CHAPTER FIVE

API

This chapter discusses the various Application Programming Interface Classes used for programming with the H.O.T. Works Development System.

CONTENTS

Application Programming Interface

C++ Interface

The following classes provided a C++ interface to the H.O.T. Works Board. The classes allow the user to interface to the PCI card from their own C++ code with a few simple function calls. The code uses the hotworks.vxd device driver to interface to the PCI board.

There are three classes making up the XC6200DS interface:

1. XC6200DS
2. PCIBOARD
3. PCICore.

XC6200DS is the top level class and provides a complete interface to the XC6200 Development System. This class contains all the 6200 chip access functions, including the reading/writing of the control registers and loading and accessing the user design. The XC6200 chip type is selected when creating an XC6200DS object e.g. XC6200DS Board (XCAddr::XC6216); creates an XC6200DS object with a XC6216 chip. Through the supporting XCAddr and MapRegister classes all the addressing of the chip is easily handled independently of chip type. A complete description of these supporting classes can be found in the XACT STEP series 6200 User Guide.

The PCIBoard is a subclass of the XC6200DS class. This class contains the board IOSpace register access functions.

The supporting low level class PCICore handles the actual reading and writing to the PCIBoard as a memory mapped device. This class contains all the necessary code for interfacing to the device driver.

Please note: All related `.h, .cpp` and `ral` files can be found in the following dierectory:

```
C:\Hotworks\Api
```

A description of each of the three classes follows.

XC6200DS - XC6200 Interface Class

Constructor and Destructor

XC6200DS();-initializes and sets the chip architecture to the default value of XC6216

XC6200DS(XCAddr::DeviceDie);- initializes sets up the selected chip architecture.

~XC6200DS();

checkBoard

int checkBoard(); - checks the XC6200DS has been correctly initialized through the Device Driver interface. This function should be called initially to verify the PCI interface, it returns zero if the board has been correctly initialized.

load cal file to 6200

int loadCalFile(const char *filename);- This function is used to load a CAL file into the 6200. The CAL file is responsible for configuring the circuit in the 6200 and is loaded by a series of writes to the device. The current format of the CAL file is very straightforward. Each line in the CAL file contains either a comment, which begins with a "#" character, or a pair of ASCII hex values. These values represent the CAL address and the data to be written to that address.

6200 register level routines

void setColumn(byte column, word data); writes data to a user-defined register in the 6200. This register must consist of a group of cells all residing in the same column. The selection of the rows is determined by the setting of the 6200 map register. This function assumes the Mask register and bus width are set appropriately before this function is called.

word getColumn(byte column);- reads data from a user-defined register in the 6200. This register must consist of a group of cells all residing in the same column. The selection of the rows is determined by the setting of the 6200 map register. This function assumes the Mask register and bus width are set appropriately before this function is called.

void setConfig(byte config);- sets the configuration register on the 6200.

byte getConfig(void);- returns the configuration setting from the 6200.

int setBusWidth(byte width);- sets the bus width. The function takes a value of 8, 16 or 32, rather than the encoded values used by the configuration register.

int getBusWidth(void);- gets the bus width and returns an integer value of the returned width (8, 16 or 32).

void setRowWildcard(byte row_wildcard); sets the row wildcard register on the 6200.

byte getRowWildcard(void);- reads the row wildcard register on the 6200.

void setColumnWildcard(byte col_wildcard); sets the column wildcard register on the 6200.

byte getColumnWildcard(void);- reads the column wildcard register on the 6200.

void setMask(word mask);- writes to the 6200 mask register. This register contains a 32 bit value which determines valid data bits on the external bus. Bits set to '1' mask off data.

word getMask(void);- reads the 6200 mask register. This register contains a 32 bit value which determines valid data bits on the external bus. Bits set to '1' mask off data.

bool setMap(byte offset,word data);- writes to the 6200 map register. This register is used to assist in reading data from columns of 6200 cells. The 6216 has 64 cells per column, each of the 64 bits is passed as two data words, offset 0 and offset 1, and are used to mask or select a row in the register column. When data is read or written, only those cells in the rows with the corresponding mask bit set to '0' are accessed.

word getMap(byte offset);- reads the 32 bits of the selected 6200 map register. This register contains a 32 bit value which determines valid data bits when reading columns of cells, bits set to '1' mask off data.

bool setMap(MapRegister MapReg);- writes the entire MapRegister object to the 6200 map register. A complete description of the supporting MapRegister class can be found in the XACT STEP series 6200 User Guide.

void getMap(MapRegister &MapReg);- reads the 6200 map registers to the MapReg object. A complete description of the supporting MapRegister class can be found in the XACT STEP series 6200 User Guide.

void setDeviceId(byte device_id);- writes to the 16 byte device identification register. The final byte is the device ID, and will change for different versions of the silicon. Setting these bytes correctly turns on all I/O. Supplying an incorrect device ID is a method of turning off all I/O via software control.

byte getDeviceId(void);- reads the currently set device ID of the silicon. If a value such as 0xff is returned, the device ID is no correctly set and the device IO is not turned on.

byte scanDeviceId(void);- determines the device ID of the silicon. This function may be used to enable the device and/or to tell the version of the silicon.

PCI Board - PCI Board Interface Class

Constructor and Destructor

PCIBoard(); - initializes the board.

~PCIBoard();

Board level routines

void reset(void);- resets the 6200 by writing a 1 to the Reset Register.

void clear(void);- clears the 6200 by writing a 2 to the Reset Register and thus asserting the GCLR.

clock chip functions

void clockOn(void);- writes to continuous clock mode location to put the clock on.

void clockOff(void);- writes to the single clock mode and puts the clock in step mode.

void clockStep(word steps = 1);- writes the number of clock steps required to the step register.

void g1clockOn(void);- writes to continuous clock mode location to put the clock on.

void g1clockOff(void);- writes to the single clock mode and puts the clock in step mode.

void g1clockStep(word steps = 1);- writes the number of clock steps required to the step register.

read con port

word getCon(void);- reads the 32-bit "CON" port of the 6200, thus allowing direct reading of data from the 6200

SRAM access modes

void setBankControl(word state); - sets the bank control register for access options to the on-board RAM. State = 0, PCI owns both banks. See *Data Path Control Interface Chapter 3*

void setBankControl(byte bank1, byte bank2); sets the bank control register for access options to the on-board RAM.

Bank1 = 1, the 6200 owns the Bank1 RAM.
Bank1 = 0, the PCI bus owns the Bank1 RAM.
Bank2 = 1, the 6200 owns the Bank2 RAM.
Bank2 = 0, the PCI bus owns the Bank2 RAM.

word getBankControl(void);- returns the contents of the Control Register (2Ch).

Event status

void setEventMask(word mask); sets the event mask register for interrupt processing.

word getEventStatus(void);- shows the event status register settings for interrupt and event detection.

void acknowledgeEvents(word mask); clear out selected bits in the event status register

clock frequency

void sendClockData(word bits, int count); This function is used to send clock data (either a command word or program data) . The single bit register allows the configuration of the clock generation chip. Each transferred bit is followed by a 1ms delay.

void sendProtocolWord(void); This function sends the protocol word used by the 'set_clock_freq' function. Each transferred bit is followed by a 1ms delay.

double setClockFreq(double ref_mhz, double desired_mhz); Sets output frequency of ICD2053B clock generator.

current management

void setAdc(byte reset_enable);- writes to the ADC threshold register with the current ADC threshold value, if reset_enable is true then the 6200 is reset when the threshold is exceeded.

byte getAdc(void);- returns current value set in the ADC threshold register

double adcToMa(byte ad_value); - This function converts the 8-bit value read from the analog to digital converter into milliamps.

void setAdcThreshold(byte adcThr); sets the ADC threshold value, but does not write to the register

byte getAdcThreshold(void);- returns the currently set threshold value

byte getResetEnable(void);- returns internal status on the reset enable flag. Returns the actual frequency set (MHz) or a negative value on failure to set the desired frequency.

void sleep(clock_t wait);- This function is used to delay each clock generator programming bit by approx. 1 clock (1MS).

PCI Core - Core Class

Constructor and Destructor

PCICore(); - Initialises the device driver and the memory map interface

~PCICore(); - Releases the device driver handles

Write and Read to 6200

void write6200(word addr, word data); writes to the 6200 chip on the board

addr - is the 6200 chip register address.

data - is the data to be written

word read6200(word addr); - reads the 6200 chip on the board.

addr - is the 6200 chip register address.

returns - the data from the address location.

Write and Read IO space

void writeIOSpace(unsigned short addr, word data); writes to the board IO Space registers.

addr - is the IO register address.

data - is the data to be written to the register.

word readIOSpace(unsigned short addr); reads from the IO space registers.

addr - is the IO register address.

returns - the data from the address location.

Write and Read to SRAM

void writeRAM(word addr, word *data, word count);

addr - is the SRAM address.

*data - is a pointer to the data to be written to SRAM.

count - is the number of 32 bits of data to be written.

void readRAM(word addr,word *data, word count);

addr - is the SRAM address.

*data - is a pointer to the data location for the retrieved data.

count - is the number of 32 bits of data to be read.

Debug routines These routines provide access to the low level Device Driver interface:

PCICONFIG_S * dumpPCIConfigHeader(); the PCI header from the board.

unsigned short getIOBase(void);- the address of the IO Space.

word getPhysicalBase(void);- the physical address of the card.

word getMembase(void);- the plug and play memory location allocation of the board.

char *getErrorMessage(byte error_code);- returns the error string associated with a returned error code.

Error codes

enum PCIError {

EPCI_SUCCESS=0, - Success

EPCI_NO_DEVICE_DRIVER, - Device Driver not installed

EPCI_DD_INIT_FAILED, - Device Driver failure

EPCI_BAD_READ, - Error in reading board

EPCI_NO_CAL_FILE, - XC6200 CAL file not found for loading

EPCI_BAD_CAL_FILE, - Error in the CAL file

EPCI_BAD_BUS_WIDTH, - Error in selected bus width

EPCI_LAST_ERROR - Error count

};

Error Message Strings

static char *PCIErrorMsg[];- list of error message strings.

XC6200DS.h File

```
/*-----------------------------------------------------------------------

  @(#)xc6200ds.h      1.3 6/27/97

  Copyright (C) 1993-1997 Xilinx Development Corporation.
  This software is the subject of a licence agreement
  with Xilinx Inc. Use outside the terms of that agreement is prohibited.

  Owner: ***

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

#ifndef  _XC6200DS_H_
#define  _XC6200DS_H_

#include "xcaddr.h"
#include "pciboard.h"


class XC6200DS:public PCIBoard {

public:
/*
** Constructors
*/
        XC6200DS();
        XC6200DS(XCAddr::DeviceDie);

/*
** Destructor
*/
        ~XC6200DS();

/*
** load cal file to 6200
*/
   int    loadCalFile(const char *filename);
/*
** 6200 register level routines
*/
   void setColumn(byte column, word  data);
   word getColumn(byte column);
   void setConfig(byte config);
   byte getConfig(void);

   int  setBusWidth(byte width);
   int  getBusWidth(void);

   void setRowWildcard(byte  row_wildcard);
   byte getRowWildcard(void);

   void setColumnWildcard(byte  col_wildcard);
   byte getColumnWildcard(void);

   void setMask(word  mask);
   word getMask(void);
```

```
        void    setMap(byte offset,word data);
   word getMap(byte offset);

        void setMap(MapRegister &MapReg);
        void getMap(MapRegister &MapReg);


   void setDeviceId(byte device_id);
   byte getDeviceId(void);
   byte scanDeviceId(void);


        enum chipId {XC6216_ID=1,XC6209_ID=2,XC6264_ID=3,XC6236_ID=4};


protected:
/*
** config register
*/

struct ConfigRegBits {
   byte  speed:2;
   byte  bus_width:2;
   byte  ttl_cmos:1;
   byte  unused1:1;
   byte  clock_enable:1;
   byte  unused2:1;
   };

union ConfigRegister {
   ConfigRegBits        bits;
   byte                         all;
        };
private:


/*
** Bus speeds
*/
        enum busSpeeds {GCLK_16=0,GCLK_8=1,GCLK_4=2,GCLK_2=3};

/*
** Bus widths
*/
        enum busWidths {BUS_8=0,BUS_16=1,BUS_32=2,BUS_ILLEGAL=3};

/*
** TTL or CMOS
*/
        enum TTLCMOS {TTL=0,CMOS};

/*
** Device ID
*/
        static const char XC6000_ID[];


}; // end of class XC6200DS

#endif  /* _XC6200DS_H_ */
```

PCIBOARD.h File

```
/*-----------------------------------------------------------------------------

  @(#)pciboard.h      1.1 6/18/97

  Copyright (C) 1993-1997 Xilinx Development Corporation.
  This software is the subject of a licence agreement
  with Xilinx Inc. Use outside the terms of that agreement is prohibited.

  Owner: ***

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

#ifndef  _PCIBOARD_H_
#define  _PCIBOARD_H_

#include <time.h>
#include "xcaddr.h"
#include "pcicore.h"

class PCIBoard:public PCICore{

public:

/*
** constructor destructor
*/
        PCIBoard();

        ~PCIBoard();

/*
** Board level routines
*/
   void         reset(void);
   void         clear(void);

/*
** clock chip functions
*/
   void         clockOn(void);
   void         clockOff(void);
   void         clockStep(word steps = 1);
   void         g1clockOn(void);
   void         g1clockOff(void);
   void         g1clockStep(word steps);
/*
** read con port
*/
   word getCon(void);
/*
** SRAM access modes
*/
   void         setBankControl(word state);
   void         setBankControl(byte bank1, byte bank2);
   word         getBankControl(void);
```

```
/*
** Event status
*/
   void         setEventMask(word mask);
   word         getEventStatus(void);
   void         acknowledgeEvents(word mask);
/*
** clock frequency
*/
   void         sendClockData(word bits, int count);
   void         sendProtocolWord(void);
   double       setClockFreq(double ref_mhz, double desired_mhz);
   void         sleep(clock_t wait);
/*
** current management
*/
   void                 setAdc(byte reset_enable);
   byte                 getAdc(void);
   double               adcToMa(byte ad_value);
   inline byte          getAdcThreshold(void) {return _adcThreshold;};
   inline void          setAdcThreshold(byte adcThr) {_adcThreshold=adcThr;};
   inline byte          getResetEnable(void) {return _resetEnable;};

/*
/** Reset enable on or off for thresholds
*/
   enum resetEnable {RESET_ENABLE_OFF=0,RESET_ENABLE_ON=1};

/*
** Bank Control
*/
   enum BankControl
{SRAM6200_NONE=0,SRAM6200_BANK1=1,SRAM6200_BANK2=2,SRAM6200_BOTH = 3};


/*
** Event and IRQ bits
*/
   enum EventBits {        I_STEP = 0x001,I_6200 = 0x002,I_ADC  = 0x004,
                                                        E_STEP = 0x100,E_6200 =
0x200,E_ADC  = 0x400};

private:
/*
** clock frequency setup
*/
   int  stuffBits(int count, word bits, word* stuffed_bits);
        byte  _last3;    /* Last 3 bits sent to IDC2053B */
        byte    _refClock;
/*
** current thresholds and state
*/
        byte    _adcThreshold;
        byte    _resetEnable;
/*
** PCI Control registers
*/
enum PCIControlReg {
        G1_STEP      = 0x04, // step g1 clock n times
        CLOCK_CTL    = 0x08, // clock control
```

```
        COL_WR       = 0x0c, // Set address for stalled dma counter
        RESET        = 0x10, // reset chip
        CLOCK_GEN    = 0x14, // clock gen
        CLOCK_ON     = 0x18, // continuous clock mode
        SINGLE_STEP  = 0x1c, // single step clock, turn off continuous mode
        CON_8        = 0x20, // 8 bit output port from 6200
        ADC          = 0x24, // Analog-to-digital current meter (read and write)
        ADC_THRESHOLD= 0x24, // ADC threshold
        STEP_REG     = 0x28, // Step
        BANK_CTL     = 0x2c, // Bank Control
        EVENT_STATUS = 0x30, // Event status
        EVENT_ACK    = 0x30, // Event acknowledge
        EVENT_MASK   = 0x34, // Event mask
};

/*
**  Conversion parameters for A/D converter / current meter
*/
        static const double A_D_RESISTOR ;
        static const double A_D_SCALE      ;
        static const double A_D_STEPS           ;


}; // end class PCIBOARD

#endif  /* _PCIBOARD_H_ */
_
```

PCICORE.h File

```
/*-----------------------------------------------------------------------------

  @(#)pcicore.h       1.4 6/24/97

  Copyright (C) 1993-1997 Xilinx Development Corporation.
  This software is the subject of a licence agreement
  with Xilinx Inc. Use outside the terms of that agreement is prohibited.

  Owner: ***


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

#ifndef  _PCICORE_H_
#define  _PCICORE_H_

//#include <windows.h>
#include <afx.h>
#include "bools.h"

#ifndef WINNT

/*
** Device Driver Interface
** Interrupt processing
*/

typedef DWORD(*OPENVXDHANDLE)(HANDLE);
ULONG __stdcall IntSvcRoutine(LPVOID);
ULONG __stdcall VxDSemaphore(LPVOID hEvent);
extern HANDLE hEvent[2];
extern OPENVXDHANDLE OpenVxDHandle;
extern DWORD r0hdl;

#endif

extern bool     b_GlobalIntFlag;
extern HANDLE hDevice;
extern word IntReg;


/*
** PCIconfig structure
*/

typedef struct
{
  /* -- PCI Configuraton Registers     */
  WORD    VendorID;        /* 00 vendor ID (read-only)         */
  WORD    DeviceID;        /* 02 device ID (read-only)         */
  WORD    Command;         /* 04 command register              */
  WORD    Status;          /* 06 status register               */
  BYTE    RevisionID;      /* 08 revision ID                   */
  BYTE    Class1;          /* 09 class code1                   */
  BYTE    Class2;          /* 0A class code2                   */
  BYTE    Class3;          /* 0B class code3                   */
  BYTE    Cache;           /* 0C cache line size               */
  BYTE    Latency;         /* 0D latency timer                 */
```

```
  BYTE    Header;          /* 0E header type                    */
  BYTE    Bist;            /* 0F Built-In Self-Test             */
  DWORD   BaseAddr;        /* 10 base address 0                 */
  DWORD   Reserved1;       /* 14 base address 1 (unused)        */
  DWORD   Reserved2;       /* 18 base address 2 (unused)        */
  DWORD   Reserved3;       /* 1C base address 3 (unused)        */
  DWORD   Reserved4;       /* 20 base address 4 (unused)        */
  DWORD   Reserved5;       /* 24 base address 5 (unused)        */
  DWORD   Reserved6;       /* 28 CardBus CIS pointer (unused)   */
  WORD    SubVendorID;     /* 2C subsystem vendor ID or 0000h   */
  WORD    SubSystemID;     /* 2E subsystem ID or 0000h          */
  DWORD   ExROMAddr;       /* 30 expansion ROM base address     */
  DWORD   Reserved7;       /* 34 reserved (unused)              */
  DWORD   Reserved8;       /* 38 reserved (unused)              */
  BYTE    IntLine;         /* 3C interrupt line                 */
  BYTE    IntPin;          /* 3D interrupt pin (read-only)      */
  BYTE    MinGrant;        /* 3E min time bm bus ownership/250ns*/
  BYTE    MaxLatency;      /* 3F max latency, in 250ns units    */
  /* -- Extended PCI Config Registers  */
  DWORD   Extended1;       /* 40 reserved (unused)              */
  DWORD   Extended2;       /* 44 reserved (unused)              */
  DWORD   Extended3;       /* 48 reserved (unused)              */
  DWORD   Extended4;       /* 4C reserved (unused)              */
  DWORD   Extended5;       /* 50 reserved (unused)              */
  DWORD   Extended6;       /* 54 reserved (unused)              */
  DWORD   Extended7;       /* 58 reserved (unused)              */
  DWORD   Extended8;       /* 5C reserved (unused)              */
  DWORD   Extended9;       /* 60 reserved (unused)              */
  DWORD   Extended10;      /* 64 reserved (unused)              */
  DWORD   Extended11;      /* 68 reserved (unused)              */
  DWORD   Extended12;      /* 6C reserved (unused)              */
  DWORD   Extended13;      /* 70 reserved (unused)              */
  DWORD   Extended14;      /* 74 reserved (unused)              */
  DWORD   Extended15;      /* 78 reserved (unused)              */
  DWORD   Extended16;      /* 7C reserved (unused)              */
  DWORD   Extended17;      /* 80 reserved (unused)              */
  DWORD   Extended18;      /* 84 reserved (unused)              */
  DWORD   Extended19;      /* 88 reserved (unused)              */
  DWORD   Extended20;      /* 8C reserved (unused)              */
  DWORD   Extended21;      /* 90 reserved (unused)              */
} PCICONFIG_S;


/*
** Core PCI class
*/

class PCICore
{
public:
/*
** Constructor, Destructor
*/
        PCICore();

        ~PCICore();


/*
```

```
** write and read to 6200
*/
   void write6200(word addr, word  data);
   word  read6200(word addr);
/*
**    write and read IO space
*/
   void writeIOSpace(unsigned short addr, word  data);
   word  readIOSpace(unsigned short addr);
/*
** write and read to SRAM
*/
   void writeRAM(word addr, word *data, word count);
   void readRAM(word addr,word *data, word count);
/*
**    Debug routines
*/
   PCICONFIG_S   * dumpPCIConfigHeader();
   inline unsigned short        getIOBase(void) {return _IOBase;};
   inline word   getPhysicalBase(void) {return _physicalBase;};
   inline word   getMembase(void) {return _memBase;};

/*
** 6200 address offset
*/
        static const word _offset6200;


/*
** Error codes
*/
enum PCIError {
               EPCI_SUCCESS=0,
               EPCI_NO_DEVICE_DRIVER=1,
               EPCI_DD_INIT_FAILED=2,
               EPCI_BAD_READ=3,
               EPCI_NO_CAL_FILE=4,
               EPCI_BAD_CAL_FILE=5,
               EPCI_BAD_BUS_WIDTH=6,
               EPCI_BAD_CONFIG=7,
               EPCI_LAST_ERROR=8
               };

/*
/* Error Message Strings
*/
        static char *PCIErrorMsg[];

/*
** debug
*/
   char *getErrorMessage(byte  error_code);

/*
** checkBoard
*/
        inline int checkBoard(){return _deviceFailure;};
```

```
protected:

  int _deviceFailure;

/*
** Device driver initialised
*/
        bool _DDInit;

/*
** IO address offset
*/
        unsigned short  _IOBase;
/*
** Physical address of PCI board
*/
        word   _physicalBase;
/*
** Memory mapped address of PCI board
*/
        word _memBase;
/*
** PCI header structure
*/
        PCICONFIG_S _pciconfig;
/*
** required for device IO control
*/
unsigned long _cb;


private:

/*
** Device Driver controls
*/
        enum DDControls {   DIOC_WRITE= 1,          // Burst Memory Write
                            DIOC_READ=2,            // Burst Memory Read
                            DIOC_INT_EN=3   ,       // Enable Interrupt Service
Routine
                            DIOC_INT_DI=4,          // Disable Interrupt Service
Routine
                            DIOC_GETDEVINFO=5,      // Get PCI Configuration
Information
                            DIOC_SNOOP=6,           // Snoop Base Linear Memory
Address
                            DIOC_IO_BASE=7,         // Snoop Base I/O Port
Address
                            DIOC_IO_WRITE=8,        // Write to I/O Port
                            DIOC_IO_READ=9 };       // Read from I/O Port

}; // end class PCICore


#endif  /* _PCICORE_H_ */
```