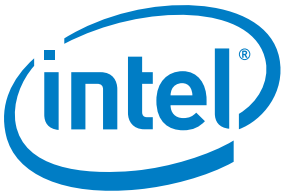

BSP for Windows* Embedded Compact 7 and 2013 for Intel® Atom™ Processor E3800 Product Family / Intel® Celeron® Processor N2807/N2930/J1900

Software Developer Guide

March 2015

Revision 1.0

Software Release version: Maintenance Release 2

Document Number: 332153-001US

Legal Disclaimer

By using this document, in addition to any agreements you have with Intel, you accept the terms set forth below.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked “reserved” or “undefined.” Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

Any software source code reprinted in this document is furnished for informational purposes only and may only be used or copied and no license, express or implied, by estoppel or otherwise, to any of the reprinted source code is granted by this document.

This document contains information on products in the design phase of development.

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. Go to: http://www.intel.com/products/processor_number/

Code Names are only for use by Intel to identify products, platforms, programs, services, etc. (“products”) in development by Intel that have not been made commercially available to the public, i.e., announced, launched or shipped. They are never to be used as “commercial” names for products. Also, they are not intended to function as trademarks.

Intel, Intel Atom, Intel Celeron, and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

Contents

Tables

Revision History

Date	Revision	Description
January 2015	1.0	Initial Release (Maintenance Release 2)

§

1 Introduction

1.1 Scope of Document

This document provides information for software developers on control codes, header files for GPIO drivers on the Windows* Embedded Compact 7 and 2013.

1.2 System Requirements

The following are required to build Intel WEC7 IO BSP on Walnut Canyon platform.

1. For WEC7*: Install Microsoft Windows* Embedded Compact 7 Platform Builder with March 2013 QFE Update. This creates a WINCE700 base directory on the default hard drive (for example, the following path should exist on C: drive: “C:\WINCE700”. If a WINCE700 base directory does not exist, the installation will fail.
2. For WEC2013*: Install Microsoft Windows* Embedded Compact 2013 Platform Builder with August 2014 QFE Update (8.0.6211). This creates a WINCE800 base directory on the default hard drive (for example, the following path should exist on C: drive: “C:\WINCE800”. If a WINCE800 base directory does not exist, the installation will fail.
3. Intel® Atom™ processor E3800 and D0 Intel® Atom™ processor.
4. Intel BIOS Intel® WEC IO Board Support Package (BSP) version: Intel® Processor WEC IO BSP.msi

1.3 Acronyms and Terminology

The acronyms and terms utilized in this document are listed in Table 1-1.

Table 1-1. Acronyms and Terminology

Term / Acronym	Definition
BSOD	Blue Screen of Death (Stop Error)
BSP	Board Support Package
GPIO	General Purpose Input/Output
IOCTL	IO Control
I/O	Input/Output
SUT	System Under Test
MSDN	Microsoft* Developer Network

§

2 Software Development and Configuration

2.1 Intel WEC7*/2013 IO Control Code

This section describes the control code and data structures that are exposed to the end user on DMA, SPI, I2C, GPIO and HS-UART drivers. HS-UART IOCTL will not be covered here, as the control code is fully adopted from Microsoft*.

Intel developed a set of IOCTL control code and data structures. End user or OEM code will call the Microsoft* WEC7*/2013 Framework API with the Intel developed IOCTL and data structure as parameters. This method is applicable to SPI, I2C, HS-UART, GPIO and DMA drivers.

For example, Intel developed test application will call the WEC7*/2013 Framework API function DeviceIoControl()and pass in Intel created IOCTL code and data structure as a parameter.

Note: Refer to the Microsoft Developer Network* (MSDN) Website for details on `DeviceIoControl()` API function:

http://msdn.microsoft.com/en-us/library/ms898288.aspx

Table 2-1. Code Example

```
result = (BOOL)DeviceIoControl(hI2CCtrl,
              (DWORD)IOCTL_I2C_EXECUTE_WRITE,
              (LPVOID)pTransBuf,
              sizeof(pTransBuf),
              NULL,
              NULL,
              &BytesHandled,
              NULL);
```

2.1.1 DMA Driver

The DMA data structure required by DMA to perform each transfer is shown in Table 2-2.

Table 2-2. DMA Driver

```
typedef struct _DMA_REQUEST_CONTEXT
{
    LPSSDMA_CHANNELS        ChannelTx;
    LPSSDMA_CHANNELS        ChannelRx;
    ULONG                   RequestLineTx;
    ULONG                   RequestLineRx;
    PHYSICAL_ADDRESS        SysMemPhysAddress;
    PUCHAR                  VirtualAdd;
    DWORD                   *pBytesReturned;
    ULONG                   PeripheralFIFOPhysicalAddress;
    DWORD                   TransactionSizeInByte;
    ULONG                   DummyDataWidthInByte;
    ULONG                   DummyDataForI2CSPI;
    DMA_TRANSACTION_TYPE    TransactionType;
    HANDLE                  hDmaCompleteEvent;
    DWORD                   IntervalTimeOutInMs;
    DWORD                   TotalTimeOutInMs;
}DMA_REQUEST_CONTEXT,*PDMA_REQUEST_CTXT;
```

Note: CTL_CODE() is an OAL macro which creates a unique system I/O control code (IOCTL). Please refer to MSDN website for detail: http://msdn.microsoft.com/en-us/library/ms904001.aspx.

Table 2-3. IOCTL used by SPI to Interface with DMA - RX

Description	This function allows the user to read the data from slave device using SPI, which interfaces with DMA. User may call DeviceIoControl() function API to pass in this IOCTL code and save output data from the read oparation to a pointer buffer.
Defined Macro	`IOCTL_LPSSDMA_REQUEST_DMA_SPI_RX \` `((ULONG)CTL_CODE( LPSSDMA_DEVICE_TYPE, 0x905, METHOD_BUFFERED, FILE_ANY_ACCESS))`
Return	None

Table 2-4. IOCTL used by SPI to Interface with DMA - TX

Description	This function allows the user to write the data to slave device using SPI which interface with DMA. User may call `DeviceIoControl()` function API to pass in this IOCTL code.
Defined Macro	`IOCTL_LPSSDMA_REQUEST_DMA_SPI_TX\` `((ULONG)CTL_CODE( LPSSDMA_DEVICE_TYPE, 0x906, METHOD_BUFFERED, FILE_ANY_ACCESS))`
Return	None

Table 2-5. IOCTL used by HS-UART to Interface with DMA - RX

Description	This function allows the user to read the data from slave device using HS UART which interface with DMA. User may call `DeviceIoControl()` function API to pass in this IOCTL code and save the output data from the read oparation to a pointer buffer.
Defined Macro	`IOCTL_LPSSDMA_REQUEST_DMA_UART_RX\` `((ULONG)CTL_CODE( LPSSDMA_DEVICE_TYPE, 0x907, METHOD_BUFFERED, FILE_ANY_ACCESS))`
Return	None

Table 2-6. IOCTL used by HS-UART to interface with DMA - TX

Description	This function allows the user to write the data to slave device using HS UART which interface with DMA.
Defined Macro	`IOCTL_LPSSDMA_REQUEST_DMA_UART_TX\` `((ULONG)CTL_CODE( LPSSDMA_DEVICE_TYPE, 0x908, METHOD_BUFFERED, FILE_ANY_ACCESS))`
Return	None

2.1.2 SPI

Table 2-7. SPI Data Structure Required by SPI to Perform each Transfer

```
typedef struct _SPI_TRANS_BUFFER
{
    DWORD               BusSpeed;
    DWORD                           Mode;
    DWORD               Direction;
    DWORD               Length;
    UCHAR               BitsPerEntry;
    DWORD               RemainingItem;
    PUCHAR              pBuffer;

}SPI_TRANS_BUFFER, *PSPI_TRANS_BUFFER;
```

Table 2-8. IOCTL used by SPI - Write

Description	This function allows the user to write the data to slave device using SPI bus. DMA. User may call `DeviceIoControl()` function API to pass in this IOCTL code.
Defined Macro	`IOCTL_SPI_EXECUTE_WRITE \` `CTL_CODE(FILE_DEVICE_SPI_CONTROLLER, 0x703, METHOD_BUFFERED, FILE_ANY_ACCESS)`
Return	None

Table 2-9. IOCTL used by SPI - Read

Description	This function allows the user to write the data to slave device using SPI bus. DMA. User may call `DeviceIoControl()` function API to pass in this IOCTL code.
Defined Macro	`IOCTL_SPI_EXECUTE_WRITE \` `CTL_CODE(FILE_DEVICE_SPI_CONTROLLER, 0x703, METHOD_BUFFERED, FILE_ANY_ACCESS)`
Return	None

2.1.3 I²C

Table 2-10. I2C Buffer Structure Required by I2C to Perform each Transfer

```
typedef struct _I2C_TRANS_BUFFER
{
    DWORD               AddressMode;
    DWORD               Address;
    DWORD               BusSpeed;
    DWORD               Direction;
    DWORD               DataLength;
    DWORD               RemainingItem;
    PUCHAR              Data;

}I2C_TRANS_BUFFER, *PI2C_TRANS_BUFFER;
```

Table 2-11. IOCTL used by I²C - Write

Description	This function allows the user to write the data to slave device using I2C bus. User may call `DeviceIoControl()` function API to pass in this IOCTL code.
Defined Macro	`IOCTL_I2C_EXECUTE_WRITE \` `CTL_CODE(FILE_DEVICE_I2C_CONTROLLER, 0x703, METHOD_BUFFERED, FILE_ANY_ACCESS)`
Return	None

Table 2-12. IOCTL used by I²C -Read

Description	This function allows the user to write the data to slave device using I2C bus. User may call `DeviceIoControl()` function API to pass in this IOCTL code.
Defined Macro	`IOCTL_I2C_EXECUTE_WRITE \` `CTL_CODE(FILE_DEVICE_I2C_CONTROLLER, 0x703, METHOD_BUFFERED, FILE_ANY_ACCESS)`
Return	None

2.1.4 GPIO

Table 2-13. Required GPIO Structure

```
typedef struct
{
    ULONG pin;
     union
    {
        ULONG data;
        GPIO_CONNECT_IO_PINS_MODE ConnectMode;
    } u;

} GPIO_PIN_PARAMETERS, *PGPIO_PIN_PARAMETERS;
```

2.1.4.1 IOCTL used by GPIO

Table 2-14. Read an Input Pin

Description	This function allows the user to read an input pin
Defined Macro	IOCTL_GPIO_READ \ CTL_CODE(FILE_DEVICE_UNKNOWN, 0x900, METHOD_BUFFERED, FILE_ANY_ACCESS)
Return	None

Table 2-15. Write to an Output Pin

Description	This function allows the user to write high or low an output selected pin
Defined Macro	IOCTL_GPIO_WRITE \ CTL_CODE(FILE_DEVICE_UNKNOWN, 0x901, METHOD_BUFFERED , FILE_ANY_ACCESS)
Return	None

Table 2-16. Set a Pin to be Input or Output

Description	This function allows the user to set as input or output pin
Defined Macro	IOCTL_GPIO_DIRECTION \ CTL_CODE(FILE_DEVICE_UNKNOWN, 0x902, METHOD_BUFFERED, FILE_ANY_ACCESS)
Return	None

Table 2-17. Configure Multiplexing for Selected GPIO Pins

Description	This function allows the user to configure multiplexing for selected GPIO pins.
Defined Macro	IOCTL_GPIO_MUX \ CTL_CODE(FILE_DEVICE_UNKNOWN, 0x903, METHOD_BUFFERED, FILE_ANY_ACCESS)
Return	None

Table 2-18. Query Current Pin's Setting

Description	This function allows the user to query current pin's setting
Defined Macro	IOCTL_GPIO_QUERY \ CTL_CODE(FILE_DEVICE_UNKNOWN, 0x904, METHOD_BUFFERED, FILE_ANY_ACCESS)
Return	None

2.2 Public Header Files

2.2.1 DMAPublic.h

Table 2-19. DMAPublic.h

```
#ifndef DMAPUBLIC_H
#define DMAPUBLIC_H

//*****************************************************************************
********
// IOCTL code definition
//*****************************************************************************
********

#define DMA1_DEVICE_NAME          TEXT("DMA1:")
#define DMA2_DEVICE_NAME          TEXT("DMA2:")

#define LPSSDMA_DEVICE_TYPE 0x8003

#define IOCTL_LPSSDMA_REQUEST_DMA_SPI_RX \
    ((ULONG)CTL_CODE( LPSSDMA_DEVICE_TYPE, 0x905, METHOD_BUFFERED,
FILE_ANY_ACCESS))

#define IOCTL_LPSSDMA_REQUEST_DMA_SPI_TX\
    ((ULONG)CTL_CODE( LPSSDMA_DEVICE_TYPE, 0x906, METHOD_BUFFERED,
FILE_ANY_ACCESS))

#define IOCTL_LPSSDMA_REQUEST_DMA_UART_RX\
    ((ULONG)CTL_CODE( LPSSDMA_DEVICE_TYPE, 0x907, METHOD_BUFFERED,
FILE_ANY_ACCESS))

#define IOCTL_LPSSDMA_REQUEST_DMA_UART_TX\
    ((ULONG)CTL_CODE( LPSSDMA_DEVICE_TYPE, 0x908, METHOD_BUFFERED,
FILE_ANY_ACCESS))

typedef enum _LPSSDMA_CHANNELS
{
    LPSSDMA_CHANNEL0 = 0,
    LPSSDMA_CHANNEL1,
    LPSSDMA_CHANNEL2,
    LPSSDMA_CHANNEL3,
    LPSSDMA_CHANNEL4,
    LPSSDMA_CHANNEL5,
    LPSSDMA_CHANNEL6,
    LPSSDMA_CHANNEL7,
    LPSSDMA_CHANNEL_INVALID,
}LPSSDMA_CHANNELS;

typedef enum _DMA_TRANSACTION_TYPE{
    TRANSACTION_TYPE_SINGLE_RX = 0,
    TRANSACTION_TYPE_SINGLE_TX,
    TRANSACTION_TYPE_SINGLE_TXRX
```

```
}DMA_TRANSACTION_TYPE;

typedef struct _DMA_REQUEST_CONTEXT
{
    LPSSDMA_CHANNELS        ChannelTx;
    LPSSDMA_CHANNELS        ChannelRx;
    ULONG                   RequestLineTx;
    ULONG                   RequestLineRx;
    PHYSICAL_ADDRESS        SysMemPhysAddress;
    PUCHAR                  VirtualAdd;
    DWORD                   *pBytesReturned;
    ULONG                   PeripheralFIFOPhysicalAddress;
    DWORD                   TransactionSizeInByte;
    ULONG                   DummyDataWidthInByte;
    ULONG                   DummyDataForI2CSPI;
    DMA_TRANSACTION_TYPE    TransactionType;
    HANDLE                  hDmaCompleteEvent;
    DWORD                   IntervalTimeOutInMs;
    DWORD                   TotalTimeOutInMs;
}DMA_REQUEST_CONTEXT,*PDMA_REQUEST_CTXT;

#define DEFAULT_TOTALTIMEOUT_IN_MS  (1000*60) /* 60 seconds*/
/*
Explain to IntervalTimeOutInMs and TotalTimeOutInMs
1. When IntervalTimeOutInMs is zero
That means don't use Interval TimeOut, TotalTimeOutInMs should not be
zero or MAXDWORD.
if dma receive TotalTimeOutInMs as zero or MAXDWORD, it will set a
default value to Total TimeOut
2. When IntervalTimeOutInMs is MAXWORD
That means dma should return immediately after it gets one bytes, so
TransactionSizeInByte should be 1.
if dam receive TotalTimeOutInMs is MAXDWORD, it will wait until get one
byte.
if dma receive TotalTimeOutInMs is zero,it will set a default value to
Total TimeOut
3. When IntervalTimeOutInMs is not zero and MAXWORD.
TotalTimeOutInMs can be MAXDWORD, that means no total timeout, the dma
will finished when Interval TimeOut or all bytes transfered
TotalTimeOutInMs can't be zero, a default value will be set to it.
*/

#endif
```

2.2.2 SPIPublic.h

Table 2-20. SPIPublic.h

```
#ifndef __SPIPUBLIC_H__
#define __SPIPUBLIC_H__

// Windows Header Files:
#include <windows.h>
#include <winioctl.h>
// C RunTime Header Files
#include <stdlib.h>

//*****************************************************************
************
// IOCTL code definition
//*****************************************************************
************
#define FILE_DEVICE_SPI_CONTROLLER FILE_DEVICE_CONTROLLER

#define IOCTL_SPI_EXECUTE_WRITE     \
        CTL_CODE(FILE_DEVICE_SPI_CONTROLLER, 0x703,
METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_SPI_EXECUTE_READ      \
        CTL_CODE(FILE_DEVICE_SPI_CONTROLLER, 0x704,
METHOD_BUFFERED, FILE_ANY_ACCESS)

/*
* Transfer direction read/write
*/
typedef enum SPI_TRANSFER_DIRECTION
{
    TransferDirectionRead = 1,
    TransferDirectionWrite = 2

}SPI_TRANSFER_DIRECTION;

/*
*  SPI Mode Definitions
*  SPI_MODE | Clock Polarity | Clock Phase
*  0            0              0
*  1            0              1
*  2            1              0
*  3            1              1
*/

typedef enum _SPI_BUS_MODE
{
```

```
    IO_SPI_MODE_0 = 0,
    IO_SPI_MODE_1 = 1,
    IO_SPI_MODE_2 = 2,
    IO_SPI_MODE_3 = 3,
    IO_SPI_MODE_MAX
}SPI_BUS_MODE;

/*
* SPI transfer speed setting
*/
typedef enum _SPI_BUS_SPEED
{
    SPI_BUS_SPEED_125KHZ    = 1,
    SPI_BUS_SPEED_10MHZ    = 2
}SPI_BUS_SPEED;


/*SPI Buffer Structure*/
typedef struct _SPI_TRANS_BUFFER
{
    DWORD               BusSpeed;
    DWORD                                       Mode;
    DWORD               Direction;
    DWORD               Length;
    UCHAR               BitsPerEntry;
    DWORD               RemainingItem;
    PUCHAR              pBuffer;

}SPI_TRANS_BUFFER, *PSPI_TRANS_BUFFER;

#endif
```

2.2.3 I²CPublic.h

Table 2-21. I²CPublic.h

```
// Windows Header Files:
#include <windows.h>
#include <winioctl.h>
// C RunTime Header Files
#include <stdlib.h>


//*****************************************************************
************
// IOCTL code definition
//*****************************************************************
************
#define FILE_DEVICE_I2C_CONTROLLER FILE_DEVICE_CONTROLLER
```

```
#define IOCTL_I2C_EXECUTE_WRITE     \
        CTL_CODE(FILE_DEVICE_I2C_CONTROLLER, 0x703,
METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_I2C_EXECUTE_READ      \
        CTL_CODE(FILE_DEVICE_I2C_CONTROLLER, 0x704,
METHOD_BUFFERED, FILE_ANY_ACCESS)

/*
* Transfer direction read/write
*/
typedef enum I2C_TRANSFER_DIRECTION
{
    TransferDirectionRead = 1,
    TransferDirectionWrite = 2

}I2C_TRANSFER_DIRECTION;

/*
* I2C transfer speed setting
*/
typedef enum _I2C_BUS_SPEED
{
    I2C_BUS_SPEED_100KHZ    = 1,
    I2C_BUS_SPEED_400KHZ    = 2
}I2C_BUS_SPEED;

/*
* I2C address mode setting
*/
typedef enum _I2C_ADDRESS_MODE
{
    AddressMode7Bit     = 1,
    AddressMode10Bit    = 2
} I2C_ADDRESS_MODE;


/*I2C Buffer Structure*/
typedef struct _I2C_TRANS_BUFFER
{
    DWORD               AddressMode;
    DWORD               Address;
    DWORD               BusSpeed;
    DWORD               Direction;
    DWORD               DataLength;
    DWORD               RemainingItem;
    PUCHAR              Data;

}I2C_TRANS_BUFFER, *PI2C_TRANS_BUFFER;


#endif
```

2.2.4 GPIOPublic.h

Table 2-22. GPIOPublic.h

```
#ifndef GPIOPUBLIC_H
#define GPIOPUBLIC_H

// Windows Header Files:
#include <windows.h>
#include <winioctl.h>
// C RunTime Header Files
#include <stdlib.h>

//*****************************************************************
************
// IOCTL code definition
//*****************************************************************
************
//
// The IOCTL function codes from 0x800 to 0xFFF are for customer
use.
//
#define IOCTL_GPIO_READ \
    CTL_CODE( FILE_DEVICE_UNKNOWN, 0x900, METHOD_BUFFERED,
FILE_ANY_ACCESS  )

#define IOCTL_GPIO_WRITE \
    CTL_CODE( FILE_DEVICE_UNKNOWN, 0x901, METHOD_BUFFERED ,
FILE_ANY_ACCESS  )

#define IOCTL_GPIO_DIRECTION \
    CTL_CODE( FILE_DEVICE_UNKNOWN, 0x902, METHOD_BUFFERED,
FILE_ANY_ACCESS  )

#define IOCTL_GPIO_MUX \
    CTL_CODE( FILE_DEVICE_UNKNOWN, 0x903, METHOD_BUFFERED,
FILE_ANY_ACCESS  )

#define IOCTL_GPIO_QUERY \
    CTL_CODE( FILE_DEVICE_UNKNOWN, 0x904, METHOD_BUFFERED,
FILE_ANY_ACCESS  )

typedef enum
{
    CONNECT_MODE_INVALID = 0,
    CONNECT_MODE_INPUT,
    CONNECT_MODE_OUTPUT,
    CONNECT_MODE_MAXIMUM  = CONNECT_MODE_OUTPUT
} GPIO_CONNECT_IO_PINS_MODE;

typedef struct
{
    ULONG pin;
```

```
     union
    {
        ULONG data;
        GPIO_CONNECT_IO_PINS_MODE ConnectMode;
    } u;

} GPIO_PIN_PARAMETERS, *PGPIO_PIN_PARAMETERS;


#endif /* GPIOPUBLIC_H */
```

2.2.4.1 IOCTL for GPIO

This section describes the IOCTL for GPIO functions.

Table 2-23. Read an Input Pin

Description	This function allows the user to read an input pin

Defined Macro	`IOCTL_GPIO_READ \`
	
	`CTL_CODE( FILE_DEVICE_UNKNOWN, 0x900, METHOD_BUFFERED, FILE_ANY_ACCESS )`

Return	None

Table 2-24. Write High or Low an Output Selected Pin

Description	This function allows the user to write high or low an output selected pin

Defined Macro	`IOCTL_GPIO_WRITE \`
	
	`CTL_CODE( FILE_DEVICE_UNKNOWN, 0x901, METHOD_BUFFERED , FILE_ANY_ACCESS )`

Return	None

Table 2-25. Set As Input or Output Pin

Description	This function allows the user to set as input or output pin

Defined Macro	`IOCTL_GPIO_DIRECTION \`
	
	`CTL_CODE( FILE_DEVICE_UNKNOWN, 0x902, METHOD_BUFFERED, FILE_ANY_ACCESS )`

Return	None

Table 2-26. Configure Multiplexing for Selected GPIO Pins

Description	This function allows the user to configure multiplexing for selected GPIO pins.

Defined Macro	`IOCTL_GPIO_MUX \`
	
	`CTL_CODE( FILE_DEVICE_UNKNOWN, 0x903, METHOD_BUFFERED, FILE_ANY_ACCESS )`

Return	None

Table 2-27. Query Settings of Current Pin

Description	This function allows the user to query current pin's setting

Defined Macro	`IOCTL_GPIO_QUERY \`
	
	`CTL_CODE( FILE_DEVICE_UNKNOWN, 0x904, METHOD_BUFFERED, FILE_ANY_ACCESS )`

Return	None

2.2.5 GPIOPublic.h

Table 2-28. GPIOPublic.h

```
#ifndef GPIOPUBLIC_H
#define GPIOPUBLIC_H

// Windows Header Files:
#include <windows.h>
#include <winioctl.h>
// C RunTime Header Files
#include <stdlib.h>

//*****************************************************************************
// IOCTL code definition
//*****************************************************************************
//
// The IOCTL function codes from 0x800 to 0xFFF are for customer use.
//
#define IOCTL_GPIO_READ \
    CTL_CODE( FILE_DEVICE_UNKNOWN, 0x900, METHOD_BUFFERED, FILE_ANY_ACCESS  )

#define IOCTL_GPIO_WRITE \
    CTL_CODE( FILE_DEVICE_UNKNOWN, 0x901, METHOD_BUFFERED , FILE_ANY_ACCESS  )

#define IOCTL_GPIO_DIRECTION \
    CTL_CODE( FILE_DEVICE_UNKNOWN, 0x902, METHOD_BUFFERED, FILE_ANY_ACCESS  )

#define IOCTL_GPIO_MUX \
    CTL_CODE( FILE_DEVICE_UNKNOWN, 0x903, METHOD_BUFFERED, FILE_ANY_ACCESS  )

#define IOCTL_GPIO_QUERY \
    CTL_CODE( FILE_DEVICE_UNKNOWN, 0x904, METHOD_BUFFERED, FILE_ANY_ACCESS  )

typedef enum
{
    CONNECT_MODE_INVALID = 0,
    CONNECT_MODE_INPUT,
    CONNECT_MODE_OUTPUT,
    CONNECT_MODE_MAXIMUM  = CONNECT_MODE_OUTPUT
} GPIO_CONNECT_IO_PINS_MODE;

typedef struct
{
    ULONG pin;
     union
    {
        ULONG data;
        GPIO_CONNECT_IO_PINS_MODE ConnectMode;
    } u;

} GPIO_PIN_PARAMETERS, *PGPIO_PIN_PARAMETERS;


#endif /* GPIOPUBLIC_H */
```

2.3 Sample Program

Below are sample programs which describe how to utilize the APIs presented in Section 1.

2.3.1 HS-UART with DMA driver

Table 2-29. HS-UART with DMA Driver

```
DWORD CPdd16550Isr::UartDmaTxThread()
{
    while(!IsTerminated())
    {
        DMA_REQUEST_CONTEXT DmaTxRequest;
        BOOL Rtn;
        DWORD BytesHandled;
        if (WaitForSingleObject(m_UartDmaTxEvent,INFINITE) ==
WAIT_OBJECT_0)
        {
            m_HardwareLock.Lock();

            /* We don't know the driver loading sequence in WEC7, so had
to check it before start dma transfer
            must do it after we get m_HardwareLock.Lock();
            */
            if (m_DMADeviceHandle == NULL)
            {
                m_DMADeviceHandle =  CreateFile(
                    DMA2_DEVICE_NAME,
                    GENERIC_READ|GENERIC_WRITE,
                    FILE_SHARE_READ|FILE_SHARE_WRITE,
                    NULL,
                    OPEN_EXISTING,
                    0,
                    NULL
                    );
                if (m_DMADeviceHandle == NULL
                    || m_DMADeviceHandle == INVALID_HANDLE_VALUE)
                {
                    m_DMADeviceHandle = NULL;
                    m_HardwareLock.Unlock();
                    continue;
                }

            }

            memset(&DmaTxRequest,0,sizeof(DmaTxRequest));
            DmaTxRequest.ChannelRx = (LPSSDMA_CHANNELS)m_DmaChannelRx;
            DmaTxRequest.ChannelTx = (LPSSDMA_CHANNELS)m_DmaChannelTx;

            DmaTxRequest.RequestLineRx = m_DmaRequestLineRx;
            DmaTxRequest.RequestLineTx = m_DmaRequestLineTx;

            DmaTxRequest.TransactionType = TRANSACTION_TYPE_SINGLE_TX;
```

```
            DmaTxRequest.hDmaCompleteEvent = m_TxDmaCompleteEvent;

            DmaTxRequest.PeripheralFIFOPhysicalAddress =
                (ULONG)(m_ioPhysicalBase.LowPart +
(TRANSMIT_HOLDING_REGISTER *(m_pIsrInfoVirt->uMultiplier)));

            DmaTxRequest.SysMemPhysAddress = m_TxDmaBufferPhyAddress;
            DmaTxRequest.SysMemPhysAddress.LowPart +=
m_TxDmaBufferReadIndex;

            DmaTxRequest.TransactionSizeInByte = m_TxDmaBufferWriteIndex
- m_TxDmaBufferReadIndex;

            DmaTxRequest.VirtualAdd = m_TxDmaBufferVirtualAddress;

            DWORD TotalTimeout;
            DWORD Timeout;
            TotalTimeout = m_CommTimeouts.WriteTotalTimeoutMultiplier *
DmaTxRequest.TransactionSizeInByte +
                m_CommTimeouts.WriteTotalTimeoutConstant;

            if ( TotalTimeout == 0)
            {
                /* Set a safty value, consider BAUD_300, 50ms for one
byte */
                TotalTimeout = 50 * DmaTxRequest.TransactionSizeInByte +
300;
            }
            else
            {
                DWORD DELTA;
                ASSERT(TotalTimeout!=0);
                DELTA = min(100,TotalTimeout/10);
                TotalTimeout = TotalTimeout - DELTA;
            }
            Timeout = TotalTimeout;
            DmaTxRequest.TotalTimeOutInMs = TotalTimeout;
            DmaTxRequest.IntervalTimeOutInMs = 0; /* No Interval Time
Out for TX*/

            *m_TxBytesReturned = (DWORD)-1;
            DmaTxRequest.pBytesReturned = m_TxBytesReturned;

            BytesHandled = 0;

            EnableXmitInterrupt(FALSE);

            m_HardwareLock.Unlock();
            ASSERT(DmaTxRequest.TransactionSizeInByte != 0);
            DEBUGMSG(ZONE_WRITE, (TEXT("UartDmaTxThread : Start Dma Tx
for length %d with TotalTimeout %d; Tx Read Index %d\r\n"),
DmaTxRequest.TransactionSizeInByte,
DmaTxRequest.TotalTimeOutInMs,m_TxDmaBufferReadIndex));
            Rtn = DeviceIoControl(m_DMADeviceHandle,
                (DWORD)IOCTL_LPSSDMA_REQUEST_DMA_UART_TX,
```

```
                &DmaTxRequest,
                sizeof(DmaTxRequest),
                NULL,
                0,
                &BytesHandled,
                NULL);

            DWORD WaitRlt;

            WaitRlt =
WaitForSingleObject(DmaTxRequest.hDmaCompleteEvent,
max(Timeout,(DWORD)(Timeout+1000)));
            DEBUGMSG(ZONE_WRITE, (TEXT("UartDmaTxThread : Dma Tx
finished with len %d and Wait Result %d\r\n"),
*m_TxBytesReturned,WaitRlt));

            ASSERT(*m_TxBytesReturned ==
DmaTxRequest.TransactionSizeInByte);

            m_HardwareLock.Lock();
            m_TxDmaBufferReadIndex += *m_TxBytesReturned;
            if (m_TxDmaBufferReadIndex == m_TxDmaBufferWriteIndex)
            {
                m_TxDmaBufferReadIndex =  m_TxDmaBufferWriteIndex = 0;
            }
            if (WaitRlt == WAIT_OBJECT_0)
            {
                EnableXmitInterrupt(TRUE);
            }
            else
            {
                ASSERT(0);
            }
            m_HardwareLock.Unlock();


        }
    }
    return 0;
}
```

2.3.2 SPI with DMA Driver

Table 2-30. SPI with DMA Driver

```
BOOL ProgramExecuteDmaTransaction(PDEVICE_CONTEXT pDevice)
{
    BOOL status  = FALSE;
   DWORD offset = 0;
   DWORD TotalTransferLength = 0;

   //DMA request struct
```

```
    DMA_REQUEST_CONTEXT DmaRequest = {0};

    //Fix DMA request Line for SPI here
     DmaRequest.RequestLineTx = 0;
     DmaRequest.RequestLineRx= 1;

    //Fix the channel 0000 - SPI Tx, 0001 - SPI Rx
    DmaRequest.ChannelTx = LPSSDMA_CHANNEL0;
     DmaRequest.ChannelRx = LPSSDMA_CHANNEL1;

    //Tx Source System Memory Address
    DmaRequest.SysMemPhysAddress = pDevice->MemPhysicalAddress;
    DmaRequest.VirtualAdd = pDevice->VirtualAddress;

    //Tx Destination Address
    DmaRequest.PeripheralFIFOPhysicalAddress = (ULONG)(pDevice-
>phyAddress.LowPart + offsetof(SPI_REGISTERS,SSDR));

     DmaRequest.DummyDataWidthInByte = 1;
     DmaRequest.DummyDataForI2CSPI = 0xFF;

    /* Event for completion DMA operation */
     DmaRequest.hDmaCompleteEvent = pDevice->hDmaCompletionEvent;

    DmaRequest.TransactionType = TRANSACTION_TYPE_SINGLE_TXRX;

    pDevice->DMABytesReturned = (DWORD*)LocalAlloc(LPTR,
sizeof(DWORD));
    if(!pDevice->DMABytesReturned)
    {
       return status;
    }

    *pDevice->DMABytesReturned = (DWORD)-1;
    DmaRequest.pBytesReturned = pDevice->DMABytesReturned;

     DmaRequest.IntervalTimeOutInMs = 0;
    DmaRequest.TotalTimeOutInMs = DEFAULT_TOTAL_TIMEOUT_IN_MS;

    /*If transfer length more than 4092, multi transfer is being
done here */
    while(pDevice->TotalDMATransRequested && offset < pDevice-
>TotalDMATransRequested)
    {
       DmaRequest.TransactionSizeInByte = DEFAULT_DMA_BUFFER_SIZE;
       TotalTransferLength += DmaRequest.TransactionSizeInByte;
       status =
StartExecuteDmaTransaction(pDevice,DmaRequest,offset);
       offset++;

    }
    //Last DMA transaction or transfer less or equal 4092
    if((pDevice->BufLength - TotalTransferLength <
```

```
DEFAULT_DMA_BUFFER_SIZE) && (pDevice->BufLength -
TotalTransferLength != 0))
   {
      DmaRequest.TransactionSizeInByte = pDevice->BufLength -
TotalTransferLength;
      status =
StartExecuteDmaTransaction(pDevice,DmaRequest,offset);


   }

    return status;
}

BOOL StartExecuteDmaTransaction(PDEVICE_CONTEXT
pDevice,DMA_REQUEST_CONTEXT DmaRequest,DWORD offset)
{
   ULONG IoctlCode = 0;
   BOOL  result  = FALSE;
   ULONG BytesHandled;
   DWORD index = 0;

   DWORD TransferLength = DmaRequest.TransactionSizeInByte;

   //Copy data to Virtual memory for DMA Transaction
   if(pDevice->Direction == TransferDirectionWrite){

      for(index = 0; index < TransferLength; index++)
      {
         *((pDevice->VirtualAddress) + index) = (UCHAR)
*((pDevice->pBuffer)+ index +(offset*DEFAULT_DMA_BUFFER_SIZE));

      }
   }
   if(pDevice->Direction == TransferDirectionRead){

      for(index = 0; index < TransferLength; index++)
      {
            *((pDevice->VirtualAddress) + index) = 0xFF;

      }
   }

    DmaPreExecute(pDevice);

   //set IOCTL base of read/write operation

    if (pDevice->Direction == TransferDirectionWrite)
    {
        SPI_TXRX_DMA_ENABLE(pDevice);
        IoctlCode = IOCTL_LPSSDMA_REQUEST_DMA_SPI_TX;
   }
    else if (pDevice->Direction == TransferDirectionRead)
```

```
    {
        SPI_TXRX_DMA_ENABLE(pDevice);
        IoctlCode = IOCTL_LPSSDMA_REQUEST_DMA_SPI_RX;
    }

   if(pDevice->hDmaHandler != NULL)
   {
      result = (BOOL)DeviceIoControl(pDevice->hDmaHandler,
                (DWORD)IoctlCode,
                (LPVOID)&DmaRequest,
                sizeof(DmaRequest),
                NULL,
                0,
                &BytesHandled,
                NULL);
   }

   if(WaitForSingleObject(DmaRequest.hDmaCompleteEvent,INFINITE)
== WAIT_OBJECT_0)
   {
      memcpy_s(pDevice->pBuffer +
(offset*DEFAULT_DMA_BUFFER_SIZE),TransferLength,pDevice-
>VirtualAddress,TransferLength);
      pDevice->Status   = TRANSFER_SUCCESS;
   }
   else
      pDevice->Status   = TRANSFER_UNSUCCESSFUL;

   return pDevice->Status;
}
```

§