_ERR_COMMUNICATION_REPORT_ERROR            = -3,
    e_POE_STATUS_ERR_HOST_COMM_MSG_LENGTH_MISMATCH         = -4,
    e_POE_STATUS_ERR_PORT_NOT_EXIST                        = -5,
    e_POE_STATUS_ERR_MSG_NOT_READY                         = -6,
    e_POE_STATUS_UNKNOWN_ERR                               = -7,
    e_POE_STATUS_ERR_TIMER_INTERVAL_ERROR                  = -8,
    e_POE_STATUS_ERR_MUTEX_INIT_ERROR                      = -9,
    e_POE_STATUS_ERR_MUTEX_LOCK_ERROR                      = -10,
    e_POE_STATUS_ERR_MUTEX_UNLOCK_ERROR                    = -11,
    e_POE_STATUS_ERR_SLEEP_FUNCTION_ERROR                  = -12,
    e_POE_STATUS_ERR_MAX_ENUM_VALUE                        = -13,
}mscc_POE_STATUS_e;
```

5 User PoE API Software Interface Description

The following five software functions control the PoE API:

- MSCC_POE_Init()
- MSCC_POE_Write()
- MSCC_POE_Read()
- MSCC_POE_Exit()
- MSCC_POE_Timer_Tick()

5.1 MSCC_POE_Init()

```
MSCC_POE_STATUS_e MSCC_POE_Init(_IN mscc_InitInfo_t *pInitInfo,_OUT S32
*pDevice_error)
```

5.1.1 Details

This function initializes *MSCC PoE API*. The *mscc_InitInfo_t* structure pointer defines PoE hardware type, I2C driver read and write functions. Up to eight PoE ICs are supported

5.1.2 Arguments

- mscc_InitInfo_t *pInitInfo:
- mscc_InitInfo_t structure contains the followed members:
  - U8 IC_Address[MAX_ASIC_ON_BOARD] : I2C Address for each PoE IC. No existing IC should be marked by 0xFF

Note

Fill I2C address from IC_Address[0], up to the number of ICs and set the rest to 0xFF.

  - U8 NumOfExpectedChannesInIC[MAX_ASIC_ON_BOARD]: Number of PoE ports for each IC.

  Use one of the following options:

    - ASIC_8_CHANNELS: 8 PoE channels
    - ASIC_12_CHANNELS - 12 PoE channels
    - ASIC_NONE_CHANNELS - None

  Note

  Fill I2C address from IC_Address[0], up to number of IC's and set the rest to 0xFF.

  - U8 NumOfActiveICsInSystem: Number Of PoE ICs In the System
  - mscc_FPTR_Write fptr_write: I2C driver write function pointer (this function should be implemented by the user). I2C function pointer has the following format:

    *typedef S32 (*mscc_FPTR_Write)(_IN U8 I2C_Address, _IN const U8* pTxdata,_IN U16 num_write_length,_IN void* pUserParam);*

    _IN U8 I2C_Address: IC's I2C address.

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

    _IN U16 num_write_length: number of bytes to be transmitted.

    _IN void* pUserParam: value to be passed by PoE software API to I2C driver for each I2C write access.

- mscc_FPTR_Read fptr_read: I2C driver read function pointer (this function should be implemented by the user). I2C function pointer has the following format:

  *typedef S32 (*mscc_FPTR_Read)(_IN U8 I2C_Address,_OUT U8* pRxdata,_IN U16 length,_IN void* pUserParam);*

  _IN U8 I2C_Address: IC's I2C address.

  _IN U8* pRxdata: pointer where driver should place received I2C data.

  _IN U16 length: number of received bytes.

  _IN void* pUserParams: value to be passed by PoE software API to I2C driver for each I2C read access

- void *pUserParams: value to be passed by PoE software API to I2C driver for each I2C read/write access

▪ OUT S32 *pDevice_error: Pointer where to place user I2C driver returned error code during init stage.

5.1.3 Return Value

MSCC_POE_STATUS_e

5.1.4 Example

```
/*===========================================================================
/ Init here the InitInfo struct
/===========================================================================*/

mscc InitInfo t mscc InitInfo;

/* type the IC's I2C addresses */
mscc InitInfo.IC Address[0] = 0x30;
mscc_InitInfo.IC_Address[1] = 0x31;
mscc_InitInfo.IC_Address[2] = 0xFF;
mscc_InitInfo.IC_Address[3] = 0xFF;
mscc InitInfo.IC Address[4] = 0xFF;
mscc InitInfo.IC Address[5] = 0xFF;
mscc InitInfo.IC Address[6] = 0xFF;
mscc InitInfo.IC Address[7] = 0xFF;

/* type the IC's Expected number of ports                *
 * ASIC 12 CHANNELS           - for   12 ports in IC    *
 * ASIC 8 CHANNELS            - for    8 ports in IC    *
 * ASIC NONE CHANNELS         - for none ports in IC    */

mscc_InitInfo.NumOfExpectedChannesInIC[0] = ASIC_12_CHANNELS;
mscc_InitInfo.NumOfExpectedChannesInIC[1] = ASIC_12_CHANNELS;
mscc InitInfo.NumOfExpectedChannesInIC[2] = ASIC NONE CHANNELS;
mscc InitInfo.NumOfExpectedChannesInIC[4] = ASIC NONE CHANNELS;
mscc InitInfo.NumOfExpectedChannesInIC[5] = ASIC NONE CHANNELS;
mscc InitInfo.NumOfExpectedChannesInIC[6] = ASIC NONE CHANNELS;
mscc InitInfo.NumOfExpectedChannesInIC[7] = ASIC NONE CHANNELS;


/* type the number of active ICs in the system */
mscc_InitInfo.NumOfActiveICsInSystem = 2;

/* type the pointer for the functions which read and write I2C */
mscc_InitInfo.fptr_write= Aardvark_Write;/* pointer for Writing driver function */
mscc InitInfo.fptr read = Aardvark Read; /* pointer for Reading driver function */


/*===========================================================================
/ End Init InitInfo struct
/===========================================================================*/
```

5.2 MSCC_POE_Write()

```
MSCC POE STATUS e MSCC POE Write( IN U8* pTxdata, IN U16 num write length, OUT S32
*pDevice_error);
```

5.2.1 Details

This function writes 15 bytes PoE communication protocol messages from the Host to the PoE API.

5.2.2 Arguments

- _IN U8* pTxdata: Pointer to 15 bytes message data array to be transmitted
- _IN U16 num_write_length: Number of bytes to write (must be 15).
- OUT S32 *pDevice_error: Pointer where to place user I2C driver returned error code during write operation.

5.2.3 Return Value

MSCC_POE_STATUS_e

5.2.4 Example

```
MSCC_POE_STATUS_e mscc_poe_status;
S32 device_error;

/* Get System status request*/
U8 data_out[] = { B_Request, 0x02, B_Global, B_SystemStatus, B_Space, B_Space,
B_Space, B_Space, B_Space, B_Space, B_Space, B_Space, B_Space, 0x03, 0x04 };

mscc_poe_status = MSCC_POE_Write(data_out, 15,&device_error);
if(mscc_poe_status != POE_STATUS_OK)
{
//Error occurred
}
```

5.3 MSCC_POE_Read()

```
MSCC_POE_STATUS_e MSCC_POE_Read (_OUT U8* pRxdata,_IN U16 num_read_length, _OUT
S32 *pDevice_error);
```

5.3.1 Details

This function reads the 15 bytes PoE communication protocol message from the PoE API to the Host.

5.3.2 Arguments

- _IN U8* pRxdata: 15 bytes message data array.
- _IN U16 num_read_length: Number of bytes to read/message length (must be 15).
- OUT S32 *pDevice_error: Contains the error code of the host driver in case of errors in I2C read or I2C write operations.

5.3.3 Return Value

MSCC_POE_STATUS_e

5.3.4 Example

```
U8 data_in[BUFFER_SIZE];
MSCC_POE_STATUS_e mscc_poe_status;
S32 device_error;

mscc_poe_status = MSCC_POE_Read(data_in, 15,&device_error);
if(mscc_poe_status != POE_STATUS_OK)
{
   //Error occurred
}
```

5.4 MSCC_POE_Exit ()

```
MSCC_POE_STATUS_e MSCC_POE_Exit(_IN mscc_CloseInfo_t *pMscc_CloseInfo, _OUT S32
*pDevice_error)
```

5.4.1 Details

This command closes the PoE software operation.

Note:
Currently it is not necessary to call the *MSCC_POE_Exit()* API function (The command is for future use).

5.4.2 Arguments

- **_IN mscc_CloseInfo_t *pMscc_CloseInfo**: Pointer to struct mscc_CloseInfo_t which contains data required for closing the PoE API software.
- **OUT S32 *pDevice_error**: Contains the error code of the host driver in case of errors in I2C read or I2C write operations.

5.4.3 Return Value

MSCC_POE_STATUS_e

5.4.4 Example

```
MSCC POE STATUS e mscc poe status;
S32 device error;

mscc CloseInfo t *pMscc CloseInfo;
mscc poe status = MSCC POE Exit( IN pMscc CloseInfo, OUT &device error);
if(mscc poe status != POE STATUS OK)
{
   //Error occurred
}
```

5.5 MSCC_POE_Timer_Tick ()

```
mscc POE STATUS e MSCC POE Timer Tick( IN U8 IntervalTime Sec, OUT S32
*pDevice_error);
```

5.5.1 Details

The purpose of this function is to apply the LLDP and Layer 2 Power Management functionality. This function should be called by the Host every 1 second.

5.5.2 Arguments

- **_IN U8 IntervalTime_Sec**: The interval of the timer, which - must be 1 second.
- **_OUT S32 *pDevice_error**: Points where to place user I^{2}C driver returned error code during write operation.

5.5.3 Return Value

mscc_POE_STATUS_e

5.5.4 Example

```
mscc POE STATUS e mscc poe status;
S32 device error;

void Tick_handler ( void *ptr )
{
   #define L2 INTERVAL TIME 1  /* seconds */

   S32 device error;                                            /* I2C
device error number */
   mscc POE STATUS e mscc poe status e;      /* microsemi PoE status number   */

    while (POE TRUE)
    {
      OS Sleep mS (L2 INTERVAL TIME);   /* sleep for L2 INTERVAL TIME (1 second)
*/

      /* call for API function MSCC_POE_Timer_Tick every 1 second */
      mscc poe status e = MSCC POE Timer Tick ( IN L2 INTERVAL TIME , OUT
&device error);
      if(mscc poe status e != e POE STATUS OK)
      {
//Error occur
      }
    }
}
```

6 Software Examples

To run Microsemi PoE API examples, the following items are required:

- A PC running Linux Fedora Core 9 or any similar Linux distribution. Verify that a GCC compiler and Eclipse IDE are installed (Eclipse IDE allows easy source code modification).

  Note To run the example "*example_poe_comm_protocol_engine*", verify that the PC communication port **ttyS0 (COM1)** is available. To check whether **ttyS0** is available, open a terminal shell, and type the following command:

  *[root@localhost ~]# setserial -g /dev/ttyS**

  A typical report should be similar to the report below:

  /dev/ttyS0, UART: 16550A, Port: 0x03f8, IRQ: 4

- PoE software API that communicates with PoE ICs over an I2C interface. To test PoE API examples, obtain the Total Phase USB to I2C interface named Aardvark from the following URL:

  http://www.totalphase.com/products/aardvark_i2cspi/

- A Microsemi PoE evaluation board, PN part number **PD-IM-7424A**.
- For the second example, it is recommended that you:
  - Install PoE manager Enhanced Mode (SS-0050-00N) GUI on a second PC running Windows XP/Vista.
  - Connect cross RS232 cable between Windows PC to Linux PC communication port.

In the second example, you can use the PoE manager GUI to translate a command into a 15 byte PoE protocol command. The example receives the command and it will call the PoE API driver to modify the PoE functionality.

Note The following two examples demonstrate a communication setup between two PD69012 PoE ICs using I2C address 0x30, 0x31.

6.1 Example 1: Description

Example 1, named **example_poe_comm_protocol**, can be used to verify proper setup of PoE hardware and I2C connectivity proper compilation. The example sends the "Get System Status" 15 byte PoE protocol command to the PoE API driver, and returns the telemetry in response.

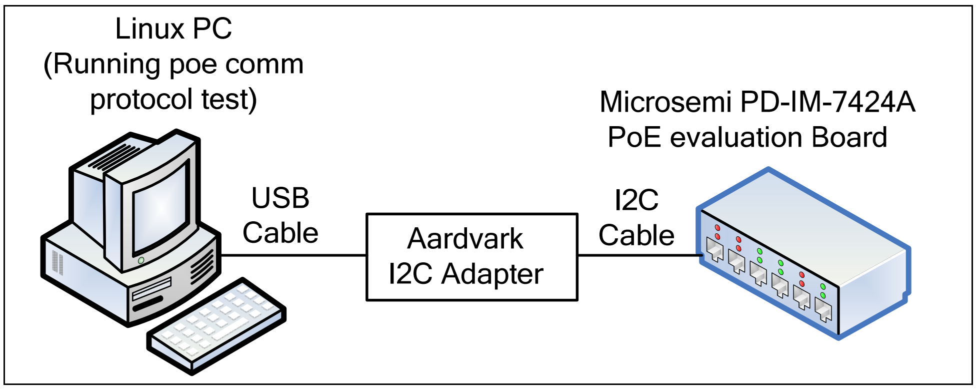

Linux PC
(Running poe comm protocol test)
USB Cable
Aardvark I2C Adapter
I2C Cable
Microsemi PD-IM-7424A PoE evaluation Board

Figure 3: Example 1 Setup

Running the Example

1. Extract and copy PoE API source code files to *./usr* folder.
2. Navigate to the example folder:
   /usr/poi_api_projects/examples/linux/test_i2c_and_poe_hw/
3. Execute:
   ./bin/example_poe_comm_protocol

The example printout should look like the following printout:

```
[root@localhost /]# cd usr/poi api projects/examples/linux/test i2c and poe hw/
[root@localhost test i2c and poe hw]# ./bin/example poe comm protocol
Writing 15Bytes:   2     2     7     3d    0     0     0     0     0     0     0
      0     0     0     48

Reading 15Bytes:   3     2     0     0     0     0     0     0     fc    fc    fc
      0     0     2     f9

[root@localhost test_i2c_and_poe_hw]#
```

6.2 Example 2: Description

This example, named **example_poe_comm_protocol_engine**, waits for a 15 byte PoE protocol command on PC UART communication ttyS0 (COM1), at a baud rate of 19200. The received 15 byte command is then forwarded to the PoE API, which is in communication with the PoE ICs. An answer is sent from the PoE software through the UART communication.

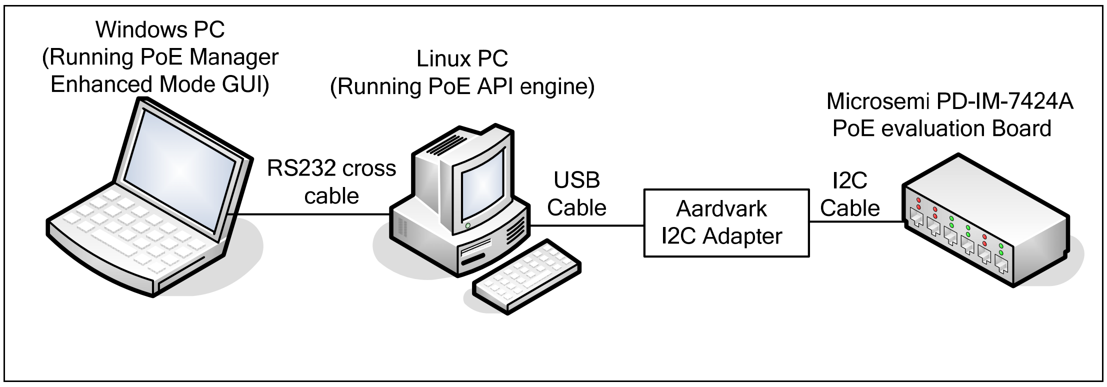


Figure 4: Example 2 Setup

Running the Example:

1. Extract and copy the PoE API source code files to:
   ./usr folder
2. Navigate to the example folder:
   /usr/poi_api_projects/examples/linux/example_uart_to_15_bytes_protocol/
3. Execute command:
   ./bin/example_poe_comm_protocol_engine

The example printout should look like the following printout:

```
[root@localhost /]#
cd usr/poi api projects/examples/linux/example uart to 15 bytes protocol/
[root@localhost example uart to 15 bytes protocol]#
./bin/example_poe_comm_protocol_engine
```

```
Writing 15Bytes:   2     62    5     25    4     4e    4e    4e    4e    4e    4e
      4e    4e    3     2

Reading 15Bytes:   3     62    0     0     0     0     0     0     4e    0     0
      4e    4e    1     4f

Writing 15Bytes:   2     64    5     25    6     4e    4e    4e    4e    4e    4e
      4e    4e    3     6

Reading 15Bytes:   3     64    0     0     0     3     0     0     4e    0     0
      4e    4e    1     54
```

7 Integrating PoE API Software with Host Software

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

7.1 Basic Procedure

1. Add architecture and PoE software API folders to the directory in which the user project is located.
   Note: PoE software API makefiles were optimized for the GNU GCC compiler.
2. Create a lib file which you can link with the entire project.
   - *mscc_arcitecture/makefile* creates lib file named *mscc_arcitecture_lib.a*.
   - *mscc_poe_api /makefile* creates lib file named *mscc_poe_api_lib.a*.

   Customize the makefile to your own compiler and linker by modifying the above makefiles.
3. Add user architecture dependent functions:
   - OS_Sleep_mS ()
   - OS_mutex_init ()
   - OS_mutex_lock ()
   - OS_mutex_unlock ()

   Note: In cases where the architecture is different than Linux, you have to define the new architecture and implement the above functions.
4. Open the following file:
   mscc_arcitecture/inc/mscc_arch_functions.h. Modify the text marked in yellow as needed.

```
// mscc arch functions.h file.

/*=======================================================================
/ Define here the Architecture
/=====================================================================*/

#define  LINUX PC ARCH
/*#define  YOUR 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 mili Seconds
 *                minimum required range is: 20 mili seconds to 50 mili seconds
 *                with resolution of 10 mili second
 *
 * 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:  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;
    /* 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( YOUR NEW ARCH )


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

 * description:     Sleep function
 *
 * input :   sleepTime_mS  - sleep value in mili Seconds
 *                minimum required range is: 20 mili seconds to 50 mili seconds
 *                with resolution of 10 mili second
 *
 * 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:  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 your architecture.

7.2 Implementing I2C Read/Write Function Calls

PoE API software communicates with the PD690XX ICs through I2C interface. Implement I2C read/write functions per the following functions prototype. Update function **MSCC_POE_Init()** I2C read/write function call pointers with the names used by your code.

One input parameter of the read/write functions is the **pUserParams**.

The pUserParams enables the user to send additional input information (input information is information that was sent during the initialization process) to the driver functions that he implemented for read and write I2C operations.

pUserParams information is sent in each Write and Read I2C operation.

When there is no needed to send any additional information, set the pUserParams to NULL.

7.2.1 Pointer for Writing Driver Function

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

```
typedef S32 (*mscc FPTR Write)( IN U8 I2C Address,  IN I2cFlags Flags, IN const
U8* pTxdata,_IN U16 num_write_length,_IN void* pUserParam);
```

- **_IN U8 I2C_Address**: IC's I2C address.
- **_IN I2cFlags Flags**: Operation mode.

  Implement I2C two operation modes as per the enum below for the write function:

```
   enum I2cFlags {
            I2C 7 BIT ADDR WITH STOP CONDITION    = 0x00,
            I2C 7 BIT ADDR WITHOUT STOP CONDITION = 0x04
            };
```

- **_IN const U8* pTxdata**: The data bytes array to be transmitted.
- **_IN U16 num_write_length**: Number of data bytes to be transmitted.
- **_IN void* pUserParam**: Parameter to be passed to I2C driver.

Below is an example of the I2C Write function usage:

```
/*------------------------------------------------------------------------
 *    description: Write data byte array to the IC
 *

 *    input :      I2C Address
 *                 I2cFlags Flags            - I2C stop condition mode
 *                 pTxdata                   - data byte array to transmit
 *                 num write length          - number of bytes to write to the IC
 *                 pUserParam                - user data
 *    output:      none
 *    return:      POE STATUS OK             - operation succeed
 *                 != POE STATUS OK          - operation failed
 *-----------------------------------------------------------------------*/
S32 mscc IC COMM I2C Write( IN U8 I2C Address, IN U16 RegisterAddress,   IN const
U8* pTxdata,  IN U16 number of bytes to write)
{
      U8 TxDataArr[number of bytes to write+2];

      TxDataArr[0] = mscc_GetFirstByte( RegisterAddress);
      TxDataArr[1] = mscc_GetSecondByte( RegisterAddress);
      U8 i;
      for (i = 0; i < number of bytes to write; i++)
          TxDataArr[i+2] = pTxdata[i];

*pDeviceErrorInternal = mscc fptr write (I2C Address,
I2C 7 BIT ADDR WITH STOP CONDITION, TxDataArr,  number of bytes to write+2,
mscc pUserData);

    if(*pDeviceErrorInternal != POE STATUS OK)
        result = POE STATUS ERR COMMUNICATION DRIVER ERROR;

      return result;
}
```

7.2.2 Pointer for Reading Driver Function

This function reads a stream of bytes from the I2C slave 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)( IN U8 I2C Address, IN I2cFlags Flags, OUT U8*
pRxdata,_IN U16 length,_IN void* pUserParam);
```

- **_IN U8 I2C_Address**: IC's I2C address.
- **_IN I2cFlags Flags**: Operation mode.

Implement I2C two operation modes as per the enum below for the read function:

```
   enum I2cFlags {
            I2C 7 BIT ADDR WITH STOP CONDITION    = 0x00,
            I2C 7 BIT ADDR WITHOUT STOP CONDITION = 0x04
            };
```

- **_IN U16 length**: Number of received bytes.
- **_IN void* pUserParam**: Parameter to be passed to I2C driver.

Below is an example of the I2C Read function usage:

```
/*----------------------------------------------------------------------
 *    description: Read data byte array from IC
 *    input :   I2C Address
 *              RegisterAddress             - address of IC register
 *              number of bytes to read     - number of bytes to read from the IC
 *    output:   Rxdata                      - received data byte array
 *    return:   POE STATUS OK               - operation succeed
 *              != POE STATUS OK                  - operation failed
 *---------------------------------------------------------------------*/
S32 mscc IC COMM I2C Read ( IN U8 I2C Address, IN U16 RegisterAddress, OUT  U8*
pRxdata,  IN U16 number of bytes to read)
{
                        /* High regAddr                      low regAddr */
U8
data out[]={mscc GetFirstByte(RegisterAddress),mscc GetSecondByte(RegisterAddress)
}
result =
mscc_fptr_write(I2C_Address,I2C_7_BIT_ADDR_WITHOUT_STOP_CONDITION,data_out,
2,mscc pUserData);

*pDeviceErrorInternal = mscc fptr read (I2C Address,
I2C_7_BIT_ADDR_WITH_STOP_CONDITION,
pRxdata,number of bytes to read,mscc pUserData);

if(*pDeviceErrorInternal != POE STATUS OK)
    result = POE STATUS ERR COMMUNICATION DRIVER ERROR;

return result;
}
```

7.3 Obtaining LLDP and Layer 2 Power Management Functionality

To obtain LLDP and Layer 2 Power Management functionality, the host calls the API function: **MSCC_POE_Timer_Tick ()** every 1 second.

Note: LLDP and Layer 2 Power Management functionality can be achieved only if the power management mode is set to static (calculation method of power management).

7.4 Simplifying the API Implementation

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

7.5 PoE API Functions Usage

```
#include "mscc poe api.h"

void Tick handler ( void *ptr );

Void main()
{
S32 device error; // used to save driver errors if occurs.  /* contain I2C device
error number     */
mscc POE STATUS e mscc poe status;    /* contain microsemi poe status number
*/
mscc InitInfo t mscc InitInfo;           /* Initialization PoE API software
Information struct       */
mscc CloseInfo t  mscc CloseInfo; /* Closing PoE API software Information struct
*/

/*=======================================================
/ Init here by using InitInfo struct
/=======================================================*/

/* type the IC's I2C addresses */
mscc InitInfo.IC Address[0] = 0x30;
mscc InitInfo.IC Address[1] = 0x31;
mscc InitInfo.IC Address[2] = 0xFF;
mscc InitInfo.IC Address[3] = 0xFF;
mscc InitInfo.IC Address[4] = 0xFF;
mscc InitInfo.IC Address[5] = 0xFF;
mscc_InitInfo.IC_Address[6] = 0xFF;
mscc InitInfo.IC Address[7] = 0xFF;


/* type the IC's Expected number of ports               *
 * ASIC_12_CHANNELS           - for   12 ports in IC
 * ASIC 8 CHANNELS                          - for    8 ports in IC
 * ASIC NONE CHANNELS         - for none ports in IC    */

mscc InitInfo.NumOfExpectedChannesInIC[0] = ASIC 12 CHANNELS;
mscc InitInfo.NumOfExpectedChannesInIC[1] = ASIC 12 CHANNELS;
mscc InitInfo.NumOfExpectedChannesInIC[2] = ASIC NONE CHANNELS;
mscc InitInfo.NumOfExpectedChannesInIC[4] = ASIC NONE CHANNELS;
mscc InitInfo.NumOfExpectedChannesInIC[5] = ASIC NONE CHANNELS;
mscc InitInfo.NumOfExpectedChannesInIC[6] = ASIC NONE CHANNELS;
mscc_InitInfo.NumOfExpectedChannesInIC[7] = ASIC_NONE_CHANNELS;

/* Type the number of active ICs in the system */
mscc_InitInfo.NumOfActiveICsInSystem = 2;
/* type the pointer for the functions which read and write I2C */
mscc InitInfo.fptr write = Aardvark Write;/* pointer for Writing driver function*/
mscc InitInfo.fptr read = Aardvark Read; /* pointer for Reading driver function*/

/*=====================================================
/ End Init InitInfo struct
/=====================================================*/


//Init PoE API software by using InitInfo struct
mscc_poe_status = MSCC_POE_Init(&mscc_InitInfo, &device_error);
if(mscc poe status != POE STATUS OK)
{
    MSCC_POE_UTIL_DumpErrorResultDataToTheScreen(_IN "Init",_IN
mscc POE STATUS e, IN device error);
     goto FINISH;
}

/* Create a new thread for Tick handler function */
pthread create (&thread a, NULL, (void *) &Tick handler, (void *) &i[0]);

/* GetSystemstatus - 15 bytes PoE protocol message */
U8 static U8 protocol_data_out[] = { 0x02, 0x00, 0x07, 0x3D, 0x4E, 0x4E, 0x4E,
0x4E, 0x4E, 0x4E, 0x4E, 0x4E, 0x4E, 0x3, 0x4 };
U8 data in[15];

// write data to PoE API software
mscc poe status = MSCC POE Write(data out, 15, &device error);
if(mscc poe status != POE STATUS OK)
{
     MSCC POE UTIL DumpErrorResultDataToTheScreen( IN "Write", IN
mscc POE STATUS e, IN device error);
     goto FINISH;
}
/* Read the 15 byte protocol msg reply from the PoE API */
mscc poe status = MSCC POE Read(data in, 15, &device error);
if(mscc_poe_status != POE_STATUS_OK)
{
     MSCC_POE_UTIL_DumpErrorResultDataToTheScreen(_IN "Reading",_IN
mscc POE STATUS e, IN device error);
      printf("%s\n\n",Aardvark status string(device error));
}

/* Close PoE API software resources*/
mscc poe status = MSCC POE Exit( IN &mscc CloseInfo,  OUT &device error);
if(mscc poe status!= POE STATUS OK)
{
      MSCC POE UTIL DumpErrorResultDataToTheScreen( IN "Exit", IN
mscc_POE_STATUS_e,_IN device_error);
      printf("%s\n\n",Aardvark status string(device error));
 }
}



/*---------------------------------------------------------------------
 *    description:    This is a thread routine, call for API function
MSCC POE Timer Tick every 1 second.
 *
 *    input :   *ptr  - argument to threads
 *    output:   none
 *    return:   none
 *-------------------------------------------------------------------------*/
void Tick handler ( void *ptr )
{
      #define L2 INTERVAL TIME 1 /* seconds */

      S32 device error;                   /* contain I2C device error number */
      mscc POE STATUS e mscc poe status e;/* contain microsemi poe status number*/

    while(POE TRUE)
    {
      usleep(L2_INTERVAL_TIME*1000);   /* sleep for L2_INTERVAL_TIME (1 second) */

      /* call for API function MSCC POE Timer Tick every 1 second */
      mscc_poe_status_e = MSCC_POE_Timer_Tick(_IN L2_INTERVAL_TIME,_OUT
&device error);
      if(mscc poe status e != e POE STATUS OK)
      {
  //Error occur
      }
    }
}
```

8 Appendix A: Partially/Modified Supported Commands

Microsemi only provides partial support to some of the 15 bytes PoE protocol commands, as described below

8.1 Added Functionality to Existing Commands

Set Enable/Disable Channels - AF Mask field:

- **0**: only IEEE802.3AF operation.
- **N**: Stay with the last mode (IEEE802.3af or IEEE802.3at).

  New functionality: IEEE802.3at operation is enabled for the specific port.

8.2 Partially Supported Commands

- **Set / Get Individual Mask**: Supported fields are:
  - Alternative A/B
  - AC/DC disconnect
  - Ignore priority upon port startup
  - Layer2 (LLPD)
  - PD_Port_Priority_by_Layer2
- **Get PoE Device Status**: Not supported: Comm status (No enhanced mode MCU)
- **Get System Status**: Not supported fields are:
  - CPU Status 1
  - CPU Status 2
  - Factory Default
  - GIE
  - User Byte
  - Interrupt register field: bit 10 (PoE device fault bit)

8.3 Commands Not Supported

- Reset Command
- Restore Factory Defaults
- Save System Settings
- Save User Byte
- Save / Get Non-volatile Memory
- Set System OK LED Mask Registers
- Set / Get Extended PoE Device Params
- Get System OK LED Mask Registers

9 Appendix B: PoE API Driver Source Code Description

9.1 Project: mscc_poe_api

- **mscc_poe_api.c**: Contains top-level PoE API interface functions such as MSCC_POE_Init, MSCC_POE_Write, MSCC_POE_Read, MSCC_POE_Exit
- **mscc_poe_cmd.c**: Contains functions that operate PoE tasks.
- **mscc_poe_comm_protocol.c**: Contains functions that decode the 15 bytes protocol and operate the proper task.
- **mscc_poe_host_communication.c**: Contains host interface functions as: WriteMsgToTxBuffer and Read_From_Msg_Tx_Buffer.
- **mscc_poe_ic_communication.c**: Contains the I2C interface for the communication between the Asics and the POE software.
- **mscc_poe_ic_func.c**: Contains System oriented functions.
- **mscc_poe_util.c**: Contains diverse Utilities functions as: formulas conversions and checksum operations.
- **mscc_poe_api.h**: Contains prototypes for PoE API interface functions.
- **mscc_poe_cmd.h**: Contains prototypes.
- **mscc_poe_comm_protocol.h**: Contains prototypes.
- **mscc_poe_db.h**: Contains the PoE API software internal types and data base.
- **mscc_poe_global_types.h**: Contains the PoE API software global types and constants.
- **mscc_poe_host_communication.h**: Contains prototypes.
- **mscc_poe_ic_communication.h**: Contains prototypes.
- **mscc_poe_ic_func.h**: Contains prototypes.
- **mscc_poe_ic_param_def.h**: Contains the PoE ICs registers definitions.
- **mscc_poe_util.h**: Contains prototypes.

9.2 Project: Architecture

- **mscc_arch_functions.c**: Contains the implementation of specific operating system functions (as system delay).
- **mscc_arch_functions.h**: Contains the definitions of specific architecture (as Linux).

9.3 Project: Examples

- **poe_util.c**: Contains utilities for general and PoE protocol purpose.
- **example_poe_comm_protocol_engine.c**: Contains PoE API Example Code using external UART.
- **poe_util.h**: Contains prototypes.
- **example_poe_comm_protocol.c**: Contains PoE API Example Code using and verify proper setup of PoE hardware, I2C connectivity proper compilation.
- **mscc_hal_i2c_aardvark.h**: Contains prototypes for Aardvark I2C operations.
- **mscc_hal_i2c_aardvark.c**: Contains the Aardvark specific functions implementation for I2C operations.
- **aardvark.so**: Linux shared object, used to access Aardvark devices through the API.
- **aardvark.h**: API header file.
- **aardvark.c**: Interface module.

Revision History

Revision Level / Date	Para. Affected	Description
1.0 / June 15th 2009	-	Initial release
1.1 / July 28, 2009	Whole Document	Formatting, English, Logo

For support contact: sales_AMSG@microsemi.com

Visit our web site at: www.microsemi.com **Catalog Number: 06-0054-056**