

AR-B6002FLCT

Software Installation and Programming Guide

Ver. 1.0

Author: Charlie Chan

Date: 2012/6/27

Revision History

Version	Author	Date	Description
Rev. 1.0	Charlie Chan	2012/6/27	The initial version.

Table of Content

1.	Introduction.....	1
1.1.	CAN bus.....	1
1.1.1.	Overview	1
1.1.2.	Installation Procedure of CAN Bus Driver.....	1
1.1.3.	The CAN bus APIs.....	1
1.1.4.	CAN Message Format.....	2
1.2.	GPIO and Watchdog	5
1.2.1.	Overview	5
1.3.	Power Subsystem	6
1.3.1.	Overview	6
2.	File Descriptions	6
2.1.	CAN Bus	7
2.2.	GPIO and Watchdog	7
2.3.	Power Subsystem	7
3.	API List and Descriptions	8
3.1.	CAN Bus	8
3.2.	GPIO and Watchdog	12
3.2.1.	GPIO	12
3.2.2.	Watchdog	16
3.3.	Power Subsystem	19
	Appendix A	28

1. Introduction

1.1. CAN bus

1.1.1. Overview

The CAN bus APIs provide interfaces to CAN bus subsystem. By invoking these APIs, programmers can implement applications which have the functions listed below:

1. Set the BAUD rate.
2. Send the CAN packages over the CAN bus.
3. Receive the CAN packages via the CAN bus hardware interface.

In this CAN bus API package, we provides:

1. On Linux platform:
Linux driver module of CAN bus subsystem and the driver load / unload scripts.
On Windows platform:
Device driver and install program of CAN bus subsystem.
2. API header file.
API libraries in static library format and shared library format.
3. CAN bus test utility and its source code.

1.1.2. Installation Procedure of CAN Bus Driver

On Linux platform:

1. Change to the 'root' user account.
2. In the 'driver' directory, execute the script 'modld'.
3. Execute 'lsmod'.
4. Make sure 'arb6002' is in the module list.
5. If the driver is no longer needed, execute the script 'modul' to unload the driver.

On Windows platform:

1. In the driver directory, execute the '**SetupDriver.exe**' program.

1.1.3. The CAN bus APIs

Before executing the applications which invoke the CAN bus APIs, users should make sure that the Linux device driver or the Windows device driver of CAN bus has been installed.

On Linux platform, after successfully installing the device driver, a character device node named "/dev/can0" will be created automatically. The APIs open the device node "/dev/can0" implicitly so acquiring a file descriptor of "/dev/can0" by users is not necessary. In order not to degrade the performance of the CAN bus

subsystem, the device node “/dev/can0” is limited to be opened at most once at any moment, i.e., if application A accesses CAN bus via the APIs, the application B which either tries to open ‘/dev/can0’ or uses CAN bus API will result in failure.

On Windows platform, after successfully installing the device driver, there is a device which shows ‘CAN Bus Driver’ in the ‘Device Manager’. The APIs on Windows platform open this device implicitly. User can call the APIs directly without opening the CAN Bus subsystem device.

1.1.4. CAN Message Format

Windows

```
typedef struct _T_CANBUS_MSG {

    DWORD    dwFlag;
    DWORD    dwObjectNumber;
    DWORD    dwMsgID;
    DWORD    dwSecond;
    DWORD    dwMicroSecond;
    WORD     wDataLen;
    BYTE     szData[ 8 ];

} T_CANBUS_MSG;
```

To transmit a CAN package, the programmer has to fill in the fields in the variable of type T_CANBUS_MSG and pass this T_CANBUS_MSG variable as an argument to invoke the APIs. The fields in CAN message are described below:

dwFlag:

This field holds the information of message type. Programmers can set the message type as:

1. Standard Data Frame:

```
T_CANBUS_MSG CanMsg;
CanMsg.dwFlag = 0x00000000;
```

2. Remote Transmission Request in Standard Data Frame format

```
T_CANBUS_MSG CanMsg;
CanMsg.dwFlag = 0x00000000;
CanMsg.dwFlag |= FLAG_RTR;
```

3. Extended Data Frame:

```
T_CANBUS_MSG CanMsg;
CanMsg.dwFlag |= FLAG_EXT;
```

4. Remote Transmission Request in Extended Data Frame format

```
T_CANBUS_MSG CanMsg;
CanMsg.dwFlag |= (FLAG_EXT | FLAG_RTR);
```

dwObjectNumber:

This field is reserved for holding a message communication object number.

dwMsgID:

CAN message ID.

dwSecond, dwMicroSecond:

When a CAN package is received, the CAN device driver will annotate a timestamp to the timestamp field in the `canmsg_t` variable and return this `canmsg_t` variable to the caller.

wDataLen:

The number of the data bytes which are sent or received in the 'szData' field of CAN message. This field is necessary while transmitting a Standard or Extended Data Frame. Programmers have to explicitly set up this field. The length of data is 0~8.

For example:

```
T_CANBUS_MSG msg;

msg.szData[0] = 0xa1;
msg.szData[1] = 0xb2;
msg.szData[2] = 0xc3;

msg.wDataLen = 3;
```

szData:

The byte array which holds the message data.

Linux

```
// TPE DEFINE
```

```
typedef char      i8;
typedef unsigned char  u8;
typedef short     i16;
typedef unsigned short u16;
```

```

typedef unsigned bng    u32;
typedef int             i32;

typedef struct timeval {
    long tv_sec;
    long tv_usec;
} timeval;

typedef struct {
    i32      flags;
    i32      cob;
    u32      id;
    struct timeval timestamp;
    i16      length;
    u8       data[8];
} canmsg_t;

```

To transmit a CAN package, the programmer has to fill in the fields in the variable of type `canmsg_t` and pass this `canmsg_t` variable as an argument to invoke the APIs. The fields in CAN message are described below:

flags:

This field holds the information of message type. Programmers can set the message type as:

5. Standard Data Frame:

```

canmsg_t msg; // Declare a variable 'msg' of type 'canmsg_t'
msg.flags = 0; // Setting the flags field to 0 defines the 'msg' as an
               // ordinary standard data frame.

```

6. Remote Transmission Request in Standard Data Frame format

```

canmsg_t msg;
msg.flags = 0; // Setting the flags field to 0 defines the 'msg' as an
               // ordinary standard data frame.

msg.flags = msg.flags | MSG_RTR; // Enable the RTR flag.

```

7. Extended Data Frame:

```

canmsg_t msg;
msg.flags = 0 | MSG_EXT; // Setting the EXT flag in the 'flags' field
                          // defines the 'msg' as an extended data frame.

```

8. Remote Transmission Request in Extended Data Frame format

```

canmsg_t msg;

```

```
msg.flags = 0 | MSG_EXT | MSG_RTR; // Enable the RTR flag.
```

cob:

This field is reserved for holding a message communication object number.

id:

CAN message ID.

timestamp:

When a CAN package is received, the CAN device driver will annotate a timestamp to the timestamp field in the `canmsg_t` variable and return this `canmsg_t` variable to the caller.

length:

The number of the data bytes which are sent or received in the 'data' field of CAN message. This field is necessary while transmitting a Standard or Extended Data Frame. Programmers have to explicitly set up this field. The length of data is 0~8.

For example:

```
canmsg_t msg;
```

```
msg.data[0] = 0xa1;
```

```
msg.data[1] = 0xb2;
```

```
msg.data[2] = 0xc3;
```

```
msg.length = 3;
```

data:

The byte array which holds the message data.

1.2. GPIO and Watchdog**1.2.1. Overview**

AR-B6002 provides both a GPIO interface and a Watchdog timer. Users can use the GPIO and Watchdog APIs to configure and to access the GPIO interface and the Watchdog timer. The GPIO has four input pins and four output pins. The Watchdog timer can be set to 1~255 seconds. Setting the timer to zero disables the timer. The remaining seconds of the timer to reboot can be read from the timer.

In this GPIO and Watchdog package, on Linux and Windows platform, we provides:

1. API source code.
2. GPIO and Watchdog test utility and the utility source code.

1.3. Power Subsystem

1.3.1. Overview

When the AR-B6002 is at Power Mode 15, the Power Subsystem APIs can be used to get and set the configuration of power subsystem. By invoking the Power Subsystem APIs, the users can:

1. Get the current status of ignition (ON or OFF).
2. Set the Power-On mode. This setting will be kept in the power subsystem and will take effect at next system boot.
3. From the power subsystem, get the stored setting of Power-On mode.
4. Get or set the time of Hard Off delay in seconds or in minutes.
5. Get or set the time of Soft Off delay in seconds or in minutes
6. Get the battery voltage.
7. Get the version number of the firmware of the Power Subsystem.
8. Set the Hard Off delay and Soft Off delay to the default value.

The power subsystem connects to the main system via the COM6. The Linux's default supported COM interfaces are COM1~COM4. The Power Subsystem APIs implicitly communicate with power subsystem through COM6. Users must take extra steps to configure Linux kernel in order to support COM6. Please refer to Appendix A for more information. Users don't need extraordinary setup on Windows platform to support COM6.

In this Power Subsystem package, we provide:

1. The APIs to access power subsystem and the source code of the APIs.
2. The utility and source code to monitor and set up power modes, ignition status, and power-off time.
3. On Linux platform, the Makefile to create API libraries and utility.

2. File Descriptions

On Windows platform, the following files are applied for CAN bus, GPIO, Power/PIC and WatchDog Timer.

1. SuperIoAPI.h

The header file of the APIs and macro definition. This header file is an aggregate header which includes APIs declarations and macros for CAN Bus, GPIO, Watchdog, and Power Subsystem.

2. SuperIoLib.lib

The API library in static library format. This library is an aggregate library. It

includes APIs for CAN Bus, GPIO, Watchdog, and Power Subsystem.

3. SuperIoLib.dll

The API library in dynamically linked library format. This library is an aggregate library. It includes APIs for CAN Bus, GPIO, Watchdog, and Power Subsystem.

2.1. CAN Bus

On Linux platform:

2. AGC_LIB.h
The header file of the API and macro definitions.
3. errcode.h
The macro definitions of returned error code.
4. libAGC_LIB.a
The API library in static library format.
5. libAGC_LIB.so
The API library in shared library format.
6. main.c
The source code of the utility.
7. Makefile

2.2. GPIO and Watchdog

On Linux platform:

1. sio_acce.c
The source code of the Watchdog and GPIO APIs for accessing the SuperIO.
2. sio_acce.h
This file includes the declarations of the APIs and macro definitions.
3. main.c
The source code of the utility.
4. Makefile

2.3. Power Subsystem

On Linux platform:

1. pwr_acce.c
The source code of the APIs for accessing the power subsystem.
2. pwr_acce.h
This file includes the declarations of the APIs and macro definitions.
3. main.c
The source code of the utility.
4. Makefile

3. API List and Descriptions

3.1. CAN Bus

Windows:

Notice: Before using CAN bus APIs, be sure CAN bus driver was installed successfully.

BOOL WINAPI CanBus_Open();

Remarks:

You must call this function at start-up of application to open CAN bus device.

Parameters:

(None)

Return Value:

Returns TRUE on success.

If the CAN bus driver has not installed, return FALSE.

void WINAPI CanBus_Close();

Remarks:

You must call this function at end of application to close CAN bus device.

Parameters:

(None)

Return Value:

(None)

BOOL WINAPI CanBus_SetBaudrate(int nBaudrate);

Remarks:

Call this function to set CAN bus baudrate.

Parameters:

nBaudrate

Speed of transferring CAN bus messages.

Value from CANBUS_BAUDRATE_10K to CANBUS_BAUDRATE_1000K.

Return Value:

Returns TRUE on success, FALSE on failure.

```
BOOL WINAPI CanBus_GetMessage( T_CANBUS_MSG* pMsg );
```

Remarks:

Call this function to receive CAN bus message.

Parameters:

pMsg

Pointer of structure T_CANBUS_MSG, to retrieve CAN bus message.

Return Value:

Returns TRUE on success, FALSE on failure.

```
BOOL WINAPI CanBus_SendMessage( T_CANBUS_MSG* pMsg );
```

Remarks:

Call this function to send CAN bus message.

Parameters:

pMsg

Pointer of structure T_CANBUS_MSG, to send out CAN bus message.

Set dwFlag of T_CANBUS_MSG to 0x00000000 if message ID length is 11 bits.

Set dwFlag of T_CANBUS_MSG to 0x00000004 if message ID length is 29 bits.

Set sending data in szData[] of T_CANBUS_MSG.

Set wDataLen of T_CANBUS_MSG to determine the byts of szData[] will be sent.

Return Value:

Returns TRUE on success, FALSE on failure.

Linux:**1. Syntax:**

```
unsigned int sendCanMessages( canmsg_t *buffer, u8 count )
```

Description: This function sends out CAN packages over the CAN bus.

Parameters: If there is more than one CAN package to send, these CAN packages are stored in a 'canmsg_t' array. This function sends out packages in a sequential fashion. The memory address of the first CAN package to send is pointed at by the parameter 'buffer'. The number of CAN packages to send is indicated by the parameter 'count'. If the resource of sending out the CAN packages is temporarily unavailable, the process which invokes this function will be blocked (Block I/O) until the resource is available again.

Return Value: If this function sends out the packages successfully, it returns ERROR_API_SUCC. If this function fails to open the CAN device node, it returns ERROR_API_CAN_OPEN_FAIL. If this function has any problem with sending out the CAN packages, it returns ERROR_API_CANSENDERMESSAGES.

Here is an example:

If the CAN packages in the array 'canAry[]' have been initialized. The code listed below will send out the CAN packages in the 'canAry[]' over the CAN bus.

```
unsigned int result = 0;
canmsg_t canAry[30];
/* ...
   Initialize the CAN packages in the canAry[30]
*/
result = sendCanMessages( canAry, 30 );
if( result == ERROR_API_CANSENDERMESSAGES ||
    result == ERROR_API_CAN_OPEN_FAIL )
    fprintf( stderr, "Send CAN package error!\n");
```

2. Syntax:

```
unsigned int getCanMessages( canmsg_t *buffer, u8 count )
```

Description: This function receives CAN packages from the CAN bus subsystem.

Parameters: This function stores received CAN packages sequentially at an array of type 'canmsg_t'. The number of packages to receive is indicated by the parameter 'count'. Before finishing receiving 'count' packages, the process which invokes this function will be temporarily blocked (Block I/O) if there is no incoming CAN package.

Return Value: If this function receives the packages successfully, it returns ERROR_API_SUCC. If this function fails to open the CAN device node, it returns ERROR_API_CAN_OPEN_FAIL. If this function has any problem with receiving the CAN packages, it returns ERROR_API_CANGETMESSAGES.

Here is an example:

If the array 'canAry[]' of type 'canmsg_t' has been declared and allocated. The code listed below will receive 30 CAN packages from the CAN bus subsystem and stores the packages in the 'canAry[]'.

```
unsigned int result = 0;
canmsg_t canAry[30];

result = getCanMessages( canAry, 30 );
if( result == ERROR_API_CANGETMESSAGES ||
    result == ERROR_API_CAN_OPEN_FAIL )
    fprintf( stderr, "Send CAN package error!\n");
```

3. Syntax:

```
unsigned int configCan( i32 baud )
```

Description: This function sets up the speed (Baud rate) of sending and receiving CAN packages.

Parameters: The parameter 'baud' could be: (the unit is Kbps)

10 , 20 , 50 , 100 , 125 , 250 , 500 , 800 , 1000

The default speed is 125 Kbps.

Return Value: This function returns ERROR_API_SUCC if it set the Baud rate

successfully. If this function fails to open the CAN device node, it returns `ERROR_API_CAN_OPEN_FAIL`. If the inputted Baud rate is not any one of the Baud rate listed above, it will return `ERRMSG(ERROR_API_CANCONFIG, ERROR_GEN_INPUT_DATA)`. If it has any other problem with setting the Baud rate, it returns `ERROR_GEN_DEVICE_FAIL`.

3.2. GPIO and Watchdog

3.2.1. GPIO

Windows:

Notice: Before using GPIO APIs, be sure Super I/O driver has installed successfully.

```
#define GPO_BIT_0          0
#define GPO_BIT_1          1
#define GPO_BIT_2          2
#define GPO_BIT_3          3
#define GPI_BIT_4          4
#define GPI_BIT_5          5
#define GPI_BIT_6          6
#define GPI_BIT_7          7
```

BOOL WINAPI SuperIo_Open();

Remarks:

You must call this function at start-up of application to open Super I/O device.

Parameters:

(None)

Return Value:

Returns TRUE on success.

If the Super I/O driver has not installed, return FALSE.

void WINAPI SuperIo_Close();

Remarks:

You must call this function at end of application to close Super I/O device.

Parameters:

(None)

Return Value:

(None)

```
BOOL WINAPI Gpio_GetBitValue( int nBit, BYTE* pValue );
```

Remarks:

Call this function to get GPIO pins value.

Parameters:

nBit

Stand for GPIO pins, value GPO_BIT_0 ~ GPO_BIT_7.

pValue

Pointer of variable to retrieve GPIO pins state.

Return Value:

Returns TRUE on success, FALSE on failure.

```
BOOL WINAPI Gpio_SetBitValue( int nBit, BYTE bValue );
```

Remarks:

Call this function to Set GPIO pins value.

Parameters:

nBit

Stand for GPIO pins, value GPO_BIT_0 ~ GPO_BIT_7.

bValue

Output value of GPIO pins, value 1 or 0.

Return Value:

Returns TRUE on success, FALSE on failure.

Linux:**1. Syntax:**


```
i32 getInChLevel( i32 channel, u8 *val )
```

Description: Get the value of GPIO Input and put the value at *val.

Parameters:

- I. The parameter 'channel' indicates the GPIO Input pins to show. Users can use the macros GPI0, GPI1, GPI2, GPI3 to indicate the GPIO Input channel.

For example:

```
getInChLevel( GPI2, &val); // Indicate the GPIO Input channel 2
getInChLevel( GPI0 | GPI3, &val); // Indicate the GPIO Input
                                   // channel 0 and channel 3
```

- II. The parameter 'val' is an unsigned character pointer. The function puts the values of the indicated GPIO channels at the memory pointed by 'val'. The bit 0 of *val shows the value of GPIO Input channel 0. The bit 1 of *val shows the value of GPIO Input channel 1. Other bits show the corresponding GPIO Input channels. Because there are only four channels, bit 4 ~ bit 7 of *val are always zero.

Here is an example:

If GPIO Input channel 1 and channel 3 are both 1.

```
unsigned char ch;
getInChLevel( GPI1|GPI3, &ch );
```

The returned value of variable 'ch' is 0xa.

Return Value: If the function gets the values successfully, it returns 0. If any error, it returns -1.

2. Syntax:

```
i32 setOutChLevel( i32 channel, u8 val )
```

Description: Set the value of GPIO Output according to the variable 'val'.

Parameters:

- I. The parameter 'channel' indicates the GPIO Output pins to set. Users can use the macros GPO0, GPO1, GPO2, GPO3 to indicate the GPIO Output

channels.

- II. The parameter 'val' indicate the value to be set to GPIO Output channel. The acceptable values is limited to 0 and 1.

For example:

```
/* Setting the GPIO Output channel 2 to 1 */
setOutChLevel( GPO2, 1 );

/* Setting the GPIO Output channel 0 and channel 3 to 0 */
getInChLevel( GPO0 | GPO3, 0 );
```

Return Value: If the function sets the values successfully, it returns 0. If any error, it returns -1.

3. Syntax:

```
i32 getOutchLevel( i32 channel, u8 *val )
```

Description: Get the value of GPIO Output and put the value at *val.

Parameters:

- I. The parameter 'channel' indicates the GPIO Output pins to show. Users can use the macros GPO0, GPO1, GPO2, GPO3 to indicate the GPIO Output channel. For example:


```
getOutChLevel( GPO2, &val); // Indicate the GPIO Output channel 2

/* Indicate the GPIO Output channel 0 and channel 3. */
getOutChLevel( GPO0 | GPO3, &val);
```
- II. The parameter 'val' is an unsigned character pointer. The function puts the values of the indicated GPIO channels at the memory pointed by 'val'. The bit 0 of *val shows the value of GPIO Output channel 0. The bit 1 of *val shows the value of GPIO Output channel 1. Other bits show the corresponding GPIO Output channels. Because there are only four channels, bit 4 ~ bit 7 of *val are always zero.

Here is an example:

If GPIO Output channel 0 and channel 2 are both 1.

```
unsigned char ch;
getOutChLevel( GPO0|GPO2, &ch );
```

The returned value of variable ‘ch’ is 0x5.

Return Value: If the function gets the values successfully, it returns 0. If any error, it returns -1.

3.2.2. Watchdog

Windows:

Notice: Before using GPIO APIs, be sure Super I/O driver was installed successfully.

```
typedef enum _E_TIME_UNIT {

    TIME_UNIT_SECOND,
    TIME_UNIT_MINUTE,

} E_TIMER_UNIT;
```

```
BOOL WINAPI SuperIo_Open();
```

Remarks:

You must call this function at start-up of application to open Super I/O device.

Parameters:

(None)

Return Value:

Returns TRUE on success.

If the Super I/O driver has not installed, return FALSE.

```
void WINAPI SuperIo_Close();
```

Remarks:

You must call this function at end of application to close Super I/O device.

Parameters:

(None)

Return Value:

(None)

BOOL WINAPI WatchDog_GetTimerCount(int* pCount);

Remarks:

Call this function to get value of current WatchDog Timer.

Parameters:

pCount

Pointer of variable to retrieve WatchDog Timer value.

Return Value:

Returns TRUE on success, FALSE on failure.

BOOL WINAPI WatchDog_GetTimerUnit(int* pTimerUnit);

Remarks:

Call this function to get unit of current WatchDog Timer, minute or second.

Parameters:

pTimerUnit

Pointer of variable to retrieve WatchDog Timer unit.

Return Value:

Returns TRUE on success, FALSE on failure.

BOOL WINAPI WatchDog_StartTimer (int nTimerCount, int nTimerUnit);

Remarks:

Call this function to start WatchDog Timer.

Parameters:

nCount

Timer value of WatchDog.

Value 2 ~ 255 if timer unit is second.

Value 1 ~ 255 if timer unit is minute.

nTimerUnit

TIME_UNIT_SECOND and TIME_UNIT_MINUTE stand for second and minute.

Return Value:

Returns TRUE on success, FALSE on failure.

BOOL WINAPI WatchDog_StopTimer ();

Remarks:

Call this function to stop WatchDog Timer.

Parameters:

(None)

Return Value:

Returns TRUE on success, FALSE on failure.

Linux:

1. Syntax:

u8 getWtdTimer(void)

Description: This function read the value of the watchdog time counter and return it to the caller.

Parameters: None.

Return Value: This function return the value of the time counter and return it to the caller as an unsigned integer.

2. Syntax:

void setWtdTimer(u8 val)

Description: This function sets the watchdog timer register to the value 'val' and starts to count down. The value could be 0 ~ 255. The unit is second. Setting the timer register to 0 disables the watchdog function and stops the countdown.

Parameters: The parameter 'val' is the value to set to watchdog timer register. The range is 0 ~ 255.

Return Value: None.

3.3. Power Subsystem

Windows:

```
#define POWER_ON_IGNITION      0xA5
#define POWER_ON_REMOTE        0x5A
#define IGNITION_ON            0xA5
#define IGNITION_OFF           0x5A
typedef enum _E_TIME_UNIT {

    TIME_UNIT_SECOND,
    TIME_UNIT_MINUTE,

} E_TIMER_UNIT;

typedef struct _T_PIC_INFO {

    BYTE PicType[ 3 ];
    BYTE PicModel[ 4 ];
    BYTE PicMajorVersion;
    BYTE PicMinorVersion;

} T_PIC_INFO;

BOOL WINAPI PowerPic_Open();
```

Remarks:

Call this function to open COM port which is connected to Power/PIC chip.

Parameters:

(None)

Return Value:

Returns TRUE on success, FALSE on failure.

```
void WINAPI PowerPic_Close();
```

Remarks:

Call this function to close COM port which is connected to Power/PIC chip.

Parameters:

(None)

Return Value:

(None)

BOOL WINAPI PowerPic_SetDefaultValue();

Remarks:

Call this function to set default value for Power/PIC.

Parameters:

(None)

Return Value:

Returns TRUE on success, FALSE on failure.

BOOL WINAPI PowerPic_GetPicMode(BYTE* pMode);

Remarks:

Call this function to get current PIC mode, the value is 0 ~ 15.

Parameters:

pMode

Pointer of variable to retrieve current PIC mode.

Return Value:

BOOL WINAPI PowerPic_GetPowerOnMode(BYTE* pMode);

Remarks:

Call this function to get current Power-On mode, If success, the value is POWER_ON_IGNITION or POWER_ON_REMOTE.

Parameters:

pMode

current Power-On mode.

Return Value:

Returns TRUE on success, FALSE on failure.

```
BOOL WINAPI PowerPic_SetPowerOnMode( BYTE bMode );
```

Remarks:

Call this function to set Power-On mode.

Parameters:

bMode

Set value POWER_ON_IGNITION or POWER_ON_REMOTE.

Return Value:

Returns TRUE on success, FALSE on failure.

```
BOOL WINAPI PowerPic_GetHardOffDelayTime( int* pSeconds );
```

Remarks:

Call this function to get how many time to turn off hardware power.

Parameters:

pSeconds

Pointer of variable to retrieve delay time in second unit.

Return Value:

Returns TRUE on success, FALSE on failure.

```
BOOL WINAPI PowerPic_GetSoftOffDelayTime( int* pSeconds );
```

Remarks:

Call this function to get how many time to shutdown Windows system.

Parameters:

pSeconds

Pointer of variable to retrieve delay time in second unit.

Return Value:

Returns TRUE on success, FALSE on failure.

```
BOOL WINAPI PowerPic_SetSoftOffDelayTime( int nTime, int
```


nTimeUnit);

Remarks:

Call this function to set how many time to shutdown Windows.

Parameters:

nTime

Time to shutdown Windows, the real value is depend on nTimeUnit.

nTimeUnit

TIME_UNIT_SECOND or TIME_UNIT_MINUTE.

If nTimeUnit is TIME_UNIT_MINUTE, the real delay time is nTime * 60.

Return Value:

Returns TRUE on success, FALSE on failure.

BOOL WINAPI PowerPic_SetHardOffDelayTime(int nTime, int nTimeUnit);

Remarks:

Call this function to set how many time to turn off hardware power.

Parameters:

nTime

Time to turn off hardware power, the real value is depend on nTimeUnit.

nTimeUnit

TIME_UNIT_SECOND or TIME_UNIT_MINUTE.

If nTimeUnit is TIME_UNIT_MINUTE, the real delay time is nTime * 60.

Return Value:

Returns TRUE on success, FALSE on failure.

BOOL WINAPI PowerPic_GetIgnitionStatus(BYTE* pStatus);

Remarks:

Call this function to get status of ignition.

Parameters:

pStatus

Pointer of variable to retrieve status of ignition.

If success, the value will be IGNITION_ON or IGNITION_OFF.

Return Value:

Returns TRUE on success, FALSE on failure.

BOOL WINAPI PowerPic_GetBatteryVoltage(float *pVoltage);

Remarks:

Call this function to get the value of Power-Supply.

Parameters:

Pointer of variable to retrieve voltage value, in float type.

Return Value:

Returns TRUE on success, FALSE on failure.

BOOL WINAPI PowerPic_GetFirmwareVersion(T_PIC_INFO *pPicInfo);

Remarks:

Call this function to get the version of PIC firmware.

Parameters:

pPicInfo

Pointer of structure T_PIC_INFO to get information about PIC firmware.

Return Value:

Returns TRUE on success, FALSE on failure.

If success, the member variable PicMajorVersion & PicMinorVersion of T_PIC_INFO are ASCII format. For example, if the both value are 0x49 & 0x50, It means the PIC firmware version is "1.2".

Linux:**1. Syntax:**

i32 getIgnStatus(u8 *ignStatus)

Description: Get the current ignition status. The ignition has two statuses: ON or OFF.

Parameters: This function puts the ignition status at the memory pointed by the unsigned character pointer 'ignStatus'. If the returned status is 0xa5, the ignition is ON. If the returned status is 0x5a, the ignition is OFF. There are macros of Ignition ON and Ignition OFF in pwr_acce.h.

Return Value: If the function gets the ignition status and put it at the memory pointed by the argument successfully, this function will return 0. If any error, the function returns -1.

2. Syntax:

```
i32 setSoftOffDelayS( u32 setTime )
```

Description: The Soft Off Delay is the interval between that the system receives a power off signal and that the system generates a power off signal. This function sets up the interval in seconds.

Parameters: The parameter is of the type of unsigned long. The value of the parameter ranges from 0~255. The unit of the value of the parameter is seconds.

Return Value: If the function sets the delay time successfully, it will return 0. If any error, the function returns -1.

3. Syntax:

```
i32 setSoftOffDelayM( u32 setTime )
```

Description: The Soft Off Delay is the interval between that the system receives a power off signal and that the system generates a power off signal. This function sets up the interval in minutes.

Parameters: The parameter is of the type of unsigned long. The value of the parameter ranges from 0~255. The unit of the value of the parameter is minutes.

Return Value: If the function sets the delay time successfully, it will return 0. If any error, the function returns -1.

4. Syntax:

```
i32 setHardOffDelayS( u32 setTime )
```

Description: The Hard Off Delay is the interval between that the system is off and that the power 5VSB is off. This functions set up the interval in seconds.

Parameters: The parameter is of the type of unsigned long. The value of the parameter ranges from 0~255. The unit of the value of the parameter is seconds.

.

Return Value: If the function sets the delay time successfully, it will return 0. If any error, the function returns -1.

5. Syntax:

```
i32 setHardOffDelayM( u32 setTime )
```

Description: The Hard Off Delay is the interval between that the system is off and that the power 5VSB is off. This functions set up the interval in minutes.

Parameters: The parameter is of the type of unsigned long. The value of the parameter ranges from 0~255. The unit of the value of the parameter is minutes.

.

Return Value: If the function sets the delay time successfully, it will return 0. If any error, the function returns -1.

6. Syntax:

```
i32 setPowerOnMode( u8 powerOnMode )
```

Description: The function sets up the source of the boot-up signal of the system. There are two choices: boot from the Ignition or boot from the Remote Switch.

Parameters:

PowerOnMode = 0xa5, boot up by the Ignition.

PowerOnMode = 0x5a, boot up by the Remote Switch.

There are macros of Ignition mode and Remote Switch mode in pwr_acce.h (Linux) and AR-B6002.h(Windows).

Return Value: If the function sets power-on mode successfully, it will return 0. If any error, the function returns -1.

7. Syntax:

```
i32 getSoftOffDelay( u32 *Time )
```

Description: The Soft Off Delay is the interval between that the system receives a power off signal and that the system generates a power off signal. This function gets the interval.

Parameters: The parameter is a pointer which points to an unsigned long variable. The returned value is stored at this variable. The unit of the returned value is in seconds.

Return Value: If the delay time is returned successfully, the function returns 0. If any error, it returns -1.

8. Syntax:

```
i32 getHardOffDelay( u32 *Time )
```

Description: The Hard Off Delay is the interval between that the system is off and that the power 5VSB is off. This function gets the interval.

Parameters: The parameter is a pointer which points to an unsigned long variable. The returned value is stored at this variable. The unit of the returned value is in seconds.

Return Value: If the delay time is returned successfully, the function returns 0. If any error, it returns -1.

9. Syntax:

```
i32 getPowerOnMode( u8 *powerOnMode )
```

Description: The function gets the setting of power-on mode. There are two modes: boot from the Ignition or boot from the Remote Switch.

Parameters: The parameter is a pointer which points to an unsigned character. The returned code is stored at this memory. There are two power-on modes:

PowerOnMode = 0xa5, boot up by the Ignition.

PowerOnMode = 0x5a, boot up by the Remote Switch.

Return Value: If the power-on mode is returned successfully, the function returns 0. If any error, it returns -1

10. Syntax:

```
i32 getBattVolt( float *volt )
```

Description: The function gets the voltage reading of the battery.

Parameters: The parameter 'volt' is a pointer which points to an variable of type 'float'. The unit of the returned value is voltage.

Return Value: If the reading of voltage is returned successfully, the function returns 0. If any error, it returns -1

11. Syntax:

```
i32 getPicFwVer( struct PicInfo *ver )
```

Description: The function gets version information of Power Subsystem firmware.

Parameters: The parameter is a pointer which points to a 'PicInfo' structure, which consists of 9 unsigned characters. Here is the definition of structure 'PicInfo':

```
type struct {
    u8 type[3];    // The type of the power subsystem
    u8 mode[4];    // The mode at which the power subsystem is
                  // operating.
    u8 majorVersion; // Major version number of the firmware
    u8 minorVersion; // Minor version number of the firmware
} PicInfo;
```

```
PicInfo picInfo;
```

```
getPicFwVer( &picInfo );
printf(“%c.%c\n”, picInfo.majorVersion, picInfo.minorVersion );
```

Return Value: If the version information is returned successfully, the function returns 0. If any error, it returns -1.

12. Syntax:

```
i32 getPicMode( u8 *mode )
```

Description: The function gets the mode number at which the Power Subsystem is operating..

Parameters: The parameter is a pointer which points to a variable of type 'unsigned char'. The returned mode number is put at the memory which is pointed by parameter 'mode'.

Return Value: If the mode information is returned successfully, the function returns 0. If any error, it returns -1

13. Syntax:

```
i32 setPicDefault( void )
```

Description: The function restores the SoftOffDelay and HardOffDelay to the default value.

Parameters: None.

Return Value: If this function works successfully, the function will return 0. If any error, it will return -1.

Appendix A

Users have to modify the boot loader configuration to support COM6. Take the grub configuration file as an example. Add '8250.nr_uares=XX noirqdebug' at the setting of kernel. Here, XX represents the number of COM ports the system will support. Because the power subsystem connects to main system via COM6, the XX must be greater or equal to 6.

1. Modify the grub.conf.

```
[root@linux ~]# vi /boot/grub/grub.conf
default=0
timeout=5
splashimage=(hd0,0)/grub/splash.xpm.gz
hiddenmenu
```

```

title Fedora Core (2.6.27.5.117.FC10)
root (hd0,0)
kernel /vmlinuz-2.6.27.5.117.FC10 ro root=/dev/hda2 rhgb quiet
8250.nr_uarts=6 noirqdebug
initrd /initrd-2.6.27.5.117.FC10.img

```

2. List the status of the COM ports in the system.

```

# setserial -g /dev/ttyS*
/dev/ttyS0, UART: 16550A, Port: 0x03f8, IRQ: 4
/dev/ttyS1, UART: 16550A, Port: 0x02f8, IRQ: 3
/dev/ttyS2, UART: 16550A, Port: 0x03e8, IRQ: 11
/dev/ttyS3, UART: 16550A, Port: 0x02e8, IRQ: 10
/dev/ttyS4, UART: 16550A, Port: 0x04f8, IRQ: 11
/dev/ttyS5, UART: 16550A, Port: 0x04e8, IRQ: 10

```

The node '/dev/ttyS5' corresponds to COM6. The IO port is 0x4e8, IRQ 10.