Power over Ethernet

Microsemi PoE Application Program Interface (API)

User Guide

Revision 1.3
Catalog Number 06-0054-056

Table of Contents

1 Introduction 4
1.1 Basic Function Information 4
1.2 Threads 5
1.3 Device Discovery Protocol – LLDP 5
1.4 Layer 2 5
1.5 PoE IC's Default Configuration 5
1.6 PoE Software Initial Configuration Parameters 6
1.7 Software Distribution Structure 6
2 Code Data Types 7
3 Code Functions Return Value 8
4 User PoE API Software Interface Description 9
4.1 MSCC_POE_Init() 9
4.1.1 Details 9
4.1.2 Arguments 9
4.1.3 Return Value 10
4.1.4 Example 11
4.2 MSCC_POE_Write() 11
4.2.1 Details 11
4.2.2 Arguments 11
4.2.3 Return Value 11
4.2.4 Example 12
4.3 MSCC_POE_Read() 12
4.3.1 Details 12
4.3.2 Arguments 12
4.3.3 Return Value 12
4.3.4 Example 12
4.4 MSCC_POE_Exit () 12
4.4.1 Details 12
4.4.2 Arguments 13
4.4.3 Return Value 13
4.4.4 Example 13
4.5 MSCC_POE_Timer_Tick () 13
4.5.1 Details 13
4.5.2 Arguments 13
4.5.3 Return Value 13
4.5.4 Example 13
5 Software Examples 15
5.1 Example 1: Description 15
5.2 Example 2: Description 16
6 Integrating PoE API Software with Host Software 18
6.1 Basic Procedure 18
6.2 Implementing I^2C Read/Write Function Calls 23
6.2.1 Pointer for Writing Driver Function 23
6.2.2 Pointer for Reading Driver Function 24
6.3 Obtaining LLDP and Layer 2 Power Management Functionality 25
6.4 Simplifying the API Implementation 25
6.5 KeepAlive - Verification 25
6.6 PoE API Functions Usage 26
7 Appendix A: Partially/Modified Supported Commands 29
7.1 Added Functionality to Existing Commands 29
7.2 Partially Supported Commands 29
7.3 Commands Which Are Not Supported 29

2381 Morse Avenue, Irvine, CA 92614, USA; Within the USA: (800) 713-4113, Outside the USA: (949) 221-7100 Fax: (949) 756-0308

8 Appendix B: PoE API Driver Source Code Description 30
8.1 Project: mscc_poe_api 30
8.2 Project: Architecture 30
8.3 Project: Examples 30
9 Appendix C: Port Disconnection on Power Budget Exhaustion 31
9.1 Port Disconnection on Power Budget Exhaustion 31
9.2 Port Disconnection on Power Budget Change 31
9.3 Further Reading 31
10 Appendix D: Software Default Parameters 32

2381 Morse Avenue, Irvine, CA 92614, USA; Within the USA: (800) 713-4113, Outside the USA: (949) 221-7100 Fax: (949) 756-0308

1 Introduction

The Microsemi's® PoE Application Program Interface (API) has been developed to enable customers using Enhanced Mode 15 byte PoE communication protocol, with PoE IC's PD69000, PD63000, PD62000, PD33000 or PDIC66000 MCUs to easily migrate to PD69012 or PD69008 Auto Mode solutions.

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 Revision 6.4).

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

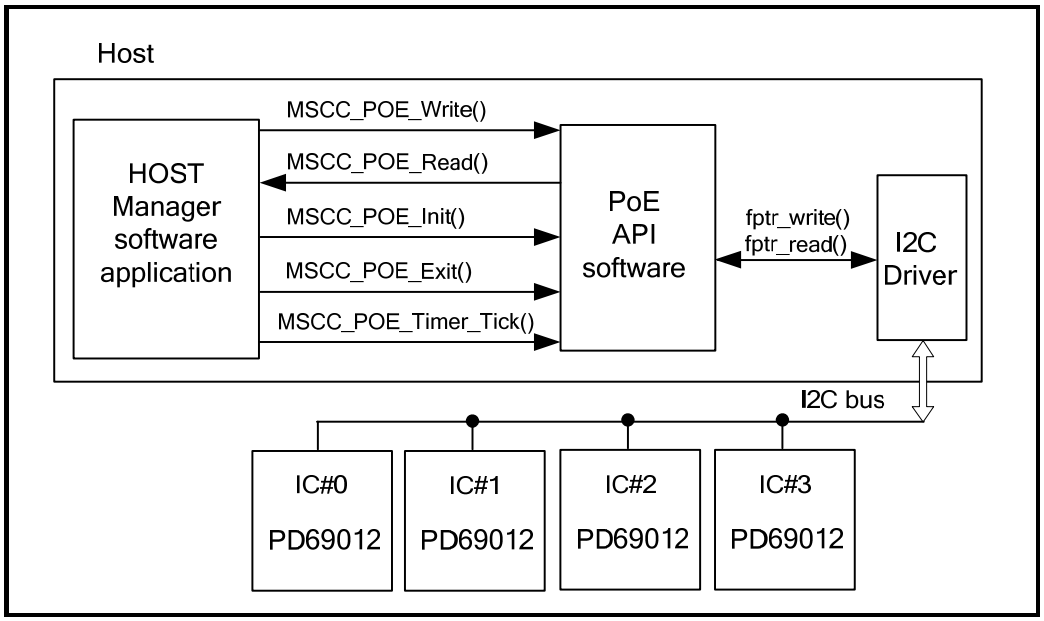

Figure 1: General System Schematic

1.1 Basic Function Information

Refer to User PoE API Software Interface Description, page 9 for complete information on these functions.

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

- Build 15 byte command messages (**Commands/requests**).
- 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 (**Reports**/ **Telemetry**) as per the Serial Communication Protocol User Guide.

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

Call function ***MSCC_POE_Exit()*** whenever the user software needs to terminate communications with Microsemi PoE API.

MSCC_POE_Timer_Tick() implements the:

- LLDP Power Management functionality (part of 802.3at spec).
- Power management - disconnect process. When activated, MSCC_POE_Timer_Tick() sends the 15 byte command "Set System Masks" whenever a Maskz field bit0 is cleared.

1.2 Threads

The PoE API is "thread safe", but doesn't support multi threading. PoE API has only a single instance, and therefore should not be initiated from several threaded functions. Every *MSCC_POE_Write()* function call must be followed by an *MSCC_POE_Read()* function call (otherwise the second sequenced *MSCC_POE_Write()* result overwrites the first *MSCC_POE_Write()* result).

The PoE API requires function *MSCC_POE_Timer_Tick()* to be called every second. Since timer actions are asynchronous, the PoE API must protect against a situation in which in the middle of *MSCC_POE_Write()* activity, timer tick occurs in an asynchronous manner and calls *MSCC_POE_Timer_Tick()* which accesses internally to PoE ICs. The use of Mutex in all PoE API functions (including MSCC_POE_Timer_Tick()) resolves this issue.

However when:

- all PoE actions are called from a single thread and
- the **MSCC_POE_Timer_Tick()** is not called in an asynchronous manner (for example from the main loop)

then there is no need to implement the Mutex functions inside the API.

The PoE API driver code does not have any thread/task.

1.3 Device Discovery Protocol – LLDP

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).

1.4 Layer 2

- LLDP and Layer 2 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.
- A configuration mask to enable / disable Layer 2 operation is enabled by default.
- A configuration mask to enable / disable priority definition by PD is disabled by default.

1.5 PoE IC's Default Configuration

As part of function **MSCC_POE_Init()**, the function will initialize PD690xx ICs as per the binary configuration file *pd690xx_configuration/cfg_mx.bin*.

1.6 PoE Software Initial Configuration Parameters

User may modify initial configuration and operation by modifying file: mscc_poe_default_parameters.h for example PoE Port Max Power in 802.3af/at modes. For additional information, refer to Appendix D: Software Default Parameters, page 32.

1.7 Software Distribution Structure

The software package is made up of three sections:

- **Examples**: Software distribution contains two Linux based examples. Each example is linked with the *mscc_poe_api_lib.a* and *mscc_arcitecture_lib.a* libraries. The examples were created and tested on Linux Fedora Core 9 distribution.
- **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.
- **PoE API** A (mscc_poe_api): A 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.
- **pd690xx_configuration**: This folder contains default binary configuration files for each IC version. The host will be asked to load the appropriate file in order to write configuration data to ICs on init process.

Figure 2: PoE API Tree Structure

2 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  ;
```

3 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 below).

```
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_CFG_HEADER_ID_VALUE               = -13,
    e_POE_STATUS_ERR_CFG_HEADER_CHECKSUM_VALUE         = -14,
    e_POE_STATUS_ERR_CFG_DATA_CHECKSUM_VALUE           = -15,
    e_POE_STATUS_ERR_ENTER_SW_CONFIG_MODE              = -16,
    e_POE_STATUS_ERR_EXIT_SW_CONFIG_MODE               = -17,
    e_POE_STATUS_ERR_FILE_OPENING_FAIL                 = -18,
    e_POE_STATUS_ERR_UNKNOWN_IC_VERSION                = -19,
    e_POE_STATUS_ERR_MAX_ENUM_VALUE                    = -20,
}mscc_POE_STATUS_e;
```

4 User PoE API Software Interface Description

The following five software functions control the PoE API:

- MSCC_POE_Init(), page 9
- MSCC_POE_Write(), page 11
- MSCC_POE_Read(), page 12
- MSCC_POE_Exit (), page 12
- MSCC_POE_Timer_Tick (), page 13

4.1 MSCC_POE_Init()

```
MSCC POE STATUS e MSCC POE Init( IN mscc InitInfo t *pInitInfo, OUT S32
*pDevice_error)
```

4.1.1 Details

This function initializes *MSCC PoE API*. The *mscc_InitInfo_t* structure pointer defines PoE hardware type, I^2C driver read and write functions. Up to eight PoE ICs are supported

4.1.2 Arguments

- mscc_InitInfo_t *pInitInfo:
- mscc_InitInfo_t structure contains the followed members:
  - U8 IC_Address[MAX_ASIC_ON_BOARD] : I^2C address for each PoE IC. No existing IC should be marked by IC_MAX_ADDRESS (0xFF).

  Note:

  Fill the I^2C address from IC_Address[0], up to the number of ICs and set the rest to IC_MAX_ADDRESS (0xFF).
  - U8 NumOfExpectedChannesInIC[MAX_ASIC_ON_BOARD]: Number of PoE ports for each IC. Use one of the following options:
    - o ASIC_8_CHANNELS: 8 PoE channels
    - o ASIC_12_CHANNELS: 12 PoE channels
    - o ASIC_NONE_CHANNELS: None

    Note:

    Fill IC's number of ports to ASIC_8_CHANNELS or _12_CHANNELS and set the rest to ASIC_NONE_CHANNELS.
  - U8 NumOfActiveICsInSystem: Number Of PoE ICs In the System
  - mscc_FPTR_Write fptr_write: I^2C driver write function pointer (this function should be implemented by the user). The I^2C 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 I^2C_Address: IC's I^2C 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 the I^2C driver for each I^2C 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 the I²C driver for each I²C read access

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

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

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

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

4.1.3 Return Value

MSCC_POE_STATUS_e

4.1.4 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
/=====================================================================*/
```

4.2 MSCC_POE_Write()

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

4.2.1 Details

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

4.2.2 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 I^2C driver returned error code during write operation.

4.2.3 Return Value

MSCC_POE_STATUS_e

4.2.4 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, 0x03, 0x04 };

mscc poe status = MSCC POE Write(data out, 15,&device error);
if(mscc poe status != POE STATUS OK)
{
//Error occurred
}
```

4.3 MSCC_POE_Read()

```
MSCC POE STATUS e MSCC POE Read ( OUT U8* pRxdata, IN U16 num read length,  OUT
S32 *pDevice_error);
```

4.3.1 Details

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

4.3.2 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 I^2C read or I^2C write operations.

4.3.3 Return Value

MSCC_POE_STATUS_e

4.3.4 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
}
```

4.4 MSCC_POE_Exit ()

```
MSCC POE STATUS e MSCC POE Exit( IN mscc CloseInfo t *pMscc CloseInfo, OUT S32
*pDevice_error)
```

4.4.1 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).

4.4.2 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 I^2C read or I^2C write operations.

4.4.3 Return Value

MSCC_POE_STATUS_e

4.4.4 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
}
```

4.5 MSCC_POE_Timer_Tick ()

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

4.5.1 Details

MSCC_POE_Timer_Tick() is used for the following issues:

- Proper implementation of LLDP Power Management functionality (part of 802.3at spec)
- Proper implementation of the "Set System Masks"15 byte command whenever the Maskz field bit0 (power-disconnect process) is cleared

If the **MSCC_POE_Timer_Tick()** is not called, an error report is not generated. However, the above two issues are not implemented.

4.5.2 Arguments

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

4.5.3 Return Value

mscc_POE_STATUS_e

4.5.4 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
      }
    }
}
```

5 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 I^2C interface. To test PoE API examples, obtain the Total Phase USB to I^2C interface named Aardvark from the following URL:

  http://www.totalphase.com/products/aardvark_i2cspi/
- A Microsemi PoE evaluation board, 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 I^2C address 0x30 and 0x31.

5.1 Example 1: Description

Example 1 (Figure 3), 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.

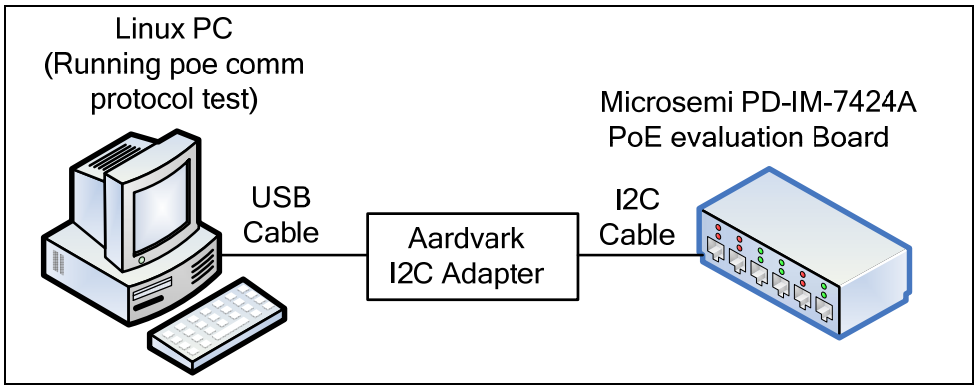

Figure 3: Example 1 Setup

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]#
```

5.2 Example 2: Description

This example (Figure 4), 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.

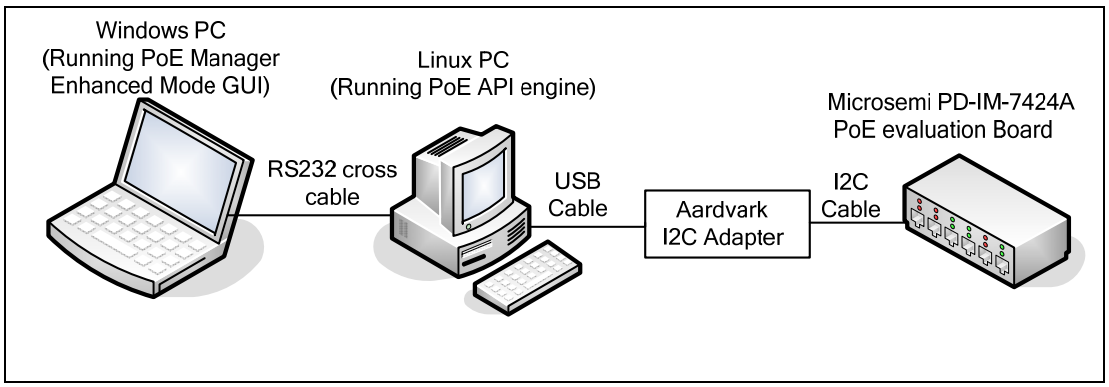

Figure 4: Example 2 Setup

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:

```
[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
```

6 Integrating PoE API Software with Host Software

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

6.1 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*.

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

4. 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_GetConfigurationFromFile(_IN U8 IC_ver, _OUT U8 **pbyConfiguration)**: The function loads the pd690xx configuration binary file (cfg_mx.bin) to memory and returns a byte pointer to the start of the memory block. The loaded cfg_mx.bin file is dependent on the IC software version (x = IC_ver).
   - **OS_mutex_init ()**: Initializes the mutex.
   - **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.
- In cases where the architecture has only one thread (there is no need to modify the OS_mutex functions) you can use the "**_NEW_ARCH_**" definition. In this case there is no needed to implement the mutex functions (OS_mutex_init (), OS_mutex_lock (), OS_mutex_unlock ()).
- OS mutex API needs to be implemented if requires the PoE API driver code to be "multi-thread safe".

  The OS mutex are not needed to be implemented only in case that only one thread is in use, and function MSCC_POE_Timer_Tick () is being called from same thread. Please note that if timers act as an asynchronous self dependent software action, then Mutex must be implemented.

5. Open the following file:

   mscc_arcitecture/inc/mscc_arch_functions.h.

6. Modify the text marked in yellow as needed.

```
//** 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: the function load the pd690xx configuration binary file
                  (cfg mx.bin) to memory and return a byte pointer to the
                  start of the memory block. the loaded cfg mx.bin file is
                  depending on the IC software version (x = IC ver).
* input :    IN U8 IC ver    -  IC version
 * output:   pbyConfiguration - byte pointer to binary block data in memory
 *                              which loading from configuration binary file
 *
 * return:   e_POE_STATUS_OK                       - operation succeed
 *           e POE STATUS ERR UNKNOWN IC VERSION   - operation failed due to
                                                     unknown IC version
 *           e_POE_STATUS_ERR_FILE_OPENING_FAIL    - operation failed due to file
                                                     opening operation failure
 *-------------------------------------------------------------------*/

S32 OS GetConfigurationFromFile( IN U8 IC ver,  OUT U8 **pbyConfiguration)
{
    U16 wNumBytes = 0;
    char *filename = NULL;

    /* build path to configuration bin file */
    /* the current directory is :example uart to 15 bytes protocol so in order to
       reach the cfg m2.bin file we have to navigate back to pd690xx configuration
       directory*/

    if ( IC ver == 2 )
        filename = "./../../../pd690xx configuration/cfg m2.bin";
    else if ( IC ver == 3 )
        filename = "./../../../pd690xx_configuration/cfg_m3.bin";
    else
        return e POE STATUS ERR UNKNOWN IC VERSION;

    /* open the configuration bin file */
    FILE *fpin = fopen(filename, "rb");
    if (fpin == NULL)
    {
        fprintf(stderr, "Cannot open file %s \n", filename);
        perror("Input file open failed\n");
        return e POE STATUS ERR FILE OPENING FAIL;/* means file opening fail */
    }
```

```
    /* printf("file name is %s\n", filename); */

    /* copy bytes from bin file to bytes array */
    while (!feof(fpin))
    {
        if (fread(&byConfigurationFileBuffer[wNumBytes],1,1, fpin) == 1)
            wNumBytes++;
    }

    /*printf("EOF reached after %d bytes read in!!!\n", wNumBytes);*/

    /* assign local bytes buffer array to function input bytes pointer */
    *pbyConfiguration = byConfigurationFileBuffer;

    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
*-----------------------------------------------------------------------*/
 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: the function load the pd690xx configuration binary file
                  (cfg mx.bin) to memory and return a byte pointer to the
                  start of the memory block. the loaded cfg mx.bin file is
                  depending on the IC software version (x = IC ver).
 *  input :    IN U8 IC ver    -  IC version *
 *  output:   pbyConfiguration -  byte pointer to binary block data in memory
                                  which loading from configuration binary file *
 *  return:   e POE STATUS OK                          - operation succeed
 *            e POE STATUS ERR UNKNOWN IC VERSION - operation failed due to
                                                     unknown IC version
 *            e POE STATUS ERR FILE OPENING FAIL  - operation failed due to file
                                                    opening operation failure
 *-------------------------------------------------------------------*/
S32 OS GetConfigurationFromFile( IN U8 IC ver,  OUT U8 **pbyConfiguration)
{
    U16 wNumBytes = 0;
    char *filename = NULL;

    /* build path to configuration bin file */
    /* the current directory is :example uart to 15 bytes protocol so in order to
       reach the cfg m2.bin file we have to navigate back to pd690xx configuration
       directory*/
```

```
      if ( IC ver == 2 )
            filename = "./../../../pd690xx configuration/cfg m2.bin";
      else if ( IC ver == 3 )
            filename = "./../../../pd690xx configuration/cfg m3.bin";
      else
            return e POE STATUS ERR UNKNOWN IC VERSION;

#error OS GetConfigurationFromFile function should be Implement.

   /* TODO - implement here the function depending your architecture :
    *     1. Open the configuration bin file
    *     2. Copy bytes from bin file to byConfigurationFileBuffer bytes array */

   /* assign local bytes buffer array (byConfigurationFileBuffer) to function
      input bytes pointer (pbyConfiguration) */
      *pbyConfiguration = byConfigurationFileBuffer;

      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_GetConfigurationFromFile(_IN U8 IC_ver, _OUT U8 **pbyConfiguration), OS_mutex_init (),OS_mutex_lock(),OS_mutex_unlock()*

depending your architecture.

6.2 Implementing I²C Read/Write Function Calls

PoE API software communicates with the PD690XX ICs through the I²C interface. Implement I²C read/write functions per the following functions prototype. Update **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.

6.2.1 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;
}
```

6.2.2 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 the 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;
}
```

6.3 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 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).

6.4 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*.

6.5 KeepAlive - Verification

Keep-Alive verification (meaning checking if hardware reset has occurred) is required and must be implemented in the host application layer.

To verify, regularly read the **Private Label** field (read the 15 Bytes Protocol command: "Get System Status").

1. KeepAlive mechanism: Set the **Private Label** to any value higher than 0x00 (for example KEEP_ALIVE_VALUE= 0xA5) by using the "Set System Status" communication protocol command.

2. Periodically (and regularly) read the **Private label** field by using the "Get System Status" communication protocol command.

If the read value is different then the set **KEEP_ALIVE_VALUE** value, then reset has occurred. The PD690XXs should reinitialize by calling **MSCC_POE_Init** PoE API function.

When reset occurs, the value of **Private label** is changed to 0x00.

Implementation example:

```
/* Keep-Alive verification */

U8 Private label;
U8 data reply buffer[15]; /* contain the 15 bytes communication protocol reply
message data  */
```

```
/* Get System status - 15Bytes communication protocol command*/
static U8 protocol data out GetSystemStatus[] = { B Request, 0x00, B Global,
B SystemStatus, B Space, B Space, B Space, B Space, B Space, B Space, B Space,
B Space, B Space, 0x3, 0x4 };

/* Reading of the Private label field by using the "Get System Status"
communication protocol command */

/* Send 15 byte protocol msg to PoE API */
mscc poe status e =
MSCC POE Write(protocol data out GetSystemStatus,15,&device error);
if(mscc poe status e != e POE STATUS OK)
{
    /* error occurred */
}

/* Read the 15 byte protocol msg reply from the PoE API */mscc poe status e =
MSCC POE Read(data reply buffer, 15, &device error) ;
if(mscc poe status e != e POE STATUS OK)
{
    /* error occurred */
}

/* check Private label data */
Private label = data reply buffer[6];
if(Private label != KEEP ALIVE VALUE)
{
      /***** Reset occurred *****/

      /* Init PoE API software - PD690XX's initializing is needed */
      mscc_poe_status_e = MSCC_POE_Init(&mscc_InitInfo,&device_error);
      if(mscc_poe_status_e != e_POE_STATUS_OK)
      {
      /* error occurred */
      }
}
```

6.6 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;       /* PoE API software Initialization
                                        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
      }
    }
}
```

7 Appendix A: Partially/Modified Supported Commands

This section describes limitations with the PoE API's capabilities to fully emulate Microsemi's traditional Enhanced solutions. The Enhanced mode reference document is the *User Guide - PD63000 & PD69000/G Serial Communication Protocol*, Catalog Number 06-0032-056 Revision 6.4.
The provided partial support to some of the 15 bytes PoE protocol commands described below.

7.1 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.

7.2 Partially Supported Commands

- **Set / Get Individual Mask**: Fields which are supported:
  - Alternative A/B
  - AC/DC disconnect
  - Ignore priority upon port startup
  - Layer2 (LLPD)
  - PD_Port_Priority_by_Layer2
- **Get PoE Device Status**: Does not support Comm status (No enhanced mode MCU)
- **Set System Masks**: MaskBit0 (Power Management): behavior changed. For more information see Appendix C: Port Disconnection on Power Budget Exhaustion.
- **Get System Status**: Fields which are not supported:
  - CPU Status 1
  - CPU Status 2
  - Factory Default
  - GIE
  - User Byte
  - Interrupt Register Field: bit 10 (PoE device fault bit)

7.3 Commands Which Are 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

8 Appendix B: PoE API Driver Source Code Description

8.1 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 I^2C 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.
- **mscc_poe_default_parameters.h**

8.2 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).

8.3 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, I^2C connectivity proper compilation.
- **mscc_hal_i2c_aardvark.h**: Contains prototypes for Aardvark I^2C operations.
- **mscc_hal_i2c_aardvark.c**: Contains the Aardvark specific functions implementation for I^2C operations.
- **aardvark.so**: Linux shared object, used to access Aardvark devices through the API.
- **aardvark.h**: API header file.
- **aardvark.c**: Interface module.

9 Appendix C: Port Disconnection on Power Budget Exhaustion

This section describes the PoE API's limitations to fully emulate Microsemi's traditional Enhanced Port Disconnection on Power Budget Exhaustion.

9.1 Port Disconnection on Power Budget Exhaustion

There are two modes for managing powering priorities, Enhanced and Auto.

In Enhanced Mode systems priorities are managed on a system basis, so that if all ports have the same priority (Low, High or Critical), the port with the lowest number receives the highest priority. When there are ports having different priorities, then those ports having a lower priority and a higher number will be disconnected when the power budget is exhausted (assuming that the lower number port has a PD present).

In Auto Mode the Critical, High and Low priorities are maintained, but there is no importance to the physical order of the ports during the disconnection.

9.2 Port Disconnection on Power Budget Change

In Enhanced Mode systems, the port matrix is used to map between logical and physical ports, so that the system layout can be designed with more freedom. When the power budget is lowered, the ports with the highest numbers for the lowest priority being powered (Low, High, or Critical) are turned off, so that the total power consumption matches the new power budget.

While the port matrix could be used in the PoE API to support mapping physical ports to logical ports for port monitoring purposes (such as LED indication), the Auto power management priorities are configured so that the Critical, High and Low priorities are maintained, but there is no importance to the physical/logical order of the ports during the disconnection.

9.3 Further Reading

- TN-113 – Power Management for PoE Units, Catalog Number 06-0002-081
- TN-144 – PD690xx Auto-Mode Power-Management, Catalog Number 06-0028-081

10 Appendix D: Software Default Parameters

Default PoE parameters are defined in file: *mscc_poe_default_parameters.h* . Calling function **MSCC_POE_Init()** initializes all PoE ICs as per the values in file *mscc_poe_default_parameters.h*. You can modify default parameters. The following h file code describes the software default parameters.

```
/* Enable/Disable Ports - All ports enabled
 * Values:
 *   0 : disabled
 *   1 : enabled  */
#define DEF ENABLE DISABLE PORTS                                    1

/* Port Power LimitPower per AF port (milliwatt units) */
#define DEF PORT POWER LIMIT PER AF mW                              15400

/* Port Power Limit Power per AT port (milliwatt units) */
#define DEF PORT POWER LIMIT PER AT mW                              36000

/* Port Priority - Lowest priority set for all ports
 * Values:
 *   e PortPriority Critical
 *   e PortPriority High
 *   e PortPriority Low                                 */
#define DEF PORT PRIORITY                                           e PortPriority Low

/* Port Standard    IEEE802.3af or IEEE802.3at
 * Values:
 *   e AF Mode
 *   e AT Mode                                  */
#define DEF PORT STANDARD                                           e AT Mode

/* Masks Status     Mask bit-1
 * Values:
 *   0 : disabled
 *   1 : enabled         */
#define DEF CAP DETECTION ENABLED                                   1

/* Private Label - Assist in detecting reset events
 * Values: 0 to 255  */
#define DEF PRIVATE LABLE                                           0

/* PM Mode-Power Management Mode parameters - default is Cat. Ref - D.
```

	PM Mode			Management Mode	Cat Ref	Total Allocated Power	Port Power Limit	Start Condition
	PM-1	PM-2	PM-3					
Default ->	0x00	0x02	0x00	Dynamic	D	Consumption	Max.	None
	0x00	0x00	0x00	Static	S1	Consumption	Predefined	None
	0x00	0x00	0x01		S2	Consumption	Predefined	Class
	> 0	0x01	0x00	Class	C1	Class	class	None
	> 0	0x02	0x00		C2	Class	Max.	None

```
#define DEF PM METHODE                                              0
#define DEF PORT ICUT LIMIT DEFINITION                              2
#define DEF STARTUP CONDITIONS                                      0

/* Power Bank 0-7 :Maximum power for each bank */
#define DEF POWER BANK                                              1612

/* Max Volt  If exceeded, the PoE ports shutdown */
#define DEF_MAX_VOLT                                                58.5

/* Min Volt  Below this value, the PoE ports shutdown */
#define DEF_MIN_VOLT                                                44

/* Interrupt Mask register - default: All unmasked
```

```
 * Values:
 *   0       - all interrupts enabled (unmasked)
 *   0xFFFF  - - all interrupts disabled (masked)          */
#define DEF_INTERRUPT_MASK_REGISTER                                  0

/* Temperature Alarm limit, used for interrupt */
#define DEF TEMPERATURE ALARM LIMIT                                  120

/**** default individual masks settings ****/

/* Backoff Time - Selects one of two valid four wire connections
 * Values:
 *   0   - Alternative A
 *   1   - Alternative B                                    */
#define DEF MASK BACKOFF TIME                                        0

/* Disconnect Method - AC Disconnect method for PD63000
 * Values:
 *   0   - DC enabled
 *   1   - AC enabled                                  */
#define DEF_AC_DISCONNECT_METHOD                                     0

/* Values:
 *   0 : disabled
 *   1 : enabled  */
#define DEF MASK LAYER2                                              1

/* Values:
 *   0 : disabled
 *   1 : enabled  */
#define DEF MASK LAYER2PRIORITY BY PD                                1

/* Values:
 *   0 : disabled
 *   1 : enabled  */
#define DEF MASK IGNORE PRIORITY UPON STARTUP                        1

/* Values:
 *   0 : enabled - allow higher-priority ports to power-up
 *   1 : disabled - power up is denied */
#define DEF POWER DISCONNECT PROCESS                                 1   /* system mask bit0
- 1:power up is denied */
```

Revision History

Revision Level / Date	Para. Affected	Description
1.0 / June 15th 2009	-	Initial release
1.1 / July 28, 2009	Whole Document	Adding supports of Layer2 and LLDP
1.2 / Nov 23, 2009	Whole Document	Adding configuration code file download feature Accelerate the Get All Telemetry commands Adding High Priority Connect Implementation
1.3 / Dec 2009	4.5.1	Correct details description

For support contact: sales_AMSG@microsemi.com

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