ATB0 Engineering Document - Software

SCALE Group
MIT Computer Science and Artificial Intelligence Laboratory

Contents

List of Figures

List of Tables

1 Introduction

This document is one in a set of three engineering documents describing the Assam Tester Baseboard (ATB0); this document describes the software interface while the other two describe the actual hardware [1] and the controller [2]. This document assumes the reader has a general understanding of ATB0 and its uses and only discusses the software interface.

The software developed for ATB0 is designed to provide a straightforward interface for a user to access both ATB0 and a daughtercard connected to ATB0. As shown in Figure 1, the software is designed to run on top of the Linux operating system and provides access at multiple levels. At the lowest level is the PLX device driver which provides communication with the PLX interface card that connects the host PC to ATB0. Using this driver, a program can manually configure the PLX card and write to or read specific addresses in ATB0's address space. The PLX diagnostic tool (diag) makes use of the driver and allows a user to use a command line interface to configure the PLX card, read and write to the onboard EEPROM, and read and write directly to ATB0. The xconfig utility uses the driver to program the Xilinx FPGA on ATB0 with a user-provided bit stream. Finally, the ATB0 API provides an extra layer of abstraction and allows applications to configure ATB0 and access the daughtercard without knowing the details involved. Applications at this level would include programs such as ATC0_LabBench, HTIF and MemIF wrappers (to interface with the Scale simulation infrastructure), or the provided application, the ATB0 Console, which provides a text based console interface to ATB0.

This document takes a bottom up approach and begins with a description of the PLX driver in Section 2 followed by the utilities that make use of the driver in Section 3. The API is described in Section 4, and utilities that make use of the API are described in Section 5.

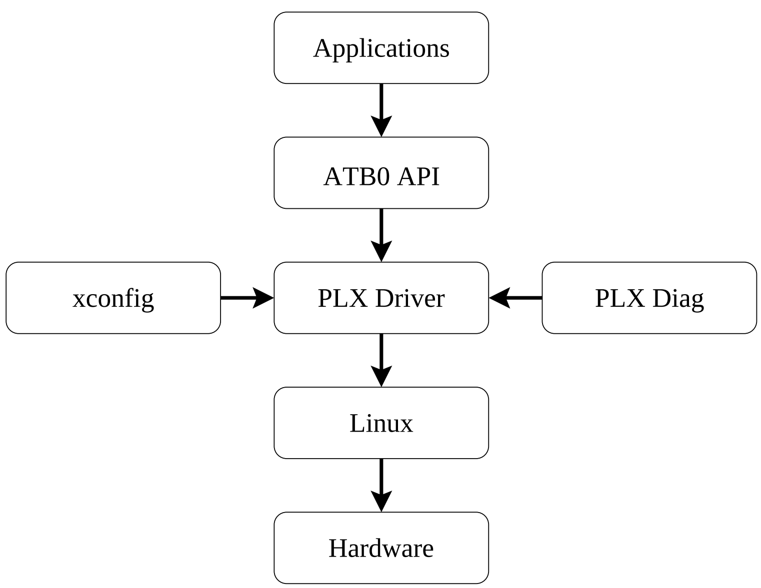

Figure 1: Software components of ATB0

Command	Argument	Description
PLX_IOC_BUSWIDTH	int buswidth	Sets the bus width of the PLX device. The LCR LAS0BRD is changed accordingly and the bus width is saved and used for seeking, reading, and writing. Supported bus widths are 8 and 32.
PLX_IOC_RESET	none	Toggles bit 30 of the CNTRL LCR of the PLX. This resets both the PLX and sends a reset signal on Local Bus 0 whose reset signal is connected to the PRGM pin of the Xilinx on ATB0.
PLX_IOC_READLCR	struct plx_ioc_reg lcr	Reads the LCR at address lcr.address and stores it in lcr.value.
PLX_IOC_WRITELCR	struct plx_ioc_reg lcr	Writes lcr.value to the LCR at address lcr.address.

Table 1: IOCTL functions provided by the PLX device driver.

2 PLX Device Driver

The PLX device driver is written as a module of the Linux 2.4 kernel to provide basic low level access to the PLX, allowing a user program to read and write to the local configuration registers and to Local Bus 0 of the PLX. Many macros and memory management functions new to version 2.4 of the kernel are used, so the driver is not backward compatibly with version 2.2, but should run on a 2.6 kernel. Much of the code is based on sample code from Allessandro Rubini's book *Linux Device Drivers* [3]. This document assumes both a general understanding of how the PLX works (see [1][4] for more information), and basic knowledge of the Linux kernel and how Linux device drivers work (see [3] for more information).

2.1 User interface

The PLX driver is written to use the device file with major number 127; therefore, it is necessary to create this file (usually /dev/plx), with mknod, before loading the device driver module into the kernel using insmod or modprobe. Once the module is loaded into the kernel, a user program can open the /dev/plx file and use it as any other character device. The bus width used in all transactions can be configured to be either 8 bits or 32 bits using the ioctl function as described below. Reading or writing to an offset within the file reads or writes to that same offset within Local Address Space 0 of the PLX which is then sent to ATB0. Currently, only one word can be written or read at a time; therefore, the number of bytes read or written must be 1 if the bus width is 8 bits, and 4 if the bus width is 32 bits.

The device driver implements a few ioctl functions, one to read and one to write to the Local Configuration Registers (LCR) of the PLX, one to set the bus width, and one to reset the PLX. The ioctl functions are described in Table 1. The plx_ioc_reg structure is used to read and write to a LCR, its elements are described in Table 2. A header file, plx.h, is provided and defines the plx_ioc_reg structure and various constants described in Table 3.

The driver also provides support for the mmap command, which allows portions of device I/O memory to be mapped into user virtual memory. See the mmap man page for more information on how this function can be used.

Element	Type	Description
offset	uint32_t	The offset of the register relative to the bottom of Local Configuration Register space on the PLX.
value	uint32_t	The value read from the register after a PLX_IOC_READLCR, or the value to write to the register during PLX_IOC_WRITELCR.

Table 2: Elements of the plx_ioc_reg structure. This structure is used to read and write values to the Local Configuration Registers using the ioctl function.

Constant	Value	Description
PLX_IOC_MAGIC	0xA5	A "magic" number used as the ioctl function type to insure the function is valid.
PLX_IOC_READLCR	_IOWR(PLX_IOC_MAGIC, 1, 4)	ioctl function for reading a LCR.
PLX_IOC_WRITELCR	_IOWR(PLX_IOC_MAGIC, 2, 4)	ioctl function for writing a LCR.
PLX_IOC_RESET	_IOWR(PLX_IOC_MAGIC, 3, 4)	ioctl function for resetting the PLX.
PLX_IOC_BUSWIDTH	_IOWR(PLX_IOC_MAGIC, 4, 4)	ioctl function for setting the bus width of the PLX.
PLX_LCR_CNTRL	0x50	Address of the CNTRL LCR.
PLX_LCR_INTCSR	0x4c	Address of the INTCSR LCR, Interrupt Control/Status.
PLX_LCR_LAS0RR	0x00	Address of the LAS0RR LCR, Local Address Space 0 Range Register.
PLX_LCR_LAS0BA	0x14	Address of the LAS0BA LCR, Local Address Space 0 Base Address (Remap).
PLX_LCR_LAS0BRD	0x28	Address of the LAS0BRD LCR, Local Address Space 0 Bus Region Descriptors.

Table 3: Constants defined in plx.h.

2.2 An example: using the PLX device driver

As an example, Figure 2 shows an entire program that uses the driver to write to a voltage set register on ATB0 to set the desired voltage of a power supply. After dealing with command line arguments, the /dev/plx file is opened and a check is made to make sure it was opened okay. The bus width is set to 32 using ioctl. The PLX User I/O pins are configured by reading the CNTRL LCR, clearing the bottom 12 bits, and reseting them to 0x490, setting the direction of all User I/O pins as output. This is not strictly necessary but done here to show how an LCR can be manipulated. The address of the VSR is calculated using the requested power supply and the data to write is calculated using the desired voltage and VREF, see [2] for more information. Once the address and data are known, lseek is used to go to the correct position within the file and the data is written using a call to write. To read the register back, lseek must be used to set the position again and the call to read reads the VSR and puts the result in data. The result is printed to the screen, the file is closed, and the program ends.

Alternatively, the block of memory containing the voltage set registers could remapped to user memory using mmap and written to directly. Figure 3 shows the code to do this, skipping the code to open and configure the PLX, which is the same as in Figure 2.

2.3 Implementation

The driver is written in one C file, plx3b.c, and one header file, plx.h. The following sections describe the implementation of the PLX device driver.

2.3.1 Initialization

When a module is loaded into the Linux kernel, the init_module() function is called. The init_module() function in the PLX driver begins by using the register_chrdev function to register the device as a character device with major number 127. This step could be replaced with dynamic device numbering, but since the driver is meant to be run on very few computers and dynamic numbering would require creating a new /dev/plx file each time the module is loaded, the number 127 is used as a static number. This is okay because 127 is currently unused.

Once the device has been registered, pci_present is called to make sure there is a PCI bus to search, and pci_find_device is used to locate the PLX device on the PCI bus. When the device is found, two memory regions are remapped into memory space using ioremap_nocache; the pci_resource_start function is used to determine the physical addresses to remap. The first region that is remapped is the local configuration register space; it is 128 bytes long and located using Base Address Register 0. The second region is local address space 0, it is 132 MB long and located using Base Address Register 2. Because the driver assumes that local address space 0 is configured to be 132 MBs, this must be programmed into the EEPROM on the PLX so the appropriate amount of memory can be set aside when the computer boots up. This can be done using the diag tool (described in Section 3.1) to set the LAS0RR LCR in the EEPROM to 0x0800000.

Once the two regions of memory have been remapped, LAS0BRD is set to a safe value with 32 bit accesses and LRDY disabled. Finally, four bytes of kernel memory are allocated using kmalloc to save the current bus width. The device is then ready to be used.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdint.h>
#include "plx.h"

#define VSR_BASE 0x0210000
#define VREF 4.10

int main(int argc, char **argv)
{
  int fd, ps;
  uint32_t address, data;
  double volts;
  struct plx_ioc_reg lcr;

  /* Check command-line arguments */
  if( argc < 3) {
    printf("Usage: vs ps# voltage\n"); exit(0);
  }

  /* Get arguments from command-line */
  ps = atoi(argv[1]);
  volts = atof(argv[2]);

  /* Open device file and check result */
  fd = open("/dev/plx", O_RDWR);
  if( fd < 1 ) {
    printf("Unable to open device! (errno = %d)\n", errno); exit(1);
  }

  /* Set buswidth to 32 */
  ioctl(fd, PLX_IOC_BUSWIDTH, 32);

  /* Setup PLX USER pins */
  lcr.offset = PLX_IOC_CNTRL;
  ioctl(fd, PLX_IOC_READLCR, &lcr);
  lcr.value &= ~0xfff;
  lcr.value |= 0x490;
  ioctl(fd, PLX_IOC_WRITELCR, &lcr);

  /* Calculate correct address and data */
  address = VSR_BASE + (ps << 4);
  data = (uint32_t) (volts / VREF * (4096.0 - 1.0));

  /* Write the data */
  lseek(fd, address, SEEK_SET);
  write(fd, &data, 4);

  /* Read the data back */
  lseek(fd, address, SEEK_SET);
  read(fd, &data, 4);
  printf("VSR%d (0x%x) = 0x%x\n", ps, address, data);

  /* cleanup and return */
  close(fd);
  return 0;
}
```

Figure 2: Example use of PLX driver, writing to a Voltage Set Register.

```
#include <sys/mman.h>

.
.
.

  /* Calculate correct offset and data */
  offset = (ps << 4);
  data = (uint32_t) (volts / VREF * (4096.0 - 1.0));

  /* Remap the voltage set registers */
  uint32_t *VSR = (uint32_t *)mmap(0, 0x100, PROT_READ | PROT_WRITE, MAP_SHARED, fd, VSR_BASE);

  /* Write the data */
  VSR[offset] = data;

 /* Read the data back */
 printf("VSR%d = 0x%x\n", ps, VSR[offset]);

 /* cleanup and return */
 munmap(VSR, 0x100);
 close(fd);
 return 0;
}
```

Figure 3: Using mmap to write to a Voltage Set Register.

2.3.2 File operations

The driver defines functions for the file operations open, release, read, write, llseek, mmap, and ioctl. In the open function, the driver first checks to see if the device is already open using the MOD_IN_USE macro, and if so, the EBUSY error is returned, only one process can open the device at a time. If the device is not in use, the MOD_INC_USE_COUNT macro is called to remember that the device is now in use. The bus width bits of the LAS0BRD LCR are checked and the saved bus width is adjusted accordingly, in case the user has configured this register manually. The f_pos element of the file structure is set to 0 and the function returns. In the release function, the MOD_DEC_USE_COUNT macro is called to indicate the device is no longer in use and the function returns.

The llseek function is used to move the file pointer contained in the f_pos element of the file structure given to all file operation routines. The f_pos contains the offset in number of units of the current bus width, this offset is translated into a byte address before being sent to ATB0. For example, a f_pos of 15 translates to an address of 0xF if the bus width is 8, but translates to an address of 0x3C (15 * 4 = 60 = 0x3C) if the bus width is 32. This is hidden from the user in the llseek function by dividing the offset argument by 4 if the bus width is 32, so the offset the user sends to the lseek function should always be a byte address.

Two seek origins are supported by the driver, SEEK_SET and SEEK_CUR. If the origin is SEEK_SET, f_pos is set to the offset passed if the bus width is 8 or offset/4 if the bus width is 32, otherwise an error is returned. If the origin is SEEK_CUR, f_pos is set f_pos + offset if the bus width is 8 or f_pos + offset/4 if the bus width is 32. An error is returned if the origin is not SEEK_SET or SEEK_CUR, an error is also returned if the bus width is 32 and the offset is not word aligned (divisible by 4).

In the read function, the count is checked to make sure it is only one unit of the current bus width (count should be 1 if bus width is 8 and 4 if bus width is 32) and that the current bus width is either 8 or 32. If the count and bus width are both okay, then either readb or readl is used to read the value pointed to by f_pos. If the bus width is 8, readb is used, if it is 32, readl is used. These functions cause the PLX to perform the requested transaction with ATB0 and return the result. The address sent to readb/readl is obtained by adding f_pos to the bottom of the remapped address space of PLX local address space 0 obtained during initialization (see Section 2.3.1). The result of the read is copied from kernel memory to user memory using copy_to_user, the f_pos is incremented by one, and the function returns.

Unfortunately, the readb and readl functions block while the PLX performs the transaction with the ATB0 controller. Because of this, care must be used when reading and writing; if LRDY is enabled in LAS0BRD and ATB0 does not respond (meaning it never drops LRDY, which will happen if the controller is not programmed onto the Xilinx or does not see the request for whatever reason), the system will hang, forcing a hard reboot. This is a problem that needs a solution.

The write function works much the same way as a read. The count and current bus width are checked, the data to write is copied from user memory to kernel memory using copy_from_user, and either writeb or writel is used to write the data to the address pointed to by the f_pos, depending on the current bus width. f_pos is incremented by one and the function returns. Like readb/readl, writeb/writel block while the PLX performs the transaction with the ATB0 controller, so care must be used when writing.

The mmap function is implemented by calling remap_page_range to generate a new page table using the virtual address and size passed to the function by the kernel and the physical address obtained by adding the given offset to the bottom of the remapped local address space 0. Both the VM_IO and VM_RESERVED flags are set so the kernel does not attempt to swap the memory to disk. Most of the work of actually mapping the memory is done by the kernel either before the function in the driver is called, or in the remap_page_range function.

When the ioctl function is called, the type of command, obtained from the cmd argument, is checked for the magic number. If the command is not okay, an error is returned. All ioctl commands read or write to a LCR; to do so the readl and writel functions are used. The offset passed to the readl/writel function is obtained by adding the supplied offset to the bottom of the 128 byte remapped local configuration register address space obtained during initialization. If the command is PLX_IOC_RESET, the CNTRL LCR is read, the value read is written back with bit 30 set high, then written back with bit 30 set low. If the command is PLX_IOC_BUSWIDTH, the LAS0BRD LCR is read, the value read is written back with bits 22 and 23 set to represent the new buswidth, and the bus width is saved for future reference. If the command is PLX_IOC_READLCR, the offset is obtained from the lcr structure in user space using the get_user function, the LCR is read, then put_user is used to set the value element of the lcr structure. If the command is PLX_IOC_WRITELCR, both the offset and value are obtained from the lcr structure using the get_user function and the value is written to the LCR. The LCR is then read back and the result is written to the value element of the lcr structure using put_user so the user can insure that the value was actually written.

3 Low-level Utilities

Two utilities are provided that make use of the device driver directly and do not need the Xilinx configured with the ATB0 controller to be used. The diag program (Section 3.1) provides low level access to the PLX device, allowing the user to read and write directly to the Local Configuration Registers, the EEPROM, and local address space 0, which is forwarded to ATB0. The xconfig program (Section 3.2) configures the Xilinx on ATB0 with a bit stream file.

3.1 PLX Diag

The PLX diag program provides low level access to the PLX device and uses the device driver directly to communicate with the PLX. The user interface is a subset of the user interface of the p9050_diag program provided with the PLX 9050 RDK. The main menu provides 3 choices: PLX Local Configuration Registers, Address Space 0, and Serial EEPROM; choosing any of these takes the user to a separate sub-menu.

The PLX Local Configuration Registers sub-menu displays the name and current value of each of the 21 LCRs on the PLX. The current values are obtained using the driver's PLX_IOC_READLCR ioctl function. The user can enter a new value for any one of the registers by entering the number displayed next to the register or go back to the main menu. If the user chooses to enter a new value, the new value is written to the LCR using the driver's PLX_IOC_WRITELCR ioctl function and the list of all LCRs is redisplayed.

The Address Space 0 sub-menu allows the user to read from or write to any address in local address space 0. When reading, the user enters an address, lseek is called to move the file pointer to that address, and read is called to read the value. The returned value is simply printed to the screen. When writing, the user enters an address and a data value, lseek is called to move the file pointer to the address, then write is called to write the value.

The EEPROM editor is the most complicated part of the program. The user can display the entire contents of the EEPROM (which is very slow), read a specific dword, or write to a specific dword. Reads and writes to the EEPROM are performed by reading and writing to bits 24 to 28 of the CNTRL LCR as described in the PLX databook [4]. Note that EEPROM offsets are NOT equal to LCR offsets. See Table 3-2 on page 3-3 of the PLX databook [4] to get the EEPROM offset of a LCR.

3.2 xconfig

The xconfig utility makes use of specific features of the PLX to configure the Xilinx with a user provided bit stream. For a description of the process of configuring the Xilinx with timing diagrams, refer to the Xilinx XC4000 databook [5]. This document assumes basic knowledge of that process. The necessary connections are made to allow local address space 0 to be used to download the bit stream to the Xilinx: HAD0 through HAD7 are also wired up to the CFD0 through CFD7 input pins to the Xilinx, the RESET_B signal is wired to the PRGM input pin, the PLX User I/O 0 pin is wired to the Xilinx RDY signal, and the PLX User I/O 1 pin is wired to the Xilinx DONE signal.

After opening the bit stream file and skipping the header information, xconfig opens the PLX device. First the PLX_IOC_RESET ioctl function is called, this resets both the PLX and the local bus, because the reset pin of the local bus is connected to the PRGM input pin to the Xilinx, this raises the Xilinx PRGM pin. The bus width is set to 8 as only 8 bits can be written to the FPGA at a time. The CNTRL register is set to default values with all User I/O pins as input to allow for the value on the RDY and DONE signals from the Xilinx to be read. The LAS0BRD register is configured to have LRDY disabled and 2 NWAD wait states, this causes the PLX to wait two cycles between sending the address and dropping the write strobe signal to send the data. This write strobe that is dropped when the data is sent is wired to the WS input bit of the Xilinx.

With this configuration the bit stream can be sent to the Xilinx by first polling the User 0 I/O pin by reading the CNTRL register and checking bit 2 and waiting for it to go high, this indicates the Xilinx is driving its RDY signal high and it is ready to receive the next byte. The next byte is then sent by writing to local address space 0, the address is ignored and the data is sent as the next byte when the write strobe drops. This process is repeated until each byte in the bit stream has been sent at which point xconfig waits for User 1 to go high to indicate the XILINX has raised the DONE signal, reporting itself configured.

31–28	27–16	15–12	11–0
unused	Value 2	unused	Value 1
4	12	4	12

Figure 4: SDRAM word format

4 ATB0 API

The ATB0 API provides a single class, **atb0**, that makes all the functionality that the ATB0 controller provides available to applications. It is not meant to be used with anything other than the controller and therefore assumes that the controller has been programmed onto the Xilinx on ATB0 before being used. Applications should use the API instead of the driver directly in case design of the controller changes.

To use the API, simply create an **atb0** object; doing so opens the device file and initializes the PLX to be used with the controller. All member functions described in Section 4.1 are then available for use. When finished, deleting the **atb0** object closes the device file.

4.1 Functions

Functions are divided into five sections: basic I/O, User pin I/O, power supply configuration and measurement, other ATB0 configuration, and utilities.

4.1.1 Basic I/O

Basic I/O functions allow the user to read and write to the SDRAM on ATB0 and the Daughtercard via AHIP. In the SDRAM functions, all addresses are offsets into SDRAM memory space and should be a multiple of 4 between 0x0 and 0x2000000. In the AHIP functions, all addresses are offsets into AHIP Daughtercard memory space and should be a mulitple of 4 between 0x0 and 0x4000000.

uint32_t read_sdram(uint32_t addr, SDRAM_WORD word = SDRAM_FULL);

Reads the 32-bit word from **addr** in the SDRAM on ATB0. If **word** is SDRAM_FULL (the default), it returns two 12-bit values from SDRAM, one in the lower 16-bits one in the higher 16-bits as shown in Figure 4. If **word** is SDRAM_HIGH it returns the high word obtained; likewise if **word** is SDRAM_LOW it returns the low word obtained.

Related definitions:
enum SDRAM_WORD {SDRAM_FULL, SDRAM_HIGH, SDRAM_LOW};

void write_sdram(unint32_t addr, uint32_t data);

Writes the 32-bit word in **data** to **addr** in the SDRAM on ATB0. **data** should be in the format shown in Figure 4, partial writes are not possible.

uint32_t *get_sdram_mem(unint32_t start, size_t size);

Maps a chunk of SDRAM memory on ATB0, of size **size** and starting at address **start**, to user memory space on the host and returns the pointer to the mapped memory. **size** is the size in bytes of the requested memory area and should also be a multiple of 4 and between 0 and 0x2000000.

void dump_sdram(std::ostream& out, DUMP_MODE mode, uint32_t start, uint32_t end);

Dumps the contents of SDRAM memory from address **start** to address **end** to the stream **out**. **mode** determines whether to output the values in binary or ASCII and can be DUMP_BINARY or DUMP_ASCII. The resulting dump is one value for every SDRAM location, so for every four bytes between **start** and **end**, two values are written to the output stream (the low bits are written first, followed by the high bits).

Related definitions:

enum DUMP_MODE {DUMP_BINARY, DUMP_ASCII};

uint32_t read_ahip(uint32_t addr);

Reads the 32-bit word from offset **addr** in the AHIP Daughtercard memory space. The controller performs the necessary AHIP operation to communicate with the Daughtercard, the result is dependant upon the daughtercard. If AHIP test mode is turned on (See **set_ahip_testmode**), this performs a test read instead of a normal read. If **addr** is 0 is will perform a test data read; otherwise, it will perform a test address read.

void write_ahip(uint32_t addr, uint32_t data);

Writes the 32-bit word in **data** to offset **addr** in the AHIP Daughtercard memory space. **data** can be any 32 bit value. Like **read_ahip**, the controller performs the necssary AHIP operation. If AHIP test mode is turned on (See **set_ahip_testmode**), this performs a test write.

uint32_t *get_ahip_mem(unint32_t start, size_t size);

Maps a chunk of the AHIP Daughtercard memory space on the ATB0, of size **size** and starting at address **start**, to user memory space on the host and returns the pointer to the mapped memory. **[NOTE: UNTESTED.]**

void set_ahip_mode(AHIP_MODE mode);

Turns AHIP test mode on or off depending on **mode**. If **mode** is AHIP_NORMAL,all AHIP reads and writes are normal 32-bit AHIP reads and writes. If **mode** is AHIP_TEST, AHIP reads and writes are test AHIP reads and writes as described in the descriptions of **read_ahip** and **write_ahip**. Both AHIP_NORMAL and AHIP_TEST have 8-bit conterparts in AHIP_8BIT and AHIP_8BITTEST respectively. Table 4 summarizes these modes.

Related definitions:

enum AHIP_MODE {AHIP_NORMAL = 0, AHIP_TEST = 1, AHIP_8BIT = 2, AHIP_8BITTEST = 3};

AHIP_MODE get_ahip_mode();

Mode	Functionality
AHIP_NORMAL	All reads and writes are normal 32-bit accesses
AHIP_TEST	Reads and writes are 32-bit test AHIP reads and writes. Any read performs a test address read, except a read of address 0, which performs a test data read. Any write performs a test write.
AHIP_8BIT	All reads and writes use the 8-bit AHIP protocol to perform normal accesses
AHIP_8BITEST	Reads and writes behave the same as AHIP_TEST except the 8-bit AHIP protocol is used.

Table 4: AHIP modes.

Returns the current AHIP mode. See the description above for **set_ahip_mode** for a description of the available modes.

void dump_ahip(std::ostream& out, DUMP_MODE mode, uint32_t start, uint32_t end);

Dumps the contents of AHIP Daughtercard memory space from address **start** to address **end** to the stream **out**. **mode** determines whether to output the values in binary or ASCII and can be DUMP_BINARY or DUMP_ASCII. The output is one value for each 32 bit word read. **[NOTE: UNTESTED]**

Related definitions:

enum DUMP_MODE {DUMP_BINARY, DUMP_ASCII};

4.1.2 User Pin I/O

These functions allow a user to access the User pins directly. If a value is assigned to a User pin (either a 0 or 1) in the controller, it is considered an output and is driven by the controller. If the pin is reset, it is considered an input and is not driven by the controller. On startup, all pins are inputs and thus not driven by the controller. In the following functions all pin numbers should be between 0 and 25.

void reset_user_pins();

Resets all User pins to be inputs. (i.e. the controller does not drive them.)

void set_user_pin(int pin, bool val);

Sets the User pin **pin** to be an output and drives it high or low depending on **val**.

void set_user_pins(uint32_t val);

Sets allthe User pins whose direction is set to be an output and drives them with the corresponding bit in **val**.

bool set_user_pin_dir(int pin, bool val);

Sets the direction of the User pin **pin**. A **val** of **true** means the pin is an output and driven by the controller and **false** means the pin is an input and driven by the daughtercard.

bool set_user_pins_dir(uint32_t val);

Sets the direction of all User pins to the corresponding bit in **val**. A bit value of of **true** means the pin is an output and driven by the controller and **false** means the pin is an input and driven by the daughtercard.

bool get_user_pin(int pin);

Returns the logical value seen on the User pin **pin**, whether the pin is driven by the controller or the daughtercard.

bool get_user_pin_dir(int pin);

Returns the direction of the User pin **pin**. Returns **true** if the pin is an output and driven by the controller and **false** if the pin is an input and driven by the daughtercard.

uint32_t get_user_pins(int start, int end);

Returns the unsigned value seen on the User pins [**end**:**start**], whether the pins are driven by the controller or the daughtercard. It assumes that the higher of the two values is the most significant bit. (i.e. if start is greater than end, it will consider start the MSB).

uint32_t get_user_pins_dir();

Returns the direction of all User pins in the low 26 bits of the return value, one bit per pin (Bit 0 is the direction of User pin 0). A value of 1 in a pin's bit indicates that pin is an output and driven by the controller, a value of 0 in a pin's bit indicates that pin is an input and driven by the daughtercard.

4.1.3 Power Supply Configuration and Measurement

These functions allow the user to configure and verify the voltage of each power supply and measure the current drawn from the power supply.

void set_voltage(int ps, double volts);

Sets the desired voltage of power supply **ps** to **volts**. **ps** should be between 0 and 15 inclusive, Volts should be between 0 and about 4 for power supplies 0 to 13 and between 0 and -4 for power supplies 14 and 15. Each time this procedure is called the Voltage Set register for the given power supply is set, then the Voltage Set Register 15 is written to, committing the change.

void set_voltages(double *volts);

Sets the desired voltage of all power supplies. **volts** should be an array of 15 double values corresponding to the desired power supply voltages (i.e. **volts[5]** should contain the desired voltage for power supply 5). Again, volts must be within the acceptable range (see **set_voltage**).

double get_voltage(int ps);

Get the measured voltage across power supply **ps**. This is used to ensure that the desired voltage is actually seen across the power supply and for more precise power measurements.

void get_voltages(double *volts);

Gets the measured voltage of each power supply. **volts** should be array of 15 double values that the voltage measurements are to be written in. Upon returning, this array will be filled with the measured voltages.

get_current(int ps);

Gets the measured current drawn from power supply **ps**. Each power supply has a unique offset and ratio that are used in calculating the current from the value returned by the current measurement ADC using Equation 1 (where CMR is the value returned from the ADC), see the controller documentation [2] for more information. The offset and ratio used for a power supply can be set using **set_calibration**.

$$current = (ratio * CMR) + offset \quad (1)$$

void get_currents(double *currents, uint16_t mask=PSALL

Gets the measured current drawn from each power supply with its bit set in **mask**. **currents** should be large enough to hold all requested measurements and upon returning will contain the measurements in order from the smallest number power supply to the largest. For example, if mask is 0x104 then the **currents[0]** would contain the current from power supply 2 and **currents[1]** would contain the current from power supply 8. To make generating the mask easier, constants are defined for each power supply, PS1 through PS14. These can be or'ed together to create the mask, for example, (PS2 | PS14) = 0x104. PSALL is defined to be the mask to measure all power supplies and is the default mask. The mask functionality is not currently supported and is ignored.

Related definitions:
#define PS0 0x1
...
#define PS14 0x4000
#define PSALL 0x7fff

set_calibration(int ps, double offset, double ratio);

Sets the offset and ratio used to calculate the measured current from power supply **ps**. See **get_current** for a description of how these values are used.

Bit #	Signal
0	LED0
1	LED1
2	LGA0
3	LGA1
4 - 31	unused

Table 5: Bit assignment of LGA_LED register.

4.1.4 Other ATB0 Configuration

These functions are used to set and read the frequency of the clock sent to the daughtercard and set and read the value on the LGA and LED signals in ATB0.

void set_clock(int freq);

Sets the frequency that is generated by the frequency synthesizer on ATB0 to **freq** MHz. **freq** must be between 25 and 400 MHz.

int get_clock();

Returns the frequency, in MHz, that the frequency synthesizer is currently set to generate.

void set_LED(int LED, bool val);

Sets the LED number **LED** to **val**. A **val** of true turns the LED on, false turns the LED off.

void set_LGA(int LGA, bool val);

Sets the LGA ouput number **LGA** to **val**.

uint32_t get_LGALED();

Returns the the two LED values and two LGA values in the bottom four bits of the return value as shown in Table 5.

4.1.5 Other utilites

Utilities are provided to save the state of the power supplies to a file and read the state back from a file.

void save_state(char *filename);

Creates a new file **filename** (overwriting the file if it exists) and saves the current state of the power supplies to that file. For each supply, the desired voltage, calibration offset, and calibration ratio are saved. No other configuration values are saved.

void read_state(char *filename);

Reads the state saved in the file **filename**. The file must be one generated using the **save_state** function. For each power supply the desired voltage, calibration offset, and calibration ratio are set. No other configuration values are changed.

```
#include "atb0API.h"

static double volts[16] = {1.8, 3.3, 3.3, 1.5, 1.8, 1.8, 3.3, 1.8, 1.8, 0, 2.5, 0. 0. 0};

int main()
{
  atb0 bb;
  bb.set_voltages(volts);
  return 0;
}
```

Figure 5: Example use of the ATB0 API. Sets voltages of all power supplies

4.2 API examples

Figure 5 shows a trivial example program that uses the API to set the voltages of all power supplies on ATB0. The voltages happen to be those necessary to power the ADB0 board.

Figure 6 shows an example program that loads test data into memory on the daughtercard and measures the power drawn by the daughtercard. It assumes that the daughtercard is already powered and configured with the test chip and that User pin 0 is an active-low reset of the module which does the AHIP interface and handles the memory and that User pin 1 is an active-high signal that tells the chip to do something with the memory. The test data is declared as an external data array with a seperate variable that contains the number of elements.

Figure 7 shows an alternative way to download the test data to the daughtercard using memory mapping instead of direct writes. This method may be perferred for more complicated applications.

5 High-level utilities

Two utilities are provided that make use of the ATB0 API. console (Section 5.1) provides a text based interface to many components of the daughtercard. sweep (Section 5.2) automates performing a voltage and frequency sweep, measures power, and can generate a schmoo plot of the results.

5.1 ATB0 Console

The console is a utility that provides a text based interface (using ncurses) to the power supplies, frequency synthesizer, and user pins on ATB0. Figure 8 shows a screen shot of the console. A list on the right hand of the screen shows available commands which are mostly self explanatory. When the display is refreshed, all voltages and currents and remeasures and the User pins are resampled. The implementation of the console is straightforward and mostly user interface so it is not described here.

5.2 sweep

Sweep is a simple program that automates the process of measuring power over a large number of voltages and frequencies and is controleld by a number of command line options described in Table 6. One power supply is swept from a minimum voltage to a maximum voltage while the rest of the power supplies are held at a constant voltage. Optionally, at each voltage, the frequency can be swept as well. At each voltage/frequency combination (or at each voltage if the frequency is not swept) the voltage and current of one or all of the power supplies is measured and the power in watts is calculated from the measured values. A

```
#include <iostream>
#include "atb0API.h"

#define MEM_NRESET_PIN 0
#define GO_PIN         1

extern uint32_t *test_data;
extern size_t test_data_count;

using std;

int main()
{
  atb0 bb;
  uint32_t *mem;
  int i;
  double volts[14];
  double currents[14];
  double power, tpower;

  /* Set all voltages using a perviously saved state. */
  bb.read_state(‘‘state.sav’’);

  /* Stop the test from running */
  bb.set_user_pin(GO_PIN, 0);

  /* Reset the memory controller on the daughter card */
  bb.set_user_pin(MEM_NRESET_PIN, 1);
  bb.set_user_pin(MEM_NRESET_PIN, 0);
  bb.set_user_pin(MEM_NRESET_PIN, 1);

  /* download the test data to memory */
  for( i=0; i<test_data_count; i++ )
    bb.write_ahip(i*4, test_data[i]);

  /* Start the test */
  bb.set_user_pin(GO_PIN, 1);

  /* Measure the voltages and currents */
  bb.get_voltages(volts);
  bb.get_currents(currents);

  /* Print out power usage */
  cout << "PowerSupply\tVoltage\tCurrent\tPower" << endl;
  tpower = 0;
  for( i=0; i<14; i++ ) {
    power = volts[i] * (currents[i]/1000);
    cout << i << "\t" << volts[i] << "\t" << currents[i] << "\t" << power << endl;
    tpower += power;
  }
  cout << "Total power: " << tpower << endl;

  return 0;
}
```

Figure 6: Example use of API to access the daughtercard and measure power.

```
.
.
.
  /* get AHIP memory */
  mem = bb.get_ahip_mem(0, test_data_count * 4);

  /* download the test data to memory */
  for( i=0; i<test_data_count; i++ )
    *mem++ = test_data[i];
.
.
.
```

Figure 7: Alternative method to download data to the daughtercard.

```
        Power Supply Values
        -------------------

     Voltage (V)   Current (mA)
     -----------   ------------
                                                          1. Refresh display
  0)    1.809031     37.186520   Clock frequency: 25      2. Refresh continuously
  1)    3.322061     75.853990                          3. Set power supply voltage
  2)    3.320837     96.895880                          4. Set clock frequency
  3)    1.499575     76.928550   SDRAM address: 0x1234567   5. Set user pin
  4)    1.790684     24.140000   SDRAM data: 0x123, 0x678   6. Read SDRAM memory
  5)    1.818816     19.039700                          7. Write SDRAM memory
  6)    3.318391     25.936950                          8. Dump SDRAM memory
  7)    1.807808     23.627500   AHIP address: 0x12345678   9. Read AHIP value
  8)    1.813923     2.360650    AHIP data: 0x87654321      10. Write AHIP value
  9)    0.019570     0.061380                               11. Load configuration file
  10)   2.503777     2.530530                          12. Save configuration to file
  11)   0.017124     0.031000                               13. Quit
  12)   0.022017     1.276580
  13)   2.068337     1.721740                               Your Selection:

             2222221111111111
             54321098765432109876543210
  User pins: 11111011111111011111111111
  Direction: IIIIIOIIIIIIIIOIIIIIOIIIII
```

Figure 8: Screen shot of the console utility.

Option	Default	Description
-v	-	Be verbose.
-ps ⟨*integer*⟩	-	*Required*. Power supply to sweep. Range is 0 - 13.
-allps	-	Include all power supplies in the tab file result.
-pin ⟨*integer*⟩	0	User pin number that indicates success or failure. Range is 0 - 25.
-successhigh	-	Indicates that a high value on the success pin means success. Default is a low value means success.
-nocheck	-	Do not check for success or failure, just measure power.
-pause ⟨*μs*⟩	0	Time, in microseconds, to pause after setting the voltage and frequency and before taking measurements.
-samples ⟨*integer*⟩	100	Number of samples to take for each measurement.
-minv ⟨*volts*⟩	0	Minimum voltage of sweep. Range is 0.0 - 4.0.
-maxv ⟨*volts*⟩	0	Maximum voltage of sweep. Range is 0.0 - 4.0
-stepv ⟨*voltage step*⟩	.1	Voltage step during sweep. Range is 0.0 - 4.0 exclusive.
-maxw ⟨*watts*⟩	2	The maximum wattage to allow before stopping the sweep.
-sf	-	Sweep the clock frequency as well as the voltage.
-minf ⟨*frequency*⟩	25	Minimum clock frequency of sweep. Range is 25 - 400.
-maxf ⟨*frequency*⟩	400	Maximum clock frequency of sweep. Range is 25 - 400.
-stepf ⟨*frequency step*⟩	1	Frequency step during sweep. Range is 1 - 375.
-res ⟨*filename*⟩	"-"	Result file for "pretty" output and schmoo plot.
-tab	-	Generate a tab-delineated result file.
-tabfile ⟨*filename*⟩	results.tab	Tab-delineated result file with all information.
-tabsuccess	-	Include success indication in tab-delineated result file.
-taball	-	Include unsuccessful trials in tab-delineate file, default is only succesful trials are included.
-schmoo	-	Generate a schmoo plot.
-reset	-	Toggle a reset User pin between trials.
-resetpin ⟨*integer*⟩	none	User pin to toggle when -reset is included. Range is 0 - 25
-resetlow	-	Indicates the reset pin is active low (default is active high).
-state ⟨*filename*⟩	none	Load console state file prior to sweeping.

Table 6: Options to the sweep program.

success pin is also checked at each combination to determine if the device under test (DUT) works for that particular combintation. Therefore, the DUT should output a success value on one of the user pins. Figure 9 shows pseudo-code of the process the program goes through.

Sweep does not measure power at a single point in time, but steady state power over time by taking multiple samples for each measurement. Statistics (mean, median, standard deviation, min, and max) for each measurement are provided in a tab-delineated result file. A "pretty" result file is also created for quickly seeing results and only includes the power drawn from the power supply being swept and does not include the statistics.

Like console, the implementation of sweep is straightforward and mostly output formatting and statistics calculation. It is thus not described here.

```
for volts = minv to maxv step stepv
   set_voltage(ps, volts)
   if (sweep_frequency)
      for freq = minf to maxf step stepf
         set_clock(freq)
         reset_DUT
         pause
         measure_power
         check_success
         output_results
         if (power > maxw) exit
      next freq
   else
      reset_DUT
      pause
      measure_power
      check_success
      output_results
      if (power > maxw) exit
   end if
next volts
generate_schmoo
```

Figure 9: Pseudo-code of sweep.

References

[1] Jared Casper. *ATB0 Engineering Document - Hardware*. `assam cvs: atb0/doc/hardware`.

[2] Jared Casper. *ATB0 Engineering Document - Controller*. `assam cvs: atb0/doc/controller`.

[3] Alessandro Rubini Jonathan Corbet. *Linux Device Drivers*. O'Reilly, 2nd edition, 2001. http://www.xml.com/ldd/chapter/book/index.html.

[4] *PLX Technology PCI 9050-1 Data Book*. `assam cvs: atb0/doc/datasheets/9050-1db-20.pdf`.

[5] *Xilinx XC4000E and XC4000X Series Field Programmable Gate Arrays*. `assam cvs: atb0/doc/datasheets/xilinx-4000-series.pdf`.