2014

Author: Robert Huang

Version: 2.6

[Portwell Engineering Toolkit]

Version History

Version	Date	Remark	Author
0.1	**2011/01/28**	Release the first version of the PET user manual.	Robert
0.2	**2011/05/30**	Modify WDT API prototype.	Robert
0.3	**2011/08/02**	Building SMbus protocol function call.	Robert
1.0	**2011/08/19**	Release 1.0 version. Fix SMbus function call and content	Robert
1.1	**2011/08/29**	Add error code.	Robert
1.2	**2011/09/26**	Add GPIO Pin explanation.	Robert
1.3	**2012/04/23**	Add Backlight control API prototype.	Robert
2.0	**2012/05/07**	Add SMBus protocol	Angela
2.1	**2012/06/05**	Add beep and function and modify set fan ration value.	Robert
2.2	**2012/11/02**	Add Portwell Engineering Tool Utility	Angela
2.3	**2013/03/13**	Add I^2C API prototype and modify SMBus prototype	Angela
2.4	**2013/3/19**	Add EC SMBus API prototype and EC I^2C API prototype	Angela
2.5	**2013/05/02**	Modify WDT API content	Angela
2.6	**2014/05/07**	Modify data format, content and GPIO explain. Delete PET IIC API Prototype (Redefine EC I^2C)	Robert

Index

Overview

✧ Introduction

PET is Portwell Engineering Toolkit, its main function is to help users coding applications when need to control hardware. The Portwell has done all the hard work for customers with the release of a suit of APIs (Application Programming Interface), we called the PET API library.

An Application Programming Interface (API) is a set of routines, protocols, and tools for building software applications. A good API makes it easier to develop programs by providing all the building blocks, the programmer then puts the blocks together.

PET provides a set of user-friendly API, which speeds development and offers add-on value for Portwell platforms board. PET plays the role of a catalyst between the developer and solution, which makes embedded platforms easier and simpler to integrate with the customer's own applications. Operating Systems that PET support Windows XP, windows 7 and Linux operating system.

✧ Benefit

Faster Time to Market

PET's unified API helps developers write applications to control the hardware without knowing the hardware specs of the chipsets and driver architecture.

Reduces Project development Effort

When customers want to connect their own devices to the onboard bus, they can either write the driver and API from scratch or they can use PET to start the integration saving a huge amount of effort. Developers can reference the sample programs to see and learn more about the software development environment.

Enhances Hardware Platform Reliability

PET provides a set of trusted APIs which combines chipset and library support; controlling application development through PET enhances reliability.

Flexible Upgrade Possibilities

PET supports an easy upgrade solution for customers. Customers just need to install the new version PET that supports the new functions.

Test and verify Board

PET provides customers to test and verify onboard function fast.

Environment

The PET API library is used in any number of the Portwell boards that have Portwell BIOS support. We support operating system environment include:

- Windows XP
- Windows 7 (32bit)
- Windows 7 (64 bit)
- Linux (Such as Ubuntu or Fedora…etc.)

If you want more detailed information that the board and the support of the API; please refer to the other document for each board, for example: I want to get Portwell board has to provide those API, I can read User_Api.h/PET_Type.h head file and readme files to get more detail.

Package Contents

The Portwell Engineering toolkit currently support four operating systems – contain Windows XP, Windows 7 (32-bit), Windows7 (64-bit) and Linux. Content list below:

- Windows version: That's a software package, it contain following below:

  1. Portwell.sys and Portwellx64.sys
  2. PET.dll and PETx64.dll
  3. PET.lib and PETx64.lib
  4. Header file (User_Api.h and PET_Type.h)
  5. Sample Code (Sample.cpp)

- Linux have two type of the library, so it contain following below:

  1. libapi.so and libapix64.so
  2. libapi_static.a and libapi_staticx64.a
  3. Header file (User_Api.h and PET_Type.h)
  4. Sample Code (Sample.c)
  5. Makefile

Install

✧ Windows XP & Windows 7

The Windows driver can use PET_API_Init to auto install on the Windows XP or 7. You can use PET_API_Uninit to release the driver. Because the API function can reduce work which is to help users achieve the effect of rapid development.

✧ Linux System

The Linux library has two formats: "Static" and "Dynamic". The static library can directly include header file and make by use "Makefile". The Makefile will automatically compile and link the library to produce executable. The Dynamic library that must be installed in the Linux user library folder, and key command "make install" to do it.

Anticipatory Knowledge

Portwell Inc., a world-leading innovator in the Industrial PC (IPC) market and development by using Intel technology. The mother board contains many features, such as CPU, FSB, BIOS system Chipset, system memory, watchdog timer, hardware status monitor, and GPIO, etc.

This document is tell user how to use PET library to operate something hardware, so more detailed section may need to refer to other document, each piece has its own circuit board. Below is the use of PET library will have the basic knowledge, and if users are already familiar with can skip to the next section.

✧ Watch dog timer

A watchdog timer is a computer hardware or software timer that triggers a system reset or other corrective action if the main program, due to some fault condition, such as a hang, neglects to regularly service the watchdog. The intention is to bring the system back from the unresponsive state into normal operation.

Watchdog timers can be more complex, attempting to save debug information onto a persistent medium. In this case, simpler, watchdog timer ensures that if the first watchdog timer does not report completion of its information saving task within a certain amount of time, the system will reset with or without the information saved. The most common use of watchdog timers is in embedded systems, where this specialized timer is often a built-in unit of a microcontroller.

Watchdog timers may also trigger fail-safe control systems to move into a safety state, such as turning off motors, high-voltage electrical outputs, and other potentially dangerous subsystems until the fault is cleared.

The watchdog timer is a chip external to the processor. However, it could also be included within the same chip as the CPU; this is done in many microcontrollers. In either case, the watchdog timer is tied directly to the processor's reset signal. Expansion card based watchdog timers exist and can be fitted to computers without an onboard watchdog.

✧ GPIO

The GPIO (General Purpose Input Output) peripheral provides dedicated general-purpose pins that can be configured as either inputs or outputs. When GPIO is configured as an output, users can write to an internal register to control the state driven on the output pin. When GPIO is configured as an input, users can detect the state of the input by reading the state of an internal register. The GPIO includes the following features, such as general purpose input/output logic supports and GPIO interrupts support.

For example, the WADE-8070 board provides 8 input/output ports from SIO that can be individually configured to perform a simple basic I/O function. Users can configure each individual port to become an input or output port by programming register bit of I/O selection. To invert port value, the setting of Inversion Register has to be made. Port values can be set to read or write through Data Register.

✧ SMBus

The System Management Bus (SMBus) is a two-wire interface through which simple power-related chips can communicate with rest of the system. It is based on the principles of operations of I2C and it is used in personal computers and servers for low-speed system management communications. With the SMBus, a device can provide manufacturer information, tell the system what its model or part number is, save its state for a suspend event, report different types of errors, accept control parameters and return its status.

✧ I^2C

The I2C (Inter-IC) bus is a bi-directional two-wire serial bus that provides a communication link between integrated circuits (ICs). Phillips introduced the I2C bus 20 years ago for mass-produced items such as televisions, VCRs, and audio equipment. Today, I2C is the de-facto solution for embedded applications. The I2C bus supports 7-bit and 10-bit address space devices and devices that operate under different voltages.

The I2C bus and the SMBus are popular 2-wire buses that are essentially compatible with each other. Normally devices, both masters and slaves, are freely interchangeable between both buses. Both buses feature addressable slaves (although specific address allocations can vary between the two buses). The buses operate at the same speed, up to 100kHz, but the I2C bus has 400kHz versions. Obviously, complete compatibility between both buses using all devices is ensured only below 100kHz.

Here are some of the features of the I2C-bus:

- Only two bus lines are required; a serial data line (SDA) and a serial clock line (SCL).
- Each device connected to the bus is software addressable by a unique address and simple master/slave relationships exist at all times; masters can operate as master-transmitters or as master-receivers.
- It is a true multi-master bus including collision detection and arbitration to prevent data corruption if two or more masters simultaneously initiate data transfer.
- Serial, 8-bit oriented, bidirectional data transfers can be made at up to 100 kbit/s in the Standard-mode, up to 400 kbit/s in the Fast-mode, up to 1 Mbit/s in Fast-mode Plus, or up to 3.4 Mbit/s in the High-speed mode.
- Serial, 8-bit oriented, unidirectional data transfers up to 5 Mbit/s in

Ultra Fast-mode

- On-chip filtering rejects spikes on the bus data line to preserve data integrity.
- The number of ICs that can be connected to the same bus is limited only by a maximum bus capacitance. More capacitance may be allowed under some conditions.

Portwell Engineering Tool Utility

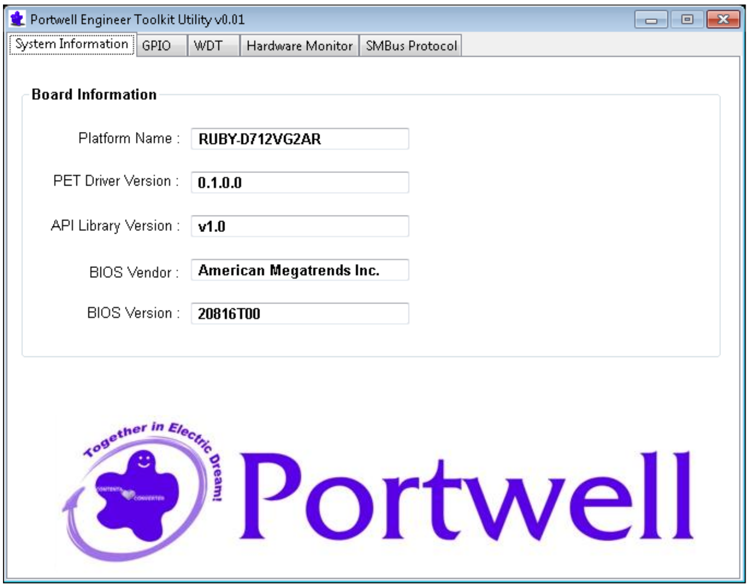
The Portwell Engineering Tool utility provides a cross-platform Graphical User Interface, which is based on the PET API function. This utility has good compatibility with Windows and Linux system. And it's an independent program that user can execute it without installing.

✧ Environment

The PET utility is used in any number of the Portwell boards that have Portwell BIOS support. We support operating system environment include:

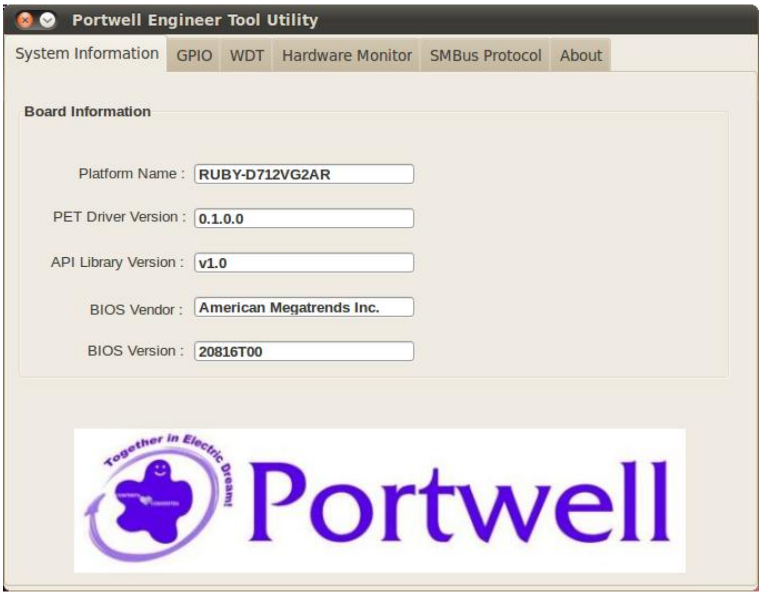
- **Windows XP**

➢ Windows 7 (32/64 bit)

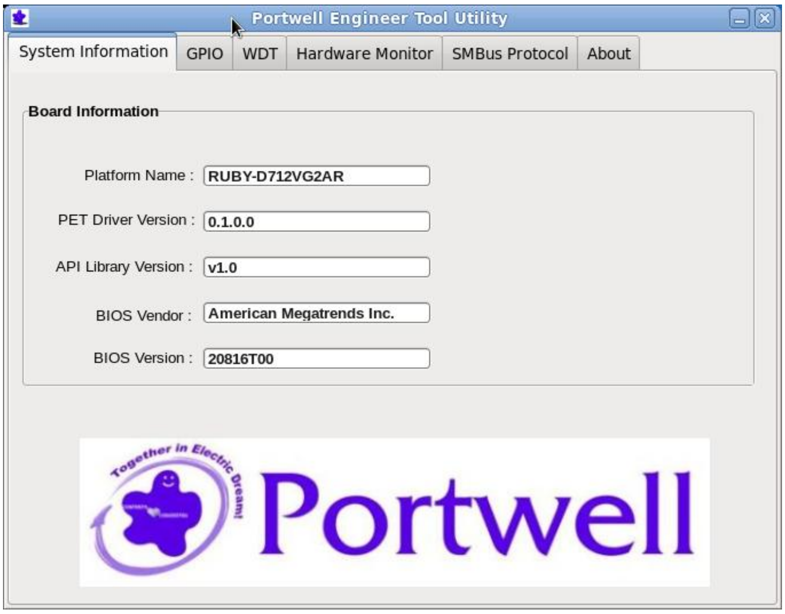


.

➢ Ubuntu10.04



➢ **Fedora 13**



✧ Operation Guide

The PET utility can only be operated in Portwell's product. If it's not, it will show this error message.

(Step 1)

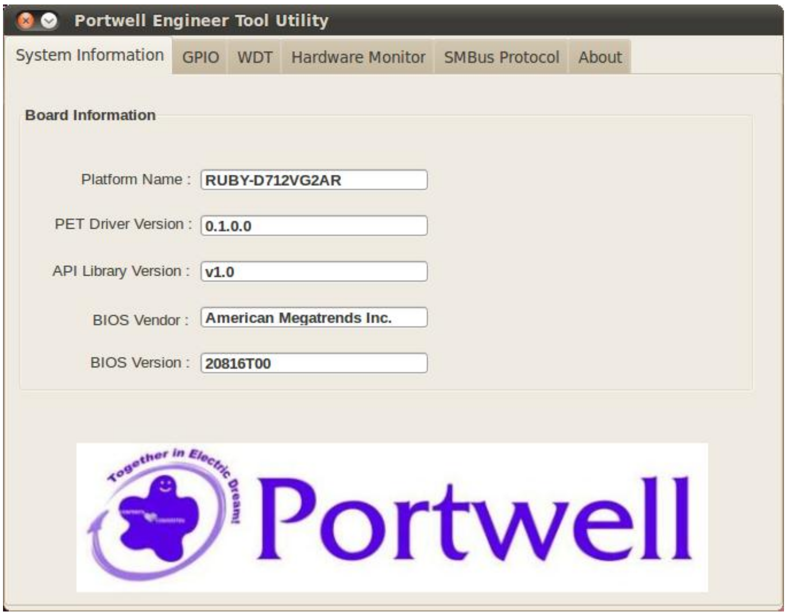
the Utility PET：On Windows, you must r**un as Administrator** to click the PET_Utility.exe. On Linux, you should **run as root mode** to execute it.

(Step 2)

If the PET program started, the tool page will show up as below.

System Information

This page displays product basic information.

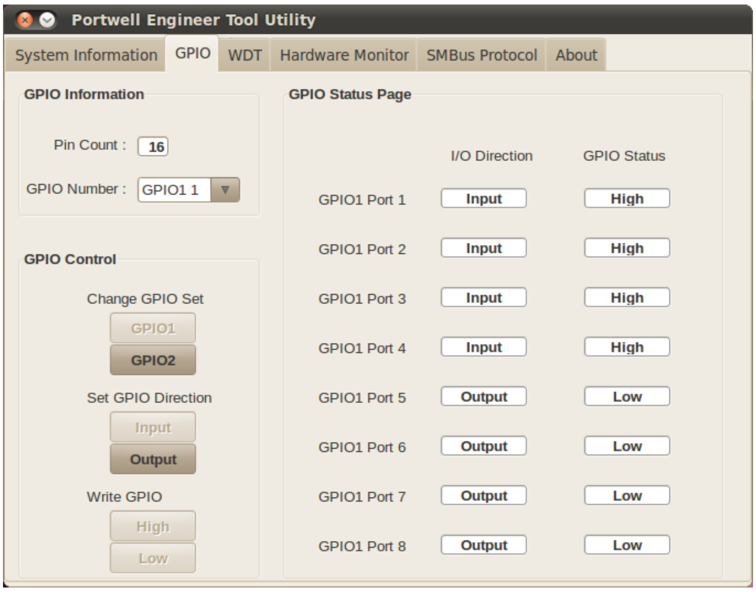


- Board information

1. Platform Name：This column shows up model only when this model is on PET supported list.
2. PET Driver Version
3. API Library version
4. BIOS Vendor
5. BIOS Version

GPIO

This page display GPIO information, GPIO control and GPIO status page. If you cannot see this page, it means PET cannot support this product in GPIO.



- GPIO Information

  1. Pin Count: show the count of available pin.
  2. GPIO Number: User can choose the GPIO pin number to control direction and High/Low.

- GPIO control

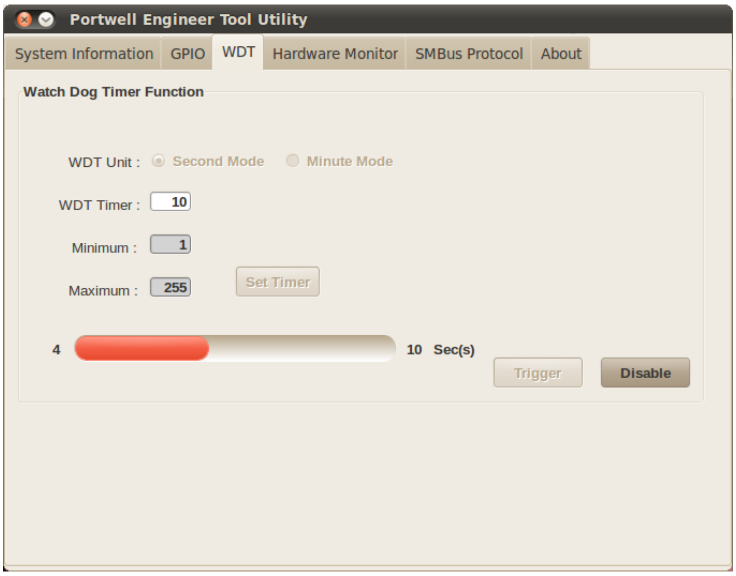
  1. Change GPIO Set: User can choose GPIO1 or GPIO2 set. If you can't click the button, it means this product only support one GPIO set.
  2. Set GPIO Direction: User can configure each individual GPIO pin to become an input or output port.
  3. Write GPIO: User can configure each individual GPIO pin to High or Low.

■ GPIO Status Page

The GPIO status will update immediately and be shown in this page.

WDT

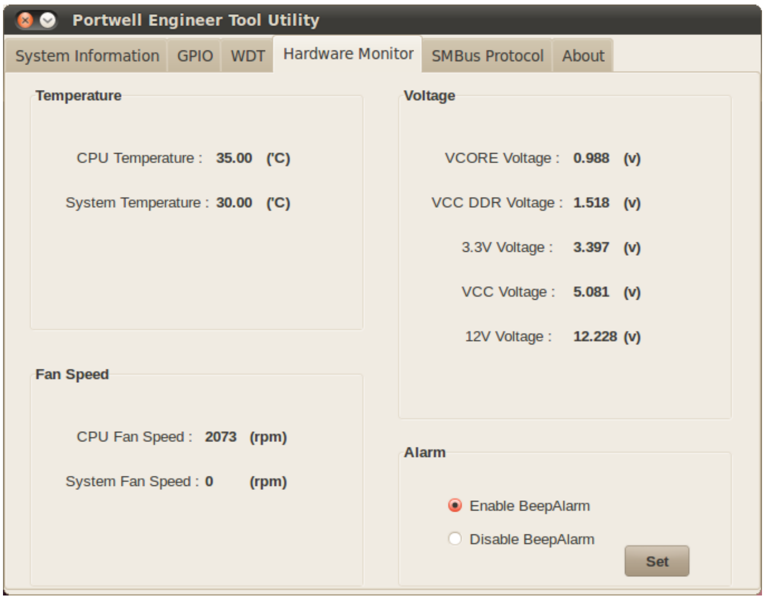
PET only supports WDT setup function to certain products. If user cannot find this WDT page, it means this product is not on the supported list.



The PET utility will show Max & Min timer range in the WDT page. User can choose the unit of time and input the timer, then click [Set Timer] to setup. Then click the [Trigger] to enable the timer and the BAR will show the progress. User can click [Disable] to disable WDT.

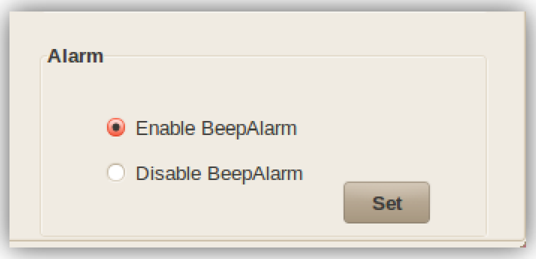
Hardware Monitor

PET only supports HWM function to certain products. If user cannot find this HWM page, it means this product is not on the supported list.



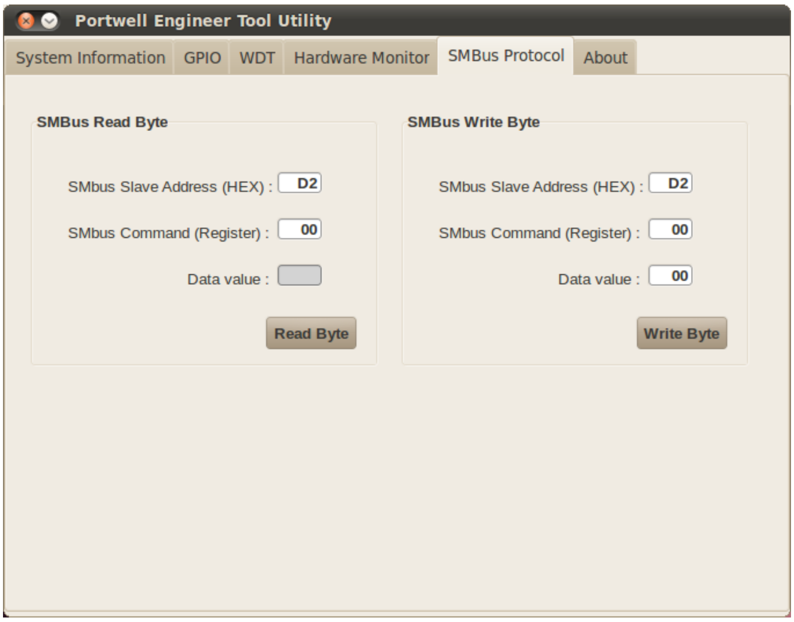
- Temperature
  1. CPU Temperature
  2. System Temperature
- Fan Speed
  1. CPU Fan Speed
  2. System Fan Speed
- Voltage
  1. VCORE Voltage
  2. VCC DDR Voltage
  3. 3.3V Voltage
  4. VCC Voltage
  5. 12V Voltage
- Alarm

User can use the Alarm Setting in Hardware Monitor page. Choose the [Enable Beep Alarm] and click [Set] to enable the Beep Alarm. And then user can choose the [Disable BeepAlarm] and click [Set] to disable the Beep Alarm.



SMBus Protocol

This page displays SMBus protocol information.



- SMBus Read Byte

  User can input SMbus Slave Address(HEX) and SMbus Command(Register), then click [Read Byte] to read the data value .

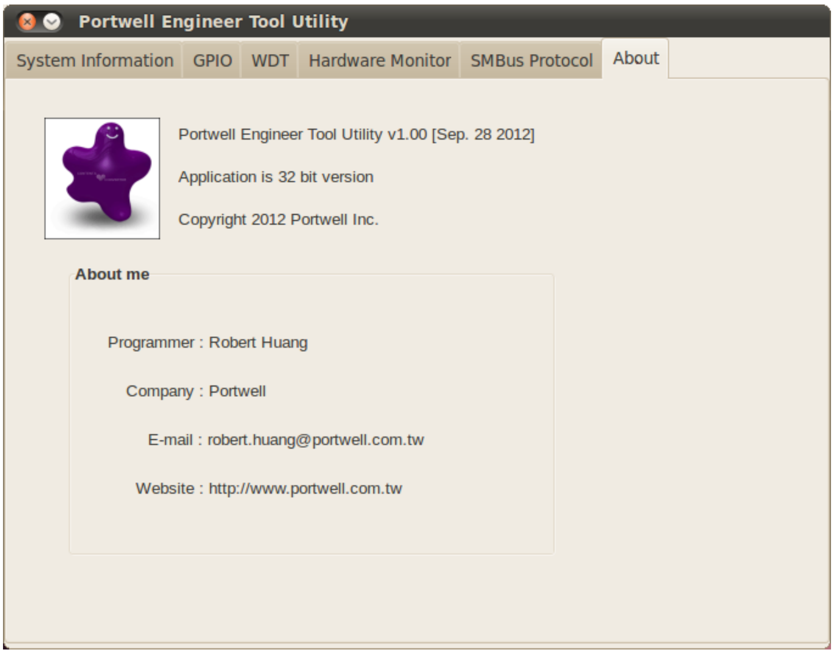
- SMBus Write Byte

  User can input SMbus Slave Address(HEX) and SMbus

Command(Register), then input the data value and click [Write Byte] to write the data value.

About

This page displays the information about PET utility and how to contact us.



PET Library Documentation

The Portwell Engineering toolkit (PET) provides many application function code to service user to control and measure our hardware. Because of every Portwell board need to write low level programming, so the engineers have to study board data sheet and reference other data to coding API function.

In order to eliminate user development time, and study the board document. We developed a PET library to provide user to control Portwell board to get functionality required. In addition, the standard PET application library can use in different operating system. The user doesn't modify any way to user PET library.

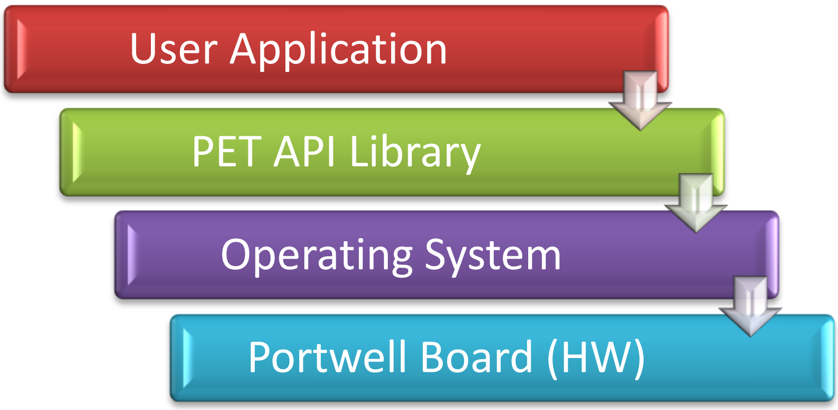


Figure 2. Basic architecture

This approach is in the hardware layer and an operating system layer was added our API layer (Figure 2). Let users don't understand the operating system and hardware operating conditions, and without changing the API prototypes and parameters. If the code doesn't depend on any operating system of the library, using only standard C/C++ library, that you can direct compiler you code on the your system.

We use the standard functions PET interface, different operating system users to see the standard interface functions and the same parameters. The details of which are based on different operating systems to be implemented, there would be the library for different systems (Figure 3).

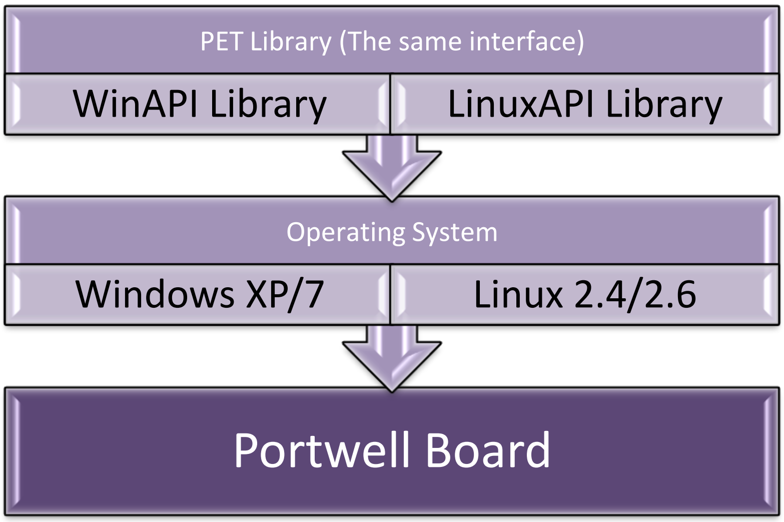


Figure 3. Different OS has different library

Additionally in a different Portwell board, we are all the standard PET application interface to write programs that allow users do not modify their own way once again code in next time upgrade our board, this way the user can further reduce development time.

When user calls the function, we use call by address to store the value, use the return value to inform the user of the calling process is correct, the user can determine whether the return value of the success or failure. The other more serious errors will be directly printed wrong reasons to force the end of the program.

All APIs return the UCHAR data type, below describes the meaning of the return value:

Table 2. Return values of the API

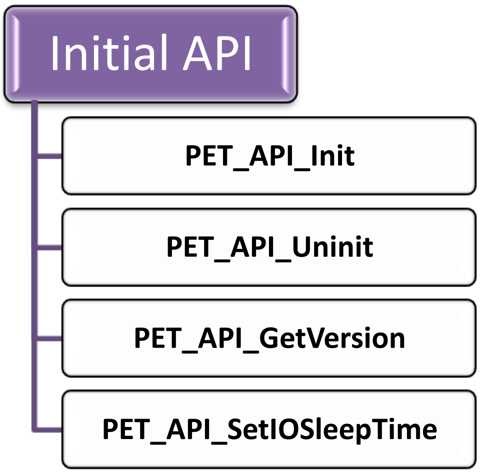
Return value	Define message
0	The function call was successfully executed.
1	Unknown error.
2	Driver not found..
3	Driver not loaded
4	Driver not loaded on network.
5	No supported platform.
6	Initial error
7	An invalid parameter
8	Read error
9	Write error
10	No support
11	Unavailable sleep time value.
12	Can't set/read I/O Permission.
13	Can't set/read back I/O Permission..
14	Not ready.
15	Unavailable sound.
16	DMI get miss.

✧ Function List

Initial API

First, the API function to do for different categories, and we will do for each API function detailed introduction and how to use. The users don't need to call any API function calls provided by the operating system API functions can directly call the PET API functions.

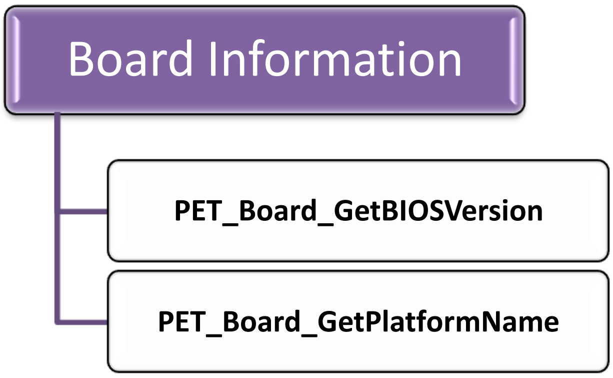
Prior to the use of PET API library, the user must initialize the API function calls to register, if you don't pre-initialized using PETAPI library will not have permission to access any of the information.



Board Information

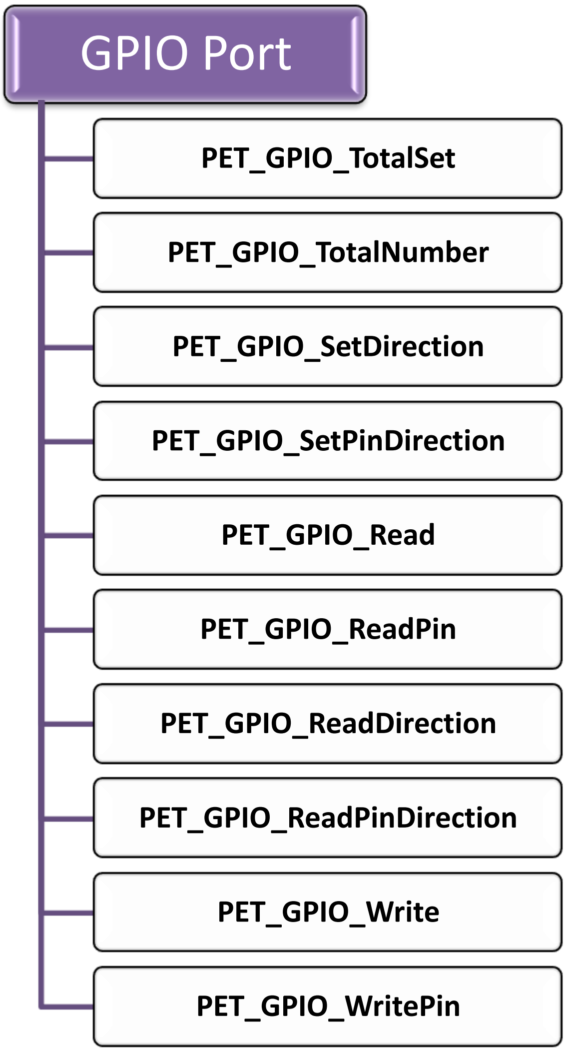
The PET API is based on the board concept. A board is a physical hardware component. At the moment each board must has BIOS that support functions for the underlying hardware.

The each board has a unique name that corresponds directly with the physical type of board. The function API can be used to query the boot time on your system and other information. So you can keep the handle by using API function for as long as you like. Later, according to new needs or where will be updated



GPIO Port

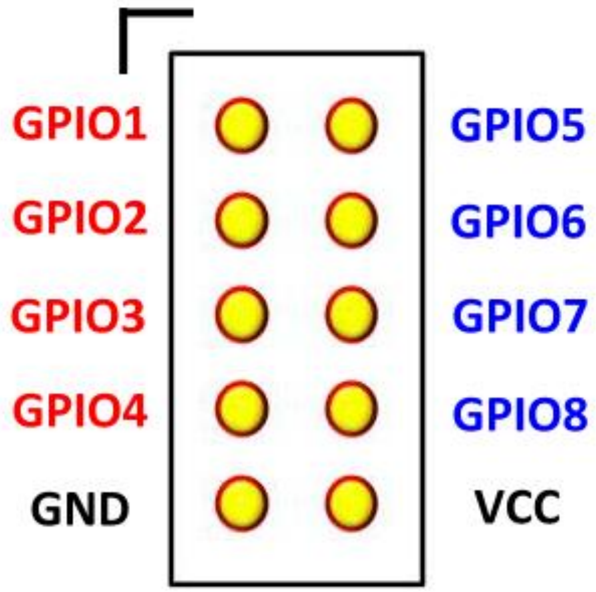
The GPIO function provides access to general purpose input output ports. Many boards don't have any user accessible I/O ports, so these functions can return I/O information or status.



Explanation GPIO code :

1. GPIO Set number

2. GPIO bit and GPIO pin mapping table



Example:

GPIO_SetDirection (1, 0xF0)

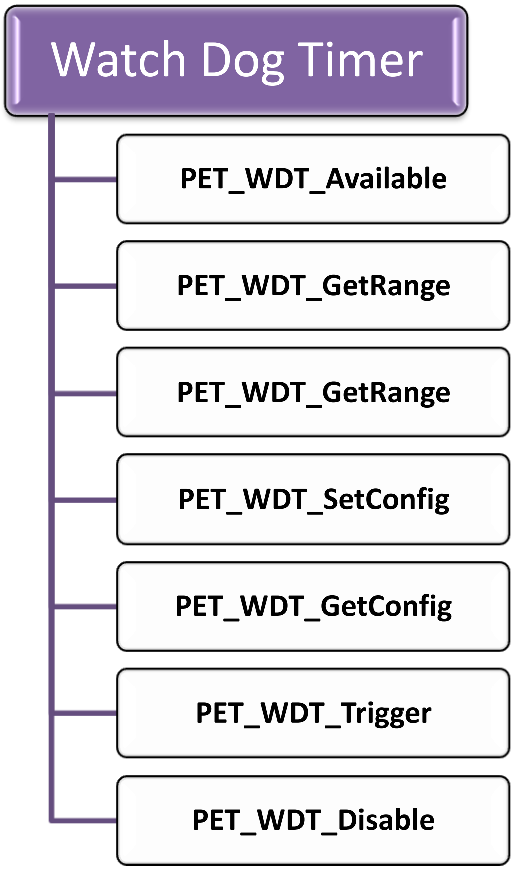
0xF0 = 11110000 (GPIO8 ~ GPIO1).

It means that GPIO8~GPIO5 is input, GPIO4~GPIO1 is output.

Watch Dog Timer

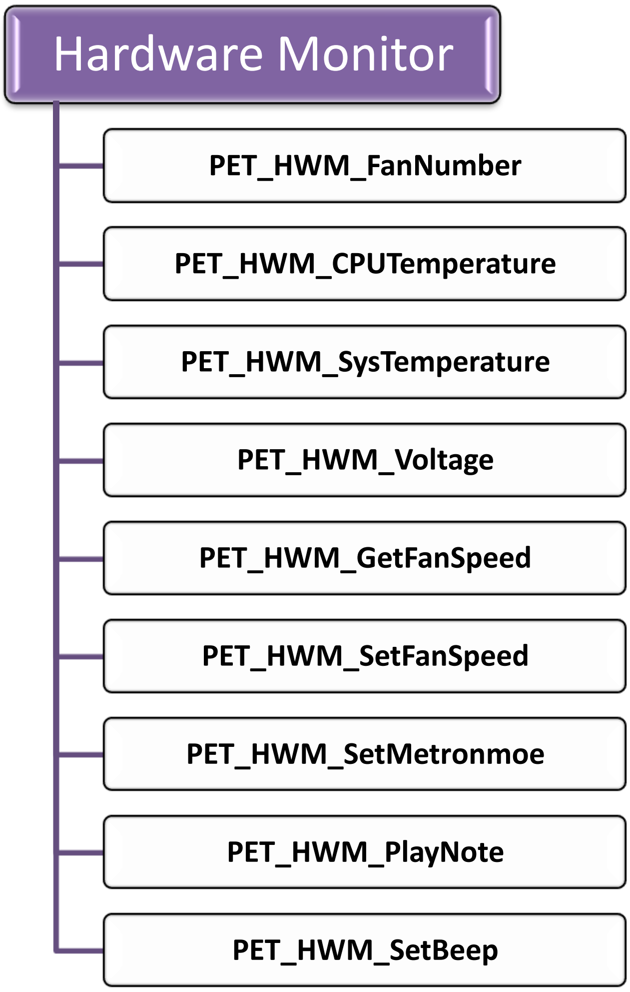
The hardware watch dog timer is a common feature of all Portwell board. In user application, they can all these API function with specific timeout values to start the watch dog timer countdown, meanwhile create a thread or timer to periodically refresh the timer.

If the application ever hangs, it will fail to refresh the timer and the watch dog reset will cause a system reboot.



Hardware Monitor

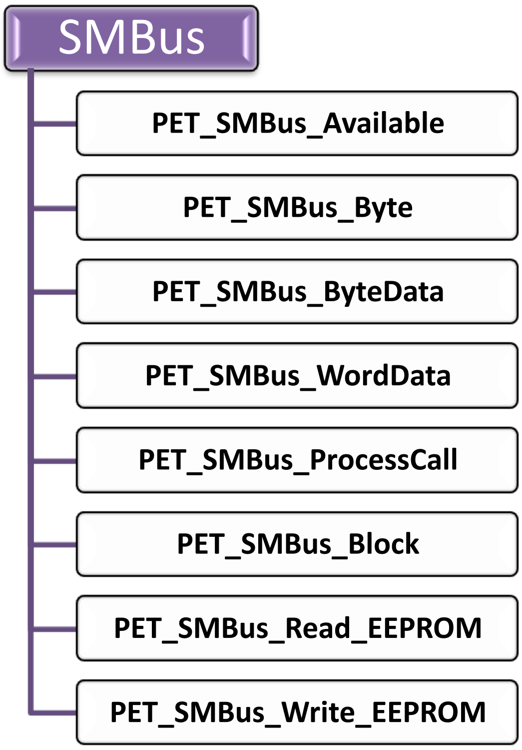
On many company need to measure or monitor hardware status, because the use of the environment isn't good, may at any time to monitor the status of the hardware itself, that just like people would regularly do health checks. So the PET library provides some API function call to support user monitor hardware, for example: CPU and North or PCH chipset's temperature or voltage. So when increase the temperature, the user can to control the fan speed to make the hardware down to a safe temperature range.



SMBus

According to previous introduction, the user should not be strange for the I^2C with knowledge of the hardware, so on the Portwell provides platform and libraries, so the user can free handle each one.

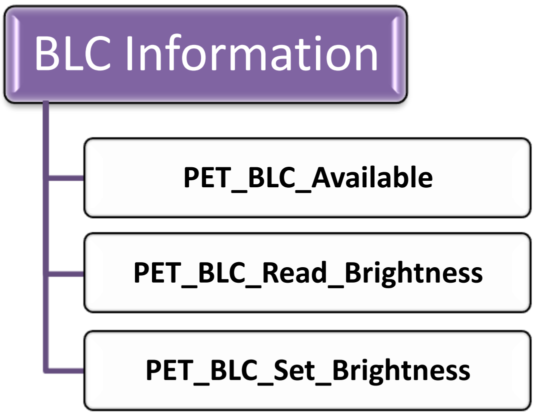
The PET SMBus function provides access to the onboard SMbus through Intel PCH or south chip. Note that since SMBus address may change you should not use these function to access any PET onboard device. Therefore, the user must have a clear understanding of SMBus devices connect. You should use these function only if you have your own devices connected to the onboard bus.



Backlight Control

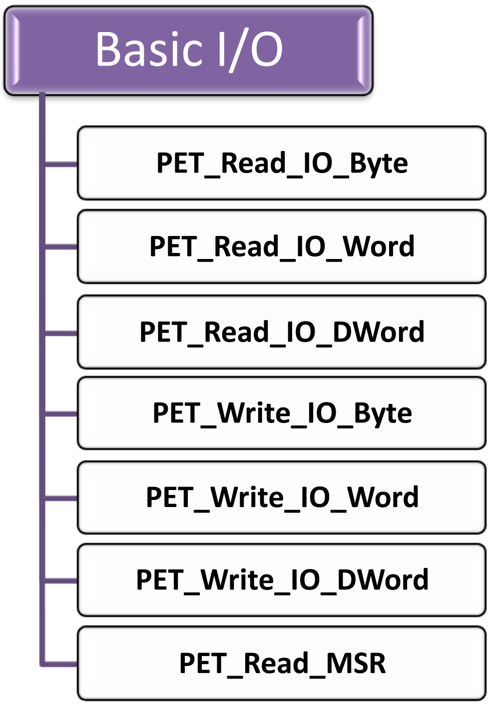
Applications that require programmatic control of the backlight brightness or provide controls for the user to do so should use this interface; otherwise, the system cannot query the current hardware brightness and may become unsynchronized.

The PET backlight control function provides access to the onboard LVDS brightness. Note: Not every board has support this feature



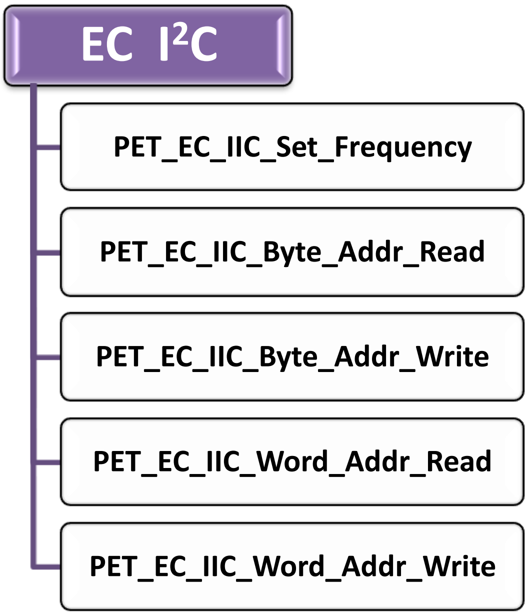
Basic I/O Access

The low level access function, it can directly to access register. Therefore, when using these functions to be especially careful, if incorrect value is written to cause the system crash.



EC I^2C

The PET EC I^2C function provides access to the onboard I^2C through Embedded controller.



✧ Function Introduction

Each function has a description, prototype, parameters, return value and sample code. You can research API function call for your needs.

➢ PET_API_Init

Declaration

Initial the application programming library of Portwell Engineering Toolkit, and setting environment parametric. You must to call function code before calling other functions.

Prototype

int PET_API_Init ();

Parameters

None.

Return Value

If the return value 0 is success, otherwise return an error code.

Sample Code

```
int initial_value = PET_API_Init ();
if ( initial_value != 0 )
{
        printf ("Initial error number is %d\n", initial_value);
        exit (1);
}
```

➢ PET_API_Uninit

Declaration

Before an application is end, it must un-initialized the PET libraries.

Prototype

int PET_API_Uninit ();

Parameters

None.

Return Value

If the return value 0 is success, otherwise return an error code.

Sample Code

```
int uninitial_value = PET_API_Uninit ();
if ( initial_value != 0 )
{
        printf (“Initial error number is %d\n”, initial_value);
        exit (1);
}
```

➢ PET_API_GetVersion

Declaration

Get PET API Library version. The user can query version is the latest version, if not update, you can exchange the new version of the library.

Prototype

int PET_API_GetVersion (char *api_version);

Parameters

api_version

[Out]receive API version.

Return Value

If the return value 0 is success, otherwise return an error code.

Sample Code

```
int test;
char api_version [10];
test = PET_API_GetVersion ( api_version );
if ( test == 0 )
        printf (“The PET API libraries version is %f\n”, version);
```

➢ PET_API_SetIOSleepTime

Declaration

Set sleep time on the I/O control. You can use this function to delay time on the I/O operating.

Prototype

int PET_API_SetIOSleepTime (int msec);

Parameters

msec

[In] sleep time value.

Return Value

If the return value 0 is success, otherwise return an error code.

Sample Code

```
int msec= 3;
/* Sleep 3 millisecond */
PET_API_SetIOSleepTime ( msec );
```

➢ PET_Board_GetBIOSVersion

Declaration

The Portwell inc. will from time to time update or debug, each over a period of time may release a new version of the BIOS, so customers can use this function to know the current BIOS version.

Prototype

```
int PET_Board_GetBIOSVersion ( char *bios_version );
```

Parameter

bios_version

[Out] receive BIOS version.

Return Value

If the return value 0 is success, otherwise return an error code.

Sample Code

```
int test;
char bios_version[20];
test = PET_Board_GetBIOSVersion ( bios_version );
if ( test == 0 )
    printf (“The PET BIOS version is %s\n”, bios_version);
```

➢ PET_Board_GetPlatformName

Declaration

The function call can get Portwell's board name.

Prototype

int PET_Board_PlatformName (char *name);

Parameter

name

[Out] receive platform name.

Return Value

If the return value 0 is success, otherwise return an error code.

Sample Code

```
char name[10];
int test    = PET_Board_GetPlatformName ( name );
if ( test == 0 )
        printf (“The platform name is %s\n”, name);
```

➢ PET_GPIO_TotalSet

Declaration

Get GPIO number set of the board.

Prototype

int PET_GPIO_TotalSet (int *gpio_set_number);

Parameter

gpio_set_number

[Out] GPIO set number.

Return Value

If the return value 0 is success, otherwise return an error code.

Sample Code

```
1.    int gpio_set_number = 0;
2.    int num = PET_GPIO_TotalSet (&gpio_set_number);
3.    if (num == 0)
4.        printf (“The GPIO Set is %d”, gpio_set_number);
```

PET_GPIO_TotalNumber

Declaration

Get number of the board's GPIO set.

Prototype

```
int PET_GPIO_TotalNumber (int gpio_set, int *available_pin_number);
```

Parameter

gpio_set

[IN] GPIO set number

available_pin_number

[Out] The number of the available GPIO pin.

Return Value

If the return value 0 is success, otherwise return an error code.

Sample Code

```
1.	int gpio_set = 0;
2.	int available_pin_number = 0;
3.	int error_code = PET_GPIO_TotalNumber (gpio_set,
	&available_pin_number);
4.	if ( error_code == 0)
5.		printf ("The GPIO%d %d pin is available", gpio_set,
	available_pin_number);
```

➢ PET_GPIO_SetDirection

Declaration

Set direction of all GPIO pin as input or output. (GPIO 1~8 port) Fixed inputs and fixed outputs cannot be changed. Each bit setting FASLE (0) indicates output and input TRUE (1) indicates input.

Prototype

int PET_GPIO_SetDirection (int gpio_set, unsigned char io_direction);

Parameter

gpio_set

[In] Set 1~n GPIO set.

io_direction

[In] Input 0 indicates output, 1 indicates input.

Return Value

If the return value 0 is success, otherwise return an error code.

Sample Code

```
int gpio_set ;
unsigned char io_direction;
printf ("GPIO set: ");
scanf("%x", &gpio_set);
printf("\nGPIO Direction:    )");
scanf("%x", &io_direction);
PET_GPIO_SetDirection (gpio_set, io_direction );
```

➢ PET_GPIO_SetPinDirection

Declaration

Set direction of one GPIO pin as input or output.

Prototype

int PET_GPIO_SetPinDirection (int gpio_set, int pin_num, unsigned char io_direction);

Parameter

gpio_set

[In] Set 1~n GPIO set.

pin_num

[In] specifies the pin of GPIO direction, ranging from 1~8.

io_direction

[In] Input 0 indicates output, 1 indicates input.

Return Value

If the return value (0) is success, otherwise return an error code.

Sample Code

```
int gpio_set = 0;
int pint_num = 0;
unsigned char io_direction;
printf("GPIO set: ");
scanf ("%x", &gpio_set);
printf ("PinNum (1 ~ 8): ");
scanf ("%x", &pin_num);
printf ("\n Direction: ( 0:output 1:input )");
scanf ("%x", &io_direction);
PET_IO_SetPinDirection( gpio_set, pin_num, io_direction );
```

➢ PET_ GPIO_Read

Declaration

Read all GPIO input pin status high or low (GPIO 1~8 port). Each bit corresponds to a pin indicates (bit 1 for pin 1, bit 2 for pin 2 etc.).

Prototype

int PET_GPIO_Read   (int gpio_set, unsigned char *value);

Parameter

gpio_set

[In]   Set 1~n GPIO set.

value

[Out]   Output the GPIO port value.

Return Value

If the return value (0) is success, otherwise return an error code.

Sample Code

```
int gpio_set = 0;
unsigned char value = 0,
int status = 0;
printf("GPIO set: ");
scanf("%x", &gpio_set);
status = PET_GPIO_Read (gpio_set, &value );
if ( status == 0)
        printf (“The port(87654321): 0x%x”, value);
```

➢ PET_GPIO_ReadPin

Declaration

Read GPIO input pin status high or low (GPIO 1~8 port).

Prototype

int PET_GPIO_ReadPin (int gpio_set, int pin_num, unsigned char * value);

Parameter

gpio_set

[In] Set 1~n GPIO set.

pin_num

[In] specifies the pin of GPIO direction, ranging from 1~8.

value

[Out] Point to a variable in which the pin status returns.

Return Value

If the return value (0) is success, otherwise return an error code.

Sample Code

```
unsgined char gpio_set = 0;
int pin_num =0;
int value = 0;
int status = 0;
printf ("GPIO set: ");
scanf ("%x", &gpio_set);
printf ("PinNum (1~8): ");
scanf ("%x", &pin_num);
status = PET_GPIO_ReadPin (gpio_set, pin_num, &value );
if( status == 0 )
        printf("\n The port %d value : %d", pin_num, value );
```

➢ PET_GPIO_ReadDirection

Declaration

Read direction of all GPIO pin as unput or output (GPIO1~8 port). Each bit that returns (0) indicates output and input (1) indicates input.

Prototype

int PET_GPIO_ReadDirection (int gpio_set, unsigned char *io_direction);

Parameter

gpio_set

[In] Set 1~n GPIO set.

io_direction

[Out] Each bit corresponds to a pin indicates (bit 1 for pin 1, bit 2 for pin 2 etc.). Value 0 as output, value 1 as input, example 0x80, bit 1~7 is output, bit 8 is input,

Return Value

If the return value (0) is success, otherwise return an error code.

Sample Code

```
int gpio_set = 0;
unsigned char io_direction =0;
int status = 0;
printf ("GPIO set: ");
scanf ("%x", &gpio_set);
printf (“0:output 1:input \n”);
status = PET_GPIO_Direction ( gpio_set, &io_direction);
if ( status == 0)
    printf (“Direction Port: 0x%x\n”, io_direction);
```

➢ PET_GPIO_ReadPinDirection

Declaration

Read direction of one GPIO pin as input or output.

Prototype

int PET_GPIO_ReadPinDirection (int gpio_set, int pin_num, unsigned char *io_direction);

Parameter

gpio_set

[In] Set 1~n GPIO set.

pin_num

[In] specifies the pin of GPIO direction, ranging from 1~8.

io_direction

[Out] Output (0) indicates output, output (1) indicates input.

Return Value

If the return value (0) is success, otherwise return an error code.

Sample Code

```
int gpio_set = 0;
int pin_num =0;
unsgined char io_direction =0
int status = 0;
printf ("GPIO set: ");
scanf ("%x", & gpio_set );
printf    (“PinNum (1~8):”);
scanf (“%x”, &pin_num);
status = PET_GPIO_ReadPinDirection( gpio_set ,    pin_num, & io_direction )
if ( ( status == 0)
     printf (“Direction (0:output 1:input) 0d\n”,    io_direction );
```

➢ PET_GPIO_Write

Declaration

Set all GPIO output pin status high or low (GPIO 1~8 Port). Each bit corresponds to a pin indicates (bit 1 for pin 1, bit 2 for pin 2 etc.).

Prototype

int PET_GPIO_Write (int gpio_set, unsigned char value);

Parameter

gpio_set

[In] Set 1~n GPIO set.

value

[In] Pointer to a variable in GPIO 1~8 status.

Return Value

If the return value (0) is success, otherwise return an error code.

Sample Code

```
int Gpio_set = 0;
unsigned char value =0;
int status = 0;
printf ("GPIO set: ");
scanf ("%x", &gpio_set);
printf (“status value : (0 output, 1 input)”);
scanf (“%x”, &value);
PET_GPIO_Write (gpio_set, value);
```

➢ PET_GPIO_WritePin

Declaration

Set one GPIO output pin as status high or low.

Prototype

int PET_GPIO_WritePin (int gpio_set, int pin_num, unsigned char value);

Parameter

gpio_set

[In] Set 1~n GPIO set.

pin_num

[In] Specifies the pin of GPIO demanded to be read, ranging from 1~8.

value

[In] Point to a variable in which the pin status.

Return Value

If the return value (0) is success, otherwise return an error code.

Sample Code

```
int gpio_set = 0;
int pin_num =0;
int value = 0;
printf ("GPIO set: ");
scanf ("%x", &gpio_set);
printf (“PortNum (1~8):”);
scanf (“%x”, & pin_num);
printf (“status value : (0 output, 1 input)”);
scanf (“%x”, &value);
PET_GPIO_WritePin (gpio_set, pin_num, value);
```

PET_WDT_Available

Declaration

It can verify the watch dog timer is available.

Prototype

int PET_WDT_Available ();

Parameter

None.

Return Value

If the return value (0) is success, otherwise return an error code.

Sample Code

```
1.  int test;
2.  test = PET_WDT_Available ();
3.  if ( test == 0)
4.      printf (“The WDT is available\n”);
5.  else
6.      printf (“The WDT is disable\n”);
```

➢ PET_WDT_GetRange

Declaration

Detect the watchdog timer type mode's maximum/minimum value.

Prototype

int PET_WDT_GetRange (int type, unsigned char *minimum, unsigned char *maximum);

Parameter

type

[In] The type specifies the watch dog timer count unit.

minimum

[Out] Minimum value.

maximum

[Out] Maximum value.

Return Value

If the return value (0) is success, otherwise return an error code.

Sample Code

```
unsigned char minimum, maximum;
PET_WDT_GetRange (SECOND_MODE, &minimum, &maximum);
printf ("The WDT seconde mode range: %d ~ %d (sec)\n",
    minimum, maximum);
```

➢ PET_WDT_SetConfig

Declaration

Once the watch dog has been activated, its timer begins to count down. The application has to periodically call PET_WDT_Trigger to refresh the timer before it expires, i.e. reload the watch dog timer within the specified timer out or the system will reboot when it counts down to 0.

Prototype

```
int PET_WDT_SetConfig (int type, unsigned char timeout);
```

Parameter

type

[In] The type specifies the watch dog timer count unit.

timeout

[In] it specifies a value in second or minute for the watch dog timer out.

Return Value

If the return value (0) is success, otherwise return an error code.

Sample Code

```
PET_WDT_SetConfig (SECOND_MODE, 10);    // 10 second
```

➢ PET_WDT_GetConfig

Declaration

Once the watch dog has been activated, its timer begins to count down. The API can get current type and timeout value.

Prototype

int PET_WDT_GetConfig (int *type, unsigned char *timeout);

Parameter

type

[Out] Two type: SECOND_MODE and MINUTE_MODE.

timeout

[Out] Timeout value.

Return Value

If the return value (0) is success, otherwise return an error code.

Sample Code

```
int wdt_type;
unsigned char timeout;
int test = PET_WDT_GetConfig ( &wdt_type , &timeout);
if ( test == 0)
printf ("WDT type:%d, timeout:%d\n", wdt_type, wdt_timeout);
```

➢ PET_WDT_Trigger

Declaration

Trigger watchdog timer or reload the watchdog timer to the timeout value by PET_WDT_SetConfig timeout value to prevent the system from rebooting.

Prototype

int PET_WDT_Trigger ();

Parameter

None.

Return Value

If the return value (0) is success, otherwise return an error code.

Sample Code

```
unsigned char timeout = 10;
int test = PET_WDT_SetConfig (SECOND_MODE, timeout);
if ( test == 0)
        printf (“PET_WDT_SetConfig is success\n”);
test = PET_WDT_Trigger();
if ( test == 0)
        printf("PET_WDT_Trigger is success”);
```

➢ PET_WDT_Disable

Declaration

Disable the watch dog and stop its timer count down.

Prototype

int PET_WDT_Disable ();

Parameter

None.

Return Value

If the return value (0) is success, otherwise return an error code.

Sample Code

```
unsigned char timeout =10;
PET_WDT_SetConfig (SECOND_MODE, timeout);
PET_WDT_Trigger ();
int test = PET_WDT_Disable ();
if ( test == 0)
        printf("PET_WDT_Disable is success”);
```

➢ PET_HWM_FanNumber

Declaration

Get CPU core number.

Prototype

int PET_HWM_FanNumber (int *fan_num);

Parameter

fan_num

[In]   Receive the fan number.

Return Value

If the return value (0) is success.

Sample Code

```
int test;
int fan_num;
test = PET_HWM_FanNumber ( &fan_num );
if (test == 0)
    printf(" The fan number is %d\n", fan_num);
```

➢ PET_HWM_CPUTemperature

Declaration

Read the current value of the CPU temperature sensors, but not every board can reads from msr mode.

Prototype

```
int PET_HWM_CPUTemperature ( int cpu_num, float *cputemp_value);
```

Parameter

cpu_num

[In] The system may have more than one CPU temperature measurement requirements, for example 1 or 2...etc.

0: From SIO mode to read CPU temperature

1: current not support

2: current not support

etc.

cputemp_value

[Out] Point to a variable in which this function returns the temperature.

Return Value

If the return value (0) is success,

Sample Code

```
float temperature = 0; /* Store Temperature */
int cpu_num = 0;
PET_HWM_CPUNumber    ( &cpu_num );
printf ("********** Temperature Measure ********\n");
HWMCPUTemperature(0, &temperature);
printf ("* CPU Temperature is %4.2f ('C)          *\n", temperature);
printf ("***************************************\n");
```

➢ PET_HWM_SysTemperature

Declaration

Read the current value of the system temperature sensors.

Prototype

int PET_HWM_SysTemperature (float *systemp_value);

Parameter

systemp_value

[Out] Point to a variable in which this function returns the temperature.

Return Value

If the return value (0) is success,

Sample Code

```
float temperature = 0; /* Store Temperature */
printf ("********** Temperature Measure ********\n");
PET_HWMGetSysTemperature(&temperature);
printf ("* SYSTIN Temperature is %4.2f ('C)    *\n", temperature);
printf ("****************************************\n");
```

➢ PET_HWM_GetVoltage

Declaration

Read the current value of one the voltage sensors, or get the type of available sensors.

Prototype

int PET_HWM_GetVoltage (int vol_type, float *vol_value);

Parameter

vol_type

[In] It specifies a voltage sensor to get value. It can be chose one such as VOL_VCORE, VOL_1P5V, VOL_3P3V, and VOL_12V or key voltage type number.

vol_value

[Out] Point to a variable in which this function returns the voltage in voltage.

Return Value

If the return value (0) is success,

Sample Code

```
 /* Store Voltage */
float voltage = 0;
/* Voltage API test and how to use */
printf ("\n*********** Voltage Measure ***********\n");
PET_HWMGetVoltage (VOL_VCORE, &voltage);
printf ("*          CPU Voltage is %3.2f (V)          *\n", voltage);
PET_HWMGetVoltage (VOL_5V, &voltage);
printf ("*          VCC Voltage is %3.2f (V)          *\n", voltage);
PET_HWMGetVoltage (VOL_3P3V, &voltage);
printf ("*          3V Voltage is %3.2f (V)          *\n", voltage);
PET_HWMGetVoltage (VOL_1P5V, &voltage);
printf ("*         1.5V Voltage is %3.2f (V)          *\n", voltage);
PET_HWMGetVoltage (VOL_12V, &voltage);
printf ("*          12V Voltage is %3.2f (V)          *\n", voltage);
printf ("***************************************\n");
```

➢ PET_HWM_GetFanSpeed

Declaration

Read the current value of the fan speed sensors, or get the types of available sensors.

Prototype

int PET_HWM_GetFanSpeed (int fan_type, int *fan_value);

Parameter

fan_type

[In] It specifies a voltage sensor to get value. It can be chose one such as CPUFAN, SYSFAN and AUXFAN.

fan_value

[Out] Point to a variable in which this function returns the fan speed value.

Return Value

If the return value (0) is success,

Sample Code

```
unsigned char fan_value = 0;
PET_HWMGetFanSpeed (SYSFAN, &fan_value);
printf ("* The SYS Fan Speed is %7d (rpm) *\n", (int) fan_value);
```

PET_HWM_SetFanSpeed

Declaration

Set the current value of the fan speed sensors.

Prototype

BYTE PET_HWM_SetFanSpeed (int fan_type, unsigned char fan_ratio);

Parameter

fan_type

[In] It specifies a voltage sensor to get value. It can be chose one such as CPUFAN, SYSFAN and AUXFAN.

fan_value

[In] The fan ratio. (0 ~ 100%)

Return Value

If the return value (0) is success,

Sample Code

```
unsigned char fan_value = 70;
PET_HWMSetFanSpeed (SYSFAN,    value );
```

➢ PET_HWM_SetMetronome

Declaration

Before use PET_HWM_PlayNote function to play your music, you can set the metronome value.

Prototype

int PET_HWM_SetMetronome (int bpm);

Parameter

bpm

[In] Set metronome value. (Default 120)

Return Value

If the return value (0) is success,

Sample Code

```
int test = PET_API_Init ();
if (int test == NOERROR)
        PET_HWM_SetMetronome (80);
```

➢ PET_HWM_PlayNote

Declaration

The function can play music by computer beep sound. This is a simply control function, that can control music level, note, key and beats by yourself.

Prototype

int PET_HWM_PlayNote (int level, int note, int up_key, float beats);

Parameter

level

[In] Set value rang is -1, 0, 1.

note

[In] Do(1) Re(2) Me(3) Fa(4) So(5) La(6) Si(7)

up_key

[In] Up key (1), Normal (0)

beats

[In] Set value range is 1 ~ 4

Return Value

If the return value (0) is success,

Sample Code

```
PET_HWM_SetMetronome (80);
PET_HWM_PlayNote (0, 5 , 0, 1);
```

➢ PET_HWM_SetBeep

Declaration

Set beep alarm function.

Prototype

int PET_HWM_SetBeep (float freq, int msec);

Parameter

freq

[In] Set frequency (Range: 500 ~ 10000).

msec

[In] Time value (Range: 1 ~ 15 sec).

Return Value

If the return value (0) is success,

Sample Code

```
int test = PET_API_Init ();
if (int test == NOERROR)
        PET_HWM_SetBeep (1000, 5); // 5 second
```

PET_SMBus_Available

Declaration

Verify whether the SMBus is available.

Prototype

int PET_SMBus_Available ();

Parameter

None.

Return Value

If the return value (0) is success,

Sample Code

1. int test;
2. test = PET_SMBus_Available ();
3. if (test == 0)
4. printf ("The SMBus is available\n");
5. else
6. printf ("The SMBus is disable\n");

➢ PET_SMBus_ByteData

Declaration

Read/Write a byte of data from the target slave device's register in the SMBus.

Prototype

int PET_SMBus_ByteData (int r_w, unsigned char slave_addr, unsigned char offset, unsigned char *value);

Parameter

r_w

[In] parameter: SM_READ or SM_WRITE.

slave_addr

[In] SMBus slave address, range from 0x00 – 0xFF.

offset

[In] SMBus register address, range from 0x00 – 0xFF.

value

[In/Out] Pointer to a variable in which the function read/write the bytes of data.

Return Value

If the return value (0) is success.

Sample Code

```
unsigned char slave_addr = 0x5E;
unsigned char offset = 0x30;
unsigned char data;
int test;
test = PET_SMBus_ByteData (SM_READ, slave_addr, offset, &data );
if (test == 0)
    printf (“Received data is %x\n”, data);
```

PET_SMbus_WordData

Declaration

Read/Write a word of data from the target slave device's register in SMBus.

Prototype

int PET_SMBus_WordData (int r_w, unsigned char slave_addr, unsigned char offset, unsigned int *value);

Parameter

r_w

[In] parameter: SM_READ or SM_WRITE.

slave_addr

[In] SMBus slave address, range from 0x00 – 0xFF.

offset

[In] SMBus register address, range from 0x00 – 0xFF.

value

[In/Out] Pointer to a variable in which the function read/write the bytes of data.

Return Value

If the return value (0) is success.

Sample Code

```
unsigned char slave_addr = 0x5E;
unsigned char offset = 0x30;
unsigned int data;
int test;
test = PET_SMBus_WordData (SM_READ, slave_addr, offset, &data );
if (test == 0)
    printf (“Received data is %x\n”, data);
```

➢ PET_SMBus_ProcessCall

Declaration

This function selects a device register (through the Comand Code), sends 16 bits of data to it, and reads 16 bits of data in return.

1	7	1	1	8	1	8	1	8	1	
S	Slave Address	Wr	A	Command Code	A	Data Byte Low	A	Data Byte High	A	…

1	7	1	1	8	1	8	1	1
Sr	Slave Address	Rd	A	Data Byte Low	A	Data Byte High	A	P
							1	

Figure 4. Process Call

Prototype

int PET_SMBus_ProcessCall (unsigned char slave_addr, unsigned char offset, unsigned int *w_value, unsigned int *r_value);

Parameter

slave_addr

[In] Specifies the device address, ranging from 0x00 – 0xFF.

offset

[In] SMBus register address, range from 0x00 – 0xFF.

w_value

[In] Pointer to a byte array which contains the bytes of data to be written.

r_value

[Out] Pointer to a byte array which contains the bytes of data to be read.

Return Value

If the return value (0) is success.

Sample Code

```
unsigned char slave_addr = 0x5E;
unsigned char offset = 0x30;
unsigned int w_value, r_value;
w_value = 0x5a;
int test;
test = PET_SMBus_ProcessCall (slave_addr, offset, &w_value, &r_value );
if (test == 0)
    printf (“Process Call received data is %x\n”, r_value);
```

➢ PET_SMBus_Block

Declaration

The Block Write function writes up to 32 bytes to a device, to a designated register that is specified through the Command Code. The amount of data is specified in the Byte Count.

1	7	1	1	8	1	8	1	8	1	
S	Slave Address	Wr	A	Command Code	A	Byte Count = N	A	Data Byte 1	A	...

8	1	...	1	1	1
Data Byte 2	A	...	Data Byte N	A	P

Figure 5. Block Write

The Block Read function reads a block of up to 32 bytes from a device, from a designated register that is specified through the Command Code. The amount of data is specified by the device in the Byte Count.

1	7	1	1	8	1	1	7	1	1	8	1	
S	Slave Address	Wr	A	Command Code	A	Sr	Slave Address	Rd	A	Byte Count = N	A	...

8	1	8	1	...	8	1	1
Data Byte 1	A	Data Byte 2	A	...	Data Byte N	A	P

Figure 6. Block Read

Prototype

int PET_SMBus_Block (int r_w, unsigned char slave_addr, unsigned char offset, int byte_count, unsigned char *value);

Parameter

r_w

[In] parameter: SM_READ or SM_WRITE.

slave_addr

[In] Specifies the 8 bit device address, ranging from 0x00 – 0xFF.

offset

[In] SMBus register address, range from 0x00 – 0xFF.

byte_count

[In] A Block Read or Write is allowed to transfer a maximum of 32 data bytes.

value

[In/Out] Pointer to a byte array which contains the bytes of data to be written.

Return Value

If the return value (0) is success.

Sample Code

```
unsigned char slave_addr = 0x5E;
unsigned char offset = 0x30;
int byte_count = 3;
unsigned char value [32];
int test, i;
test = PET_SMBus_Block (SM_READ, slave_addr, offset, byte_count, value );
if (test == 0)
{
    for (i=0; i<32; i++)
            printf ("Data %d is %x\n", i, buffer [i]);
}
```

➢ PET_SMbus_Read_EEPROM

Declaration

The function can read EEOROM of the 2 byte address, if the memory address is 1 byte, you can use PET_SMbus_ByteData to access.

Prototype

int PET_SMBus_Read_EEPROM (unsigned char slave, unsigned short address, int length, unsigned char *value);

Parameter

slave_addr

[In] SMBus slave address, range from 0x00 – 0xFF.

address

[In] EEPROM's address (2 byte)

length

[In] SMBus register address, range from 0x00 – 0xFF.

value

[Out] read EEPROM data.

Return Value

If the return value (0) is success.

Sample Code

None

➢ PET_SMbus_Write_EEPROM

Declaration

The function can write EEOROM of the 2 byte address, if the memory address is 1 byte, you can use PET_SMbus_ByteData to access.

Prototype

int PET_SMBus_Write_EEPROM (unsigned char slave, unsigned short address, int length, unsigned char *value);

Parameter

slave_addr

[In] SMBus slave address, range from 0x00 – 0xFF.

address

[In] EEPROM's address (2 byte)

length

[In] SMBus register address, range from 0x00 – 0xFF.

value

[In] Write data in EEPROM.

Return Value

If the return value (0) is success.

Sample Code

None

➢ PET_BLC_Read_Brightness

Declaration

The function can read current brightness ratio of the LVDS.

Prototype

int PET_BLC_Read_Brightness (unsignedchar *blc_ratio);

Parameter

blc_ratio

[In] Brightness ratio, range from 0% ~ 100%.

Return Value

If the return value (0) is success,

Sample Code

```
1.  int error_code;
2.  unsigned char blc_ratio;
3.  error_code = PET_BLC_Read_Brightness (&blc_ratio);
4.  if ( error_code == 0)
5.      printf ("The brightness ratio is %d%%\n", blc_ratio);
6.  else
7.      printf ("Read failed. Error number %d\n", error_code);
```

➢ PET_BLC_Set_Brightness

Declaration

The function can set brightness ratio on the board's LVDS.

Prototype

int PET_BLC_Set_Brightness (unsignedchar *blc_ratio);

Parameter

blc_ratio

[Out] Brightness ratio, range from 0% ~ 100%.

Return Value

If the return value (0) is success,

Sample Code

```
1.  int error_code;
2.  unsigned char blc_ratio = 80;
3.  error_code = PET_BLC_Set_Brightness (&blc_ratio);
4.  if ( error_code == 0)
5.      printf ("Brightness set %d%% successful\n", blc_ratio);
6.  else
7.      printf ("Set failed. Error number %d\n", error_code);
```

➢ PET_EC_IIC_Set_Frequency

Declaration

The function can set EC IIC frequency.

Prototype

int PET_EC_IIC_Set_Frequency (EC_FREQUENCY fquency);

Parameter

fquency

[In] Three mode, there are F400K, F100K and F50K (Hz).

Return Value

If the return value (0) is success.

Sample Code

1. int error_code;
2. error_code = PET_EC_IIC_Set_Frequency (F400K);
3. if (error_code == 0)
4. printf (“Set IIC frequency is successful \n”);

➢ PET_EC_IIC_Byte_Addr_Read

Declaration

If the I^2C device access address range is 0x00 ~ 0xFF, you can use the function to read, which exists on EC I^2C.

Prototype

int PET_EC_IIC_Byte_Addr_Read (unsigned char slave_addr, unsigned int offset, unsigned char *value, int length)

Parameter

slave_addr

[In] I^2C device slave address

offset

[In] I^2C device address

value

[Out] Pointer to it in which the function store the data.

length

[In] data size (1 ~ 32 byte)

Return Value

If the return value (0) is success.

Sample Code

```
int length=8;
int slaveAddress = 0xAE, address=0x00;
 error_code=PET_EC_IIC_Byte_Addr_Read(slaveAddress,address,value,length);
 if(error_code==0)
 {
        printf("Read:");
        for(j=0;j<length;j++)
        {
               printf("%X,",value[j]);
        }
        printf("from %X\n",address);
        address+=length;
}
else
{
      printf("Read Error\n");
}
```

➢ PET_EC_IIC_Byte_Addr_Write

Declaration

If the I^2C device access address range is 0x00 ~ 0xFF, you can use the function to write, which exists on EC I^2C.

Prototype

int PET_EC_IIC_Byte_Addr_Write (unsigned char slave_addr, unsigned int offset, unsigned char *value, int length)

Parameter

slave_addr

[In] I^2C device slave address

offset

[In] I^2C device address

value

[Out] Pointer to it in which the function write the data.

length

[In] data size (1 ~ 32 byte)

Return Value

If the return value (0) is success.

Sample Code

```
int i;
int slaveAddress = 0xAE, address=0x00;
for(i=0;i<length;i++)
        value[i]=i;

  error_code=PET_EC_IIC_Byte_Addr_Write(slaveAddress,address,value,length);
  if(error_code==0)
  {
        printf("Write OK\n");
  }
  else
  {
        printf("Write Error\n");
  }

```

➢ PET_EC_IIC_Word_Addr_Read

Declaration

If the I^2C device access address range is 0x0000 ~ 0xFFFF, you can use the function to read, which exists on EC I^2C.

Prototype

int PET_EC_IIC_Word_Addr_Read (unsigned char slave_addr, unsigned int offset, unsigned char *value, int length)

Parameter

slave_addr

[In] I^2C device slave address

offset

[In] I^2C device address

value

[Out] Pointer to it in which the function store the data.

length

[In] data size (1 ~ 32 byte)

Return Value

If the return value (0) is success.

Sample Code

```
int length=8;
int slaveAddress = 0xAE, address=0x1000;
 error_code=PET_EC_IIC_Byte_Addr_Read(slaveAddress,address,value,length);
 if(error_code==0)
 {
         printf("Read:");
         for(j=0;j<length;j++)
         {
                 printf("%X,",value[j]);
         }
         printf("from %X\n",address);
         address+=length;
}
else
{
        printf("Read Error\n");
}
```

➢ PET_EC_IIC_Word_Addr_Write

Declaration

If the I^2C device access address range is 0x00 ~ 0xFF, you can use the function to write, which exists on EC I^2C.

Prototype

int PET_EC_IIC_Byte_Word_Write (unsigned char slave_addr, unsigned int offset, unsigned char *value, int length)

Parameter

slave_addr

[In] I^2C device slave address

offset

[In] I^2C device address

value

[Out] Pointer to it in which the function write the data.

length

[In] data size (1 ~ 32 byte)

Return Value

If the return value (0) is success.

Sample Code

```
int i;
int slaveAddress = 0xAE, address=0x1000;
for(i=0;i<length;i++)
        value[i]=i;

error_code=PET_EC_IIC_Word_Addr_Write(slaveAddress,address,value,length);
if(error_code==0)
{
        printf("Write:");
        for(j=0;j<length;j++)
        {
                printf("%X,",value[j]);
                value[j]+=length;
        }
        printf("to %X\n",address);
        address+=length;
}
else
{
        printf("Write Error\n");
}
```

➢ PET_Read_IO_Byte

Declaration

Read low level I/O to read one byte register.

Prototype

int PET_Read_IO_Byte (unsigned short addr, unsigned char *data);

Parameter

addr

[In]   I/O address

data

[Out]   Store register data

Return Value

If the return value (0) is success.

Sample Code

```
1.    unsigned char addr = 0x40, data;
2.    int error_code = PET_Read_IO_Byte (addr, &data);
3.    if (error_code == 0)
4.        printf ("The address 0x%x is %x\n", addr, data);
```

➢ PET_Read_IO_Word

Declaration

Read low level I/O to read two byte register.

Prototype

int PET_Read_IO_Word (unsigned short addr, unsigned short *data);

Parameter

addr

[In] I/O address

data

[Out] Store register data

Return Value

If the return value (0) is success.

Sample Code

```
unsigned short addr = 0x40, data;
int error_code = PET_Read_IO_Word (addr, &data);
if (error_code == 0)
    printf ("The address 0x%x is %x\n", addr, data);
```

➢ PET_Read_IO_DWord

Declaration

Read low level I/O to read long register.

Prototype

int PET_Read_IO_DWord (unsigned short addr, unsigned long *data);

Parameter

addr

[In] I/O address

data

[Out] Store register data

Return Value

If the return value (0) is success.

Sample Code

```
1. unsigned char addr = 0x40;
2. unsigned long data;
3. int error_code = PET_Read_IO_Byte (addr, &data);
4. if (error_code == 0)
5.     printf (“The address 0x%x is %lx\n”, addr, data);
```

➢ PET_Write_IO_Byte

Declaration

Read low level I/O to write one byte register.

Prototype

int PET_Write_IO_Byte (unsigned short addr, unsigned char data);

Parameter

addr

[In] I/O address

data

[In] Data will be write in register

Return Value

If the return value (0) is success.

Sample Code

```
1.  unsigned char addr = 0x40, data = 0x77;
2.  int error_code = PET_Write_IO_Byte (addr, data);
3.  if (error_code == 0)
4.      printf (“Write data is successful\n”);
```

➢ PET_Write_IO_Word

Declaration

Read low level I/O to write two byte register.

Prototype

int PET_Write_IO_Word (unsigned short addr, unsigned short data);

Parameter

addr

[In] I/O address

data

[Out] Data will be write in register

Return Value

If the return value (0) is success.

Sample Code

1. **unsigned short addr = 0x40, data = 0x1234;**
2. **int error_code = PET_Write_IO_Word (addr, data);**
3. **if (error_code == 0)**
4. **printf ("Write data is successful\n");**

➢ PET_Write_IO_DWord

Declaration

Read low level I/O to write long register.

Prototype

int PET_Write_IO_DWord (unsigned short addr, unsigned long data);

Parameter

addr

[In] I/O address

data

[Out] Store register data

Return Value

If the return value (0) is success.

Sample Code

```
1.    unsigned char addr = 0x40, data = 0x12345678;
2.    unsigned long data;
3.    int error_code = PET_Write_IO_DWord (addr, &data);
4.    if (error_code == 0)
5.        printf ("The address 0x%x is %lx\n", addr, data);
```

➢ PET_Read_MSR

Declaration

Read CPU MSR register.

Prototype

int PET_Read_MSR (unsigned long index, unsigned long *eax, unsigned long *edx);

Parameter

index

[In] CPU MSR Address

eax

[Out] High DWord value

edx

[Out] Low DWord value

Return Value

If the return value (0) is success.

Sample Code

None.