STT Motherboard Support Package

Evgeny Popkov

October 17, 2001

Abstract

This document describes the STT Motherboard Support Package written by Axel Naumann, Evgeny Popkov and Gabriel Redner. The software provides a device independent way to initialize and access PCI devices on the STT motherboards. The software can automatically detect and initialize the motherboards and PCI devices. PCI bus access routines for the single and burst transfers are available. User-developed device drivers can be easily added to the package. Python-based Client module is provided for remote/interactive access to PCI devices. A description of user interface as well as some examples are included. The document corresponds to the version 2.00 of the package.

1 Introduction

A 9Ux400mm VME64x-compatible motherboard (MB) can host up to 6 PC-MIP and 2 double-size PMC mezzanine cards, provides complete PCI bus services and VME-to-PCI bridge logic.[1] MB has three PCI buses connected by two Texas Inst. PCI2301 PCI-to-PCI bridges.[2] The VME-to-PCI bridge is implemented using Tundra UniverseII chip (U2).[3] MB accommodates up to 14 PCI interfaces. The list of the PCI interfaces is given in Table 1.The column one of the Table shows the names of the interfaces (which correspond to the geographical positions for PC-MIP and PMC sites as illustrated in Figure 1). The columns two and three contain PCI bus and device numbers respectively (the bus numbers in brackets correspond to internal PCI bus numeration scheme which is used in the package). The column four contains device aliases which are used in the package.

The package consist of two parts: VME Input/Output Controller (IOC) software and client software. The VME IOC software is written in C++ and compiled with VxWorks "gcc" compiler for Motorolla MVME-2303 CPU

Device Name	Bus #	Dev #	Alias	Device Type
PCI2301	3 (0)	0	br3	PCI-to-PCI bridge
Logic Board	3 (0)	1	log3	PMC
UniverseII	3 (0)	2	u2	VME-to-PCI bridge
Buffer Controller	3 (0)	3	bc	PMC
Local Control	3 (0)	4	lc	Altera FPGA
PCI2301	2 (1)	0	br2	PCI-to-PCI bridge
Logic Board	2 (1)	1	log2	PMC
MB4	2 (1)	2	mip4	PC-MIP
MB5	2 (1)	3	mip5	PC-MIP
MB6	2 (1)	4	mip6	PC-MIP
Logic Board	1 (2)	1	log1	PMC
MB1	1 (2)	2	mip1	PC-MIP
MB2	1 (2)	3	mip2	PC-MIP
MB3	1 (2)	4	mip3	PC-MIP

Table 1: List of available PCI interfaces on MB.

board. The client part is written in Python and allows remote/interactive access to PCI devices on the MBs.

The package is available from the d0cvs server under the name "stt_mbsp" and from [5] (the later also includes the compiled library for VxWorks).

2 VME IOC Software

It consists of several classes (see more detailed description of the corresponding user interfaces further in the Section):

- **list** class. It is a simple linked list container class.
- **CCrate** class. An instance of that class corresponds to a VME crate. It serves as a top level class which uses lower level classes for detection and initialization of all U2s and PCI devices from a crate. The class contains a pointer to a linked list of **CUniverse2** objects ("list of MBs in a crate").
- **CUniverse2** class. It is derived from **list**. An object of **CUniverse2** class is instantiated for each detected MB. The class allows low level access to settings and transfers with the U2 chip as VME-to-PCI bridge. It is also used to detect and initialize available PCI devices. The class

contains a pointer to a linked list of **CPciDevice** objects ("list of PCI devices on a MB").

- **CU2image** class. It contains parameters of an address space image of U2 chip. The class is used by **CUniverse2** to store information about the U2-specific address space images.

- **CPciDevice** class. It is a purely virtual class derived from **list**. All PCI device ("driver") classes must inherit from it. **CPciDevice** allows low level access to settings and transfers with the PCI devices. It is also used for automatic detection and allocation memory for PCI devices. The class contains a pointer to a linked list of **CMemory** objects ("list of allocated PCI memory space regions for a device"). IMPORTANT: All user-provided device ("driver") classes must be registered with the **CUniverse2::getDeviceObject** method.

- **CMemory** class. It is derived from "list". An instance of that class corresponds to a requested/allocated PCI memory space region for a device.

- **CDefaultDriver** class. It is derived from **CPciDevice**. An object of that class is instantiated when the software detects an "unknown" device (a necessary amount of PCI address space is automatically allocated for device).

- **CPci2031** class. It is derived from **CPciDevice**. An object of that class is instantiated for each detected Texas Inst. PCI2301 PCI-to-PCI bridge. A provided method allows to initialize some device-specific registers of the bridge.

- **CLclCtl** class. It is derived from **CPciDevice**. An object of that class is instantiated for the Local Control FPGA. Provided methods allow to erase and program the on-board flash memory which is used to store the PCI core firmware for a MB. A method is provided to reconfigure (soft reset) the FPGA using the PCI core image from one of the four sectors of the flash memory.

- **CLtb** and **CLrb** classes. They are derived from **CPciDevice**. An object of those classes are instantiated for each detected STT Link Transmitter/Receiver Board (LRB/LTB). Provided methods allow to perform bit error rate tests for data transfers from LTB to LRB.

Several VxWorks/PCI-inspired type and symbol definitions are used throughout the code:

- **DWORD** - 32-bit-wide word ("unsigned int")
- **UCHAR** - at least 8-bit-wide word ("unsigned char")
- **STATUS** - "int"
- **OK** - "0"
- **ERROR** - "-1"

All methods of type **STATUS** return **OK** in case of normal completion and **ERROR** otherwise.

2.1 CCrate

Constructor:

- **CCrate**(int crateNumber)

  where "crateNumber" specifies the crate number (is not used currently). The constructor detects and initializes MBs and PCI devices. A necessary amount of PCI address space is allocated automatically for PCI devices.[1] The following flags are enabled in the Control and Status Register (Configuration Space) for detected PCI devices:

  - PCI address space (PCI Memory and/or I/O Spaces). An address space is enabled only if a device supports it.

  - System and Parity Error response.

  - Master capability. It is enabled only if a device is master-capable.

Accessors:

- Returns a pointer to the "CUniverse2" object which corresponds to VME slot "slotNo" (returns "NULL" if the MB is not accessible).

  const CUniverse2* **getU2**(UCHAR slotNo) const

- Returns a pointer to the first "CUniverse2" object from the "linked list of MBs" (returns "NULL" if the list is empty).

  const CUniverse2* **getFirstU2**() const

[1] If a user-provided device ("driver") class explicitly requests for a region of PCI address space (via **CMemory** contrsructor), the software allocates only the address space region(s) explicitly requested by the driver (no BAR scan is performed).

Address Space Alias	PCI address space	Access type
"conf"	Configuration	single transfers
"mem"	Memory	single transfers
"fmem"	Memory	single/burst transfers
"io"	I/O	single transfers

Table 2: PCI address space types which are supported by the software.

- Returns a pointer to the next "CUniverse2" object from the "linked list of MBs" (returns "NULL" if "pU2" is the last element of the list).

  const CUniverse2* **getNextU2**(const CUniverse2* pU2) const

Setters:

- Sets debugging level for the MB from the VME slot "mbSlotNo" (debugging information is printed at the console terminal of the VME IOC). The following information is printed out on the console terminal for a single read/write access to a PCI device:

  - Alias of PCI address space type accessed. The supported PCI address space type aliases are shown in Table 2.

  Note that the burst transfers are not allowed for the "mem" type of PCI address space only for the case when the burst transfers are performed across U2 chip. Burst transfers between local PCI master and target are allowed for the "mem" type of PCI address space.

  - VME address accessed

  - PCI address accessed

  - Data which was send/received

  - Parity and bus errors as signalled by U2 chip (if any).

  If "debugLevel" is equal to "0" - debugging is off, if "1" - on. The second parameter is optional, if it is omitted the method applies to all MBs.

  void **setDebugLevel**(UCHAR debugLevel, UCHAR mbSlotNo) const

Miscellaneous methods:

- Starts server for remote/interactive access to PCI devices. "portNo" specifies the network port number to be used for connections. The argument is optional, it is defaulted to "40404".

  void **pciAccessServer**(int portNo) const

- Prints some status information about the detected PCI devices on a MB at VME slot "slotNo" (the argument is optional, if it is omitted the method applies to all MBs). The status information is printed at the console terminal of the VME IOC. The following information is printed for each detected device: MB slot number, alias, PCI Id, PCI Revision number, PCI bus number, PCI device number. The following information is printed for each address space region allocated for device: memory type alias, PCI base address, size and Configuration Space Base Address Register (BAR). Hexadecimal representation is used for all addresses.

  void **pciDeviceStatus**(UCHAR slotNo) const

- **This method is disabled in the current version.** Device Configuration file parser method. This method allows to initialize a PCI device using commands from file "fileName". The format of the file is described in Appendix A. "fileName" is the path/name of a Configuration file. The file must be accessible by the VME IOC.

  STATUS **execConfigFile**(const char* fileName) const

- Prints VME and PCI memory maps of the U2 address images and allocated memory regions of the PCI devices which correspond to the MB in the slot number "slotNo". The maps are printed at the console terminal of the VME IOC. The information which is printed for each U2 address image is as follows (hexadecimal representation is used for all addresses): MB slot number, image type, VME address space (VAS) type, VME base address, VME bound address, image size, PCI base address, PCI bound address and the "Misc" word containing some parameters of the U2 images. The format of the "Misc" word is as follows (digits from left to right): image number, code of the PCI address space type, code of the VME Program/Data address modifier, code of the VME Supervisor/User address modifier, posted write flag, prefetch read flag.[2] The information printed for each allocated memory region of the detected PCI device is as follows: MB slot number, device alias, VME address space type, VME base address, VME bound address, image size, PCI base address, PCI bound address, PCI memory type and PCI Base Address Register alias.

  void **printMemMap**(UCHAR slotNo) const

[2]See UniverseII User Manual[3] for the description of the U2 address image codes and flags.

- Saves VME and PCI memory maps of allocated memory regions of the PCI devices which correspond to the MB in the slot number "slotNo". The maps are saved to the file "fileName". The format of the file is described in the previous bullet list item.

  STATUS **saveMemMap**(const char* fileName, UCHAR slotNo) const

- Reads/writes from/to the VME address "addr" of VME address space type code "vas". "vas" code: 0 - A16, 1 - A24, 2 - A32. If "endianessSwapFlag" is "true" the endianess swap of read/write data is performed. The last two parameters are optional and are defaulted to "1" and "true", respectively.

  void **readDW**(DWORD addr, DWORD& value, UCHAR vas, bool endianessSwapFlag) const

  void **writeDW**(DWORD addr, DWORD value, UCHAR vas, bool endianessSwapFlag) const

- Resets PCI bus on a MB in slot number "slotNo". If "slotNo" is equal to zero the reset is applied to all detected MBs. The input parameter is optional and is defaulted to zero. Note that all PCI devices on MB(s) must be reinitialized after the reset.

  STATUS **resetPci**(UCHAR slotNo) const

2.2 CUniverse2

Accessors:

- Returns a pointer to the PCI device which corresponds to the device alias "alias" (the aliases are defined in Table 1). Returns "NULL" if the device is not found.

  const CPciDevice* **getDevice**(const char* alias) const

- Returns a pointer to the first "CPciDevice" object from the "linked list of PCI devices" (returns "NULL" if the list is empty).

  const CPciDevice* **getFirstDev**() const

- Returns a pointer to the next CPciDevice object from the "linked list of PCI devices" (returns "NULL" if "pDev" is the last element of the list).

  const CPciDevice* **getNextDevice**(const CPciDevice* pDev) const

- Returns VME slot number which corresponds to the object.

  UCHAR **getSlotNo**() const

- Resets PCI bus on the MB.

  STATUS **resetPci**()

2.3 CPciDevice

Setters:

- Sets an explicit request for allocation of a region of a PCI address space[3]. Must be used in conjunction with the "CMemory" constructor.

  void **addMemReq**(CMemory* pMem)

- Sets the name of the device class ("driver").

  void **setDrvName**(const char* drvName)

Accessors:

- Returns PCI Vendor Id.

  const DWORD **getVendorID**() const

- Returns PCI Device Id.

  const DWORD **getDeviceID**() const

- Returns PCI Id (Vendor Id + (Device Id <<16)).

  const DWORD **getPciInfo**() const

- Returns Revision Number of PCI device.

  const DWORD **getRevisionNo**() const

- Return PCI bus and device numbers.

  const int **getDevNo**() const

  const int **getBusNo**() const

- Returns the type of device (regular or bridge).

  const bool **connectsBus**() const

[3] If a user-provided device ("driver") class explicitly requests for a region of PCI address space (via **CMemory** contrsructor), the software allocates only the address space region(s) explicitly requested by the driver (no BAR scan is performed).

- Returns an alias for the geographical position of device (see Table 1 for the description of the aliases).

  const char* **getDevAlias**() const

- Returns the name of the driver for device.

  const char* **getDrvName**() const

- Returns pointer to the first "CMemory" object from the "device's memory list" (returns "NULL" if the list is empty).

  const CMemory* **getFirstMem**() const

- Returns pointer to the next "CMemory" object from the "device's memory list" (returns "NULL" if "pMem" is the last element of the list).

  const CMemory* **getNextMem**(const CMemory* pMem) const

- Returns pointer to the "CMemory" object which corresponds to the device's PCI address space region addressed by the BAR with offset "barOffs" (returns "NULL" if there is no "CMemory" object which corresponds to the BAR).

  const CMemory* **getMem**(DWORD barOffs) const

PCI access methods:

- PCI Configuration, Memory and I/O address space access routines (single transfers).

  STATUS **readDW**(DWORD offs, DWORD& value, const char* image_type, DWORD barOffs, UCHAR devFunc) const

  STATUS **writeDW**(DWORD offs, DWORD value, const char* image_type, DWORD barOffs, UCHAR devFunc) const

  where "offs" is address offset relative to the base of the corresponding PCI address space region of device. "image_type" specifies the type of the PCI address space accessed (see the "address space alias" column from the Table 2 for the list of aliases). "barOffs" - Configuration Space offset for the BAR which points to the accessed PCI address space region, "devFunc" specifies the PCI Device Function (used for Configuration Space access only). The routines respect the endianess of input and output. The last three arguments are optional: "image_type" is defaulted to "conf", "barOffs" - to 0x10 (BAR0) and "devFunc" - to "0".

- PCI burst transfer routines. They utilize the DMA Channel of the U2 chip of MVME2303 board. The burst transfer CPU <->PCI device is only possible when the device has PCI "burst-transfer-capable" Memory Space allocated ("fmem"). The software does not allocated such a memory automatically so a device class ("driver") must explicitly request for such memory (using the "CPciDevice::addMemReq" method and the "CMemory" constructor). The routines respect the endianess of input and output.

  STATUS **readBurst**(DWORD offs, DWORD* pArray, int size, DWORD barOffs) const

  STATUS **writeBurst**(DWORD offs, DWORD* pArray, int size, DWORD barOffs) const

  where "offs" is the address offset relative to the base of the "fmem" address space region of device, "pArray" - pointer to array containing the output/input data, "size" - the total size of data block to be transferred (bytes), "barOffs" - address offset of the BAR which corresponds to the "fmem" address space region. The last argument is optional and is defaulted to 0x10 (BAR0).

- Data file parser method. This method allows to write data from a file to a PCI device.

  STATUS **execAdpFile**(const char* fileName, const char* memType, DWORD barOffset, UCHAR funcNo) const

  The last three arguments of the method are similar to the ones defined in the "writeDW" method. The last three arguments are optional and defaulted to "mem", "0x10" (BAR0) and "0", respectively. "fileName" is the path/name of the file containing address offset / data pairs. The file must be accessible by the VME IOC. The address offset is taken to be the offset with the respect of the base of the allocate region of PCI address space of device as it specified by the method arguments. The file must contain one offset/data word pair per line (offset word is first, data is second). The offset and data words must be in hex format and separated by at least one empty/tab space.

- Data file parser method. This method allows to write data from a file to a PCI device.

  STATUS **execIHexFile**(const char *fileName, bool byteFlag, const char* memType, DWORD barOffset, UCHAR funcNo) const

The last three arguments are the same as the ones defined in the description of the previous method. "fileName" is the path/name of the file containing data in the Intel Hex (IHex) format.[7]. "byteFlag" specifies the IHex file parsing mode. If "byteFlag" is "true" the data is parsed using one byte alignment, otherwise - four byte alignment.

Static parameters:

- Offsets for generic PCI configuration space registers.[6]

  static const UCHAR **pciRegID** = 0x0

  static const UCHAR **pciRegCSR** = 0x04

  static const UCHAR **pciRegCLS** = 0x08

  static const UCHAR **pciRegLAT** = 0x0C

  static const UCHAR **pciRegBAR0** = 0x10

  static const UCHAR **pciRegBAR1** = 0x14

  static const UCHAR **pciRegBAR2** = 0x18

  static const UCHAR **pciRegBAR3** = 0x1C

  static const UCHAR **pciRegBAR4** = 0x20

  static const UCHAR **pciRegBAR5** = 0x24

2.4 CMemory

New type definition:

- PCI address space type.

  typedef enum _MEMTYPE {

  undef, // undefined

  mem, // PCI Memory Space accessible only via single transfers

  fmem, // PCI Memory Space accessible via single and burst transfers

  io // PCI I/O Space

  } **MEMTYPE**

Constructor (used for explicit requests for allocation of a region of a PCI address space for a device):

- **CMemory**(MEMTYPE type, DWORD barOffs, DWORD size, DWORD memAlnt)

  where: "type" - PCI address space type requested for device (see the "address space alias" column from the Table 2 for the list of aliases), "barOffs" - offset of the BAR which points to the requested PCI address space region, "size" - requested size of the PCI address space region, "memAlnt" - address space alignment (must be equal to "size" for regular PCI devices). IMPORTANT: If a device driver explicitly requests a region of a PCI address space (by invoking the "CPciDevice::addMemReq" method and the "CMemory" constructor), the software does not scan BARs of device and allocates only the region(s) of PCI address space which was (were) requested by the "CMemory" constructor.

Accessors:

- Returns a string containing memory type alias which corresponds to a literal from the "MEMTYPE" type definition (PCI address space type).

  const char* **getType_s**() const

- Returns a variable of type "MEMTYPE" (PCI address space type).

  const MEMTYPE **getType**() const

- Returns address offset of BAR.

  const DWORD **getBarOffset**() const

- Returns address offset of the address space region base relative to the base of the corresponding VME Slave Image of U2 chip.

  const DWORD **getOffset**() const

- Returns the size of the PCI address space region.

  const DWORD **getSize**() const

- Returns the address alignment of the address space region.

  const DWORD **getAlignment**() const

"eraseFlashSector" method	other "CLclCtl" methods
sectors 0-3	sector 0
sector 4	sector 1
sector 5	sector 2
sector 6	sector 3

Table 3: Correspondence of flash memory sector of "eraseFlashSector" method and other methods of "CLclCtl" class

2.5 CLclCtl

The Local Control FPGA is configured at power-up using data from a first non-empty sector of the on-board flash memory (the flash memory has four sectors). Flash memory access methods:

- Erases a sector of the local control flash memory.

  STATUS **eraseFlashSector**(UCHAR sector) const

  where "sector" can take values 0-6 (the sector numeration scheme of this method is different from one of the other methods of the "CLclCtl" class - see Table 3):

- Erases the whole content of the flash memory.

  STATUS **eraseFlash**() const

- Programs the sector "sector" of the flash memory using the Altera file "dataFileName". The file must be in Altera "hex" format. "sector" can take values from 0 to 3.

  STATUS **programFlashSector**(UCHAR sector, char *dataFileName) const

- Loads the configuration data from the sector "sector" of the flash memory. There are 4 possible configurations stored in the flash (sectors 0-3).

  STATUS **loadConfiguration**(UCHAR sector) const

2.6 CLtb

Miscellaneous methods:

- Fills the array "data" with a test data set of length "nWords" using either the algorithm implemented in the firmware pattern generator of LTB or a random number generator ("genType" = "pattern" or "random" respectively).

STATUS **formPatternArray**(DWORD* data, int nWords, const char* genType) const

- Transmits content of array "data".

  STATUS **directArray**(const DWORD* data) const

2.7 CLrb

Miscellaneous methods:

- Resets and enables all channels of LRB. If "accessMode" is equal to "0" - LRB's FIFO is accessible via single transfers, if "1" - LRB's FIFO is accessible via burst transfers. The argument is optional and is defaulted to "0".

  void **reset**(UCHAR accessMode) const

- Prints out the content of non-zero Configuration Space registers of LRB.

  void **printStatus**() const

- Reads the content of the LRB's FIFOs, checks for bit errors and calculates the number of errors detected for each channel of LRB. Array "data" must contain the data which was sent to LRB. The last three arguments return the number of detected bit errors for channels 0-2, respectively.

  STATUS **testBitErr**(DWORD* data, int& numErr0, int& numErr1, int& numErr2) const

2.8 Examples

This section provides some examples of accessing PCI devices.

```
#include "types.hh"
#include <stdio.h>
#include "crate.hh"
#include "universe2.hh"
#include "memory.hh"
```

```
#include "lcl_ctl.hh"
#include "lrb.hh"
#include "ltb.hh"
/*****
* This main() is used for testing purposes only. Some examples
* of accessing PCI devices are included.
*****/
int main() {

// Detect and initialize MBs and PCI devices from a VME crate
const CCrate c;

// Print out some information about detected PCI devices (for all MBs)
c.pciDeviceStatus();

// Turn on debugging output (for all MBs)
c.setDebugLevel(1)

// Execute device initialization commands from the file "dev_init.txt"
c.execConfigFile("dev_init.txt");

// Start PCI access server (allows remote access to PCI devices via EPICS)
c.pciAccessServer();

/*****************************************
* access PCI Configuration and Memory space registers of the
* Local Control FPGA (bus # 0, device # 4) on the MB from
* slot "slotNo"
*****************************************/
// get pointer to CUniverse2 object which corresponds to MB
// from VME slot number "slotNo"
```

```
UCHAR slotNo = 21;
const CUniverse2* pMB = c.getU2(slotNo);
if (pMB == NULL) {
 printf("MB was not detected in slot # %d\n",slotNo);
 return 1;
}

// get pointer to CPciDevice object which corresponds to the Local
// Control FPGA
const CPciDevice* pLC = pMB->getDevice("lc");
if (pLC == NULL) {
 printf("Local Control was not detected\n");
return 1;
}

// read DWORD (4 bytes) from offset zero of the Local Control's
// Configuration Space (PCI Vendor and Device Ids)
DWORD pciId;
pLC->readDW(0,pciId);
printf("PCI Id of the Local Control: %x\n",pciId);

// read a DWORD from offset zero of the Local Control's memory
DWORD value;
pLC->readDW(0,value,"mem");
printf("First DWORD of the Local Control's memory: %x\n",value);

// write a DWORD to the offset zero of the Local Control's memory
pLC->writeDW(4,value,"mem");

/************************************
* *advanced* stuff: burst write/read CPU <->1 Kby memory
* of the Local Control FPGA. Uses the DMA Channel of
* UniverseII chip of MVME2303 CPU board. Requires a
```

```
* certain configuration of the FPGA.
************************************/
// initialize arrays
DWORD in[256],out[256];
for (int i=0; i<256; i++) {
 in[i] = 0xaaaaaaa000 + i;
 out[i] = 0;
}
// number of DWORDs to write/read in one PCI burst
int nWords = 255;
STATUS stat;
// burst write
stat = pLC->writeBurst(0,&in[0],nWords*4);
printf(”writeBurst status: %s\n”,(stat==OK)?”OK”:”ERROR”);
// burst read
stat = pLC->readBurst(0u,&out[0],nWords*4);
printf(”readBurst status: %s\n”,(stat==OK)?”OK”:”ERROR”);
printf(”First three DWORDS read: %x %x %x\n”,out[0],out[1],out[2]);
*/

/************************************
* *advanced* stuff: erase/program sectors of flash memory
* on the STT motherboards. There are four sectors 64 Kby
* each. The Local Control FPGA is configured at power-up
* using data from the first non-empty sector. The FPGA can
* also be reconfigured (soft reset) using data from a particular
* sector of the flash memory.
************************************/
// use pointer type cast to allow access to the device-specific methods
// of class ”CLclCtl” (”CLclCtl” is a derived class of ”CPciDevice”)
CLclCtl* pLC_c = (CLclCtl*)pLC;
```

```
// erase the whole flash memory
pLC_c->eraseFlash();
// program sectors 0-3 of the flash memory using hex Altera files
pLCc->programFlashSector(0,"l_ctl_flash.hex");
pLCc->programFlashSector(1,"l_ctl_fl.hex");
pLCc->programFlashSector(2,"l_ctl_flash.hex");
pLCc->programFlashSector(3,"l_ctl_fl.hex");
// configure FPGA using data from the sector 3 of the flash memory
pLCc_c->loadConfiguration(3);

/**************************************
* Loop over all detected motherboards and all detected
* PCI devices and print out some information about the
* devices and the memory allocated for them.
**************************************/
printf("\nMB Alias PciId Rev# Bus# Dev# MemType "
"MemOffs MemSize MemBar\n");
// loop over detected MBs
for ( const CUniverse2* pMbI = c.getFirstU2();
            pMbI != NULL; pMbI = c.getNextU2(pMbI) ) {
// loop over detected PCI devices from the current MB
for ( const CPciDevice* pDevI = pMbI->getFirstDev();
   pDevI != NULL; pDevI = pMbI->getNextDev(pDevI) ) {
// read PCI device and vendor Ids (PCI Id of the device)
DWORD pciId;
pDevI->readDW(0,pciId);
// print out slot number of MB and some info on the current device
printf("%2d %5s %8x %2d %1d %1d",pMbI->getSlotNo(),
pDevI->getDevAlias(),pDevI->getPciInfo(), pDevI->getRevisionNo(),
pDevI->getBusNo(),pDevI->getDevNo());
// loop over allocated memory objects of the device and print out
```

```
// some parameters of the memory
UCHAR count = 0;
for ( const CMemory* pMemI = pDevI->getFirstMem();
    pMemI != NULL; pMemI = pDevI->getNextMem(pMemI) ) {
// print out only memory of non zero size
if (pMemI->getSize() == 0)
continue;
// get pointer to the corresponding VME Slave Image (VSI) of U2
CU2image* vsi = pMbI->univ_get_image(pMemI->getType_s());
// check if vsi exists
if (vsi == NULL) {
printf(”main: Error: VSI of type %s is not found\n”,pMemI->getType_s());
continue;
}
// calculate the PCI base address of the memory range allocated
// for device
DWORD pciAddr = vsi->vmeBase + vsi->to + pMemI->getOffset();
// print out some memory parameters
if (count == 0)
printf(” %5s %8x %8x %4s\n”,pMemI->getType_s(),pciAddr,
pMemI->getSize(), (pMemI->getBarOffset()==0x10)?”bar0”:”bar1..”);
else {
printf(” –//– –//– ”); printf(” %5s %8x %8x %4s\n”,
pMemI->getType_s(),pciAddr,pMemI->getSize(),
(pMemI->getBarOffset()==0x10)?”bar0”:”bar1”);
}
count++;
}
if (count == 0)
printf(”\n”);
}
```

```
}

/********************************************
* Test CPU->LTB->LRB->CPU chain. CPU sends some data
* over PCI to LTB, LTB transmits data to LRB over a cable,
* CPU reads data from LRB over PCI.
********************************************/
// get pointer to LTB object which corresponds to LTB
// in "MB6" PC-MIP site
const CLtb* pLTB = (CLtb*)pMB->getDevice("mip6");
if (pLTB == NULL) {
printf("LTB was not detected in site MB6\n");
return 1;
}

// get pointer to LRB object which corresponds to LRB
// in "MB5" PC-MIP site
const CLrb* pLRB = (CLrb*)pMB->getDevice("mip5");
if (pLRB == NULL) {
printf("LRB was not detected in site MB5\n");
return 1;
}
// fill the "testData" array with random numbers
nWords = 256;
DWORD testData[nWords];
if (pLTB->formPatternArray(&testData[0], nWords, "random") ==
ERROR) {
printf("formPatternArray returns ERROR\n");
return 1;
}
// reset and enable all channels of LRB
pLRB->reset();
```

```
// print out the content of some configuration registers of LRB
pLRB->printStatus();
// Transmit data LTB to LRB ( the data is received by LTB
// from CPU over PCI)
pLTB->directArray(&testData[0], nWords);
// print out the content of some configuration registers of LRB
pLRB->printStatus();
// check for transfer errors
int ber0 = 0, ber1 = 0, ber2 = 0;
pLRB->testBitErr(&testData[0], nWords, ber0, ber1, ber2);
printf("Number of bit errors detected for ch0: %d ch1: "
    "%d and ch2: %d\n",ber0, ber1, ber2);
// print out the content of some configuration registers of LRB
pLRB->printStatus();

return 0;
}
```

3 PCI Access Client

The Client allows interactive access to PCI devices on a MB.

3.1 Setup

The following conditions have to be met for the Client to run properly:

- The Client's host must have Python installed (version 1.52 or newer).
- The Client's host must have a network access to the VME IOC (the VME IOC should be accessible via "ping" or "telnet" from the Client's host).
- Module mbsp_lib.o must be loaded to the memory of the VME IOC and the "pciAccServer" function must be started after that (see provided start-up script for VxWorks available from [5]).

3.2 Description

The Python "pciAccess" class allows to communicate with the "CCrate::pciAccessServer()" method which runs on the VME IOC. The following methods of "pciAccess" are available:

- Constructor (it establishes connection to the "CCrate::pciAccessServer()" via a network socket).

  pciAccess(hostName, portNo)

  where "hostName" is a string containing the IP address of the VME IOC, "portNo" is the network port number used. The second parameter is optional, it is defaulted to "40404".

- Turns on/off debugging output on the console terminal of the VME IOC.

  debug(command, mbSlotNo)

  where "command" can be "on" or "off" (quotes can be omitted), "mbSlotNo" - VME slot number of a MB. The second argument is optional, if it is omitted the function applies to all MBs.

- Prints out status information about detected PCI devices on a MB from slot "mbSlotNo".

  status(mbSlotNo)

  The argument is optional, if it is omitted the function prints out the status information for all MBs.

- Device access methods (single transfers):

  read(mbSlotNo, devAlias, offset, memType, BAR, funcNo) - reads data

  write(mbSlotNo, devAlias, offset, data, memType, BAR, funcNo) - writes data

  r(mbSlotNo, devAlias, offset, memType, BAR, funcNo) - reads data

  w(mbSlotNo, devAlias, offset, data, memType, BAR, funcNo) - writes data

  where "mbSlotNo" - VME slot number of a MB, "devAlias" - alias of the geographical position of a device (see Table 1 for list of aliases), "offset" - address offset relative to the beginning of the address space region of device, "memType" - alias of PCI address space type (see Table 2 for the list of aliases), "BAR" - offset of BAR which points to the requested PCI address space region (symbols "bar0, ..., "bar5" can be used; quotes can be omitted), "funcNo" - PCI Function Number (used for PCI Configuration Space access only). Last three arguments are optional: "memType" is defaulted to "conf", "BAR" - to "bar0" and "funcNo" - to zero. Method "r" prints out the received data using hexadecimal format (the method does not return anything). Method "read" returns received data. Method "w" is an alias for method "write". The methods access PCI devices by using the "CPciDevice" class from the C++ part of the library (that is a device should be detected by the VME IOC software for these methods to work properly. If that's not the case, one can use methods "rcf" and "wcf" to access the PCI Configuration space of devices directly).

- PCI Configuration Space access methods (the access is performed using the "CUniverse2" class directly so a "CPciDevice" object does not have to be instantiated for a device):

  rcs(mbSlotNo, busNo, devNo, offset, funcNo) - reads data

  wcs(mbSlotNo, busNo, devNo, offset, data, funcNo) - writes data

  where "mbSlotNo" - VME slot number of a MB, "busNo" and "devNo" - PCI bus and device numbers of device, "offset" - address offset relative to the base of the Configuration Space of device, "funcNo" - PCI function number. The last argument is optional and is defaulted to zero. Method "rcs" prints out the received data in hexadecimal format (it does not return anything).

- Sets destination for device access:

  setDestination(mbSlotNo, devAlias, memType, BAR, funcNo)

  where arguments are similar to ones defined for "read" and "write" methods.

- Device access methods (single transfers):

  fr(offset) - reads data

  fw(offset, data) - writes data

  Method "fr" returns data received from device. Method "setDestination" must be invoked before using a group of calls of the "fr"/"fw" methods.

- Device access methods (burst transfers):

  burstRead(mbSlotNo, devAlias, offset, dataSize, BAR) - reads data

burstWrite(mbSlotNo, devAlias, offset, data, dataSize, BAR) - writes data

where "mbSlotNo", "devAlias", "offset" and BAR are defined in the description of the "read" method. "dataSize" is the size of the requested burst transfer in DWORDs (1 DWORD = four bytes). "data" is a Python list containing at least "dataSize" number of words to be written. "burstRead" method returns a list containing "dataSize" number of words. Note that an accessed device must have an allocated PCI memory region of type "fmem".

- Data file parser method:

  execAdpFile(mbSlotNo, devAlias,fileName,memType,BAR,funcNo)

  This method allows to write data from the file "fileName" to a PCI device. The file must be accessible by the VME IOC. Most of the arguments of the method are similar to the ones defined in the description of the "write" method. The last three arguments are optional and defaulted to "mem", "bar0' and "0", respectively. "fileName" is the path/name of the file containing address offset/data pairs. The address offset is taken to be the offset with the respect of the base of the allocate region of PCI address space of device as it specified by the method arguments. The file must contain one offset/data word pair per line (offset word is first, data is second). The offset and data words must be in hex format and separated by at least one empty/tab space.

- Data file parser method. This method allows to write data from a file to a PCI device.

  execIHexFile(mbSlotNo,devAlias,fileName,byteFlag,memType,barOffset,funcNo)

  This method allows to write data from the file "fileName" to a PCI device. The file must be accessible by the VME IOC. The arguments are similar to the ones defined in the description of the previous method. "fileName" is the path/name of a file in the Intel Hex (IHex) format.[7]. "byteFlag" specifies the IHex file parsing mode. If "byteFlag" is "1" the data is parsed using one byte alignment, if "0" - four byte alignment.

- **This method is disabled in the current version.** Device Configuration file parser method:

  execConfigFile(fileName)

  This method allows to initialize a PCI device using commands from file "fileName". The format of the file is described in Appendix A.

"fileName" is the path/name of a Configuration file. The file must be accessible by the VME IOC.

- Programs a sector of flash memory of the Local Control FPGA using data from a file:

  progLcFlash(mbSlotNo, fileName, sectorNo)

  where "mbSlotNo" is VME slot number of a MB, "fileName" is the path/name of the file containing PCI Core data in the IHex format, "sectorNo" is the sector number of the flash memory (0-3). The file must be accessible by the VME IOC.

- Configures the Local Control FPGA using data from the flash memory:

  loadLcConf(mbSlotNo, sectorNo)

  Parameters are defined in the description of the previous method.

- Prints the VME and PCI memory maps which correspond to MB in slot number "slotNo":

  printMemMap(mbSlotNo)

  The output format is described in the section about the "CCrate::printMemMap" method. The input parameter is optional and if it is omitted the methods applies to all MBs.

- Reads/write from/to a VME address:

  rvme(addr, vas, endianessSwapFlag)

  wvme(addr, data, vas, endianessSwapFlag)

  Input parameters are described in the section about the "CCrate::readDW" and "CCrate::writeDW" methods. "endianessSwapFlag" can take values "0" and "1" which correspond to "false" and "true". The last two parameters are optional and are both defaulted to "1".

- Resets PCI bus on the MB in the slot number "mbSlotNo":

  resetPci(mbSlotNo)

  The input parameter is optional and if it is omitted the method applies to all detected MBs.

3.3 Examples

An example of Client session is shown in this Section. Lines starting with "#" are comments, lines starting with ">>>" are python commands, and lines starting with "0x" show the output of the command from a previous line):

```
# import client module
>>>from client import *
# create an instance of class pciAccess
>>>a=pciAccess('quark.bu.edu')
# print out device status information (for all MBs)
a.status()
# turn on debugging output (for all MBs)
a.debug(on)
# read from the Configuration Space at offset 0 (PCI Id of device) of
PCI-3 interface of STC (STC3) of MB at slot 21
>>>a.r(21, log3, 0)
0x30060
# read from the Memory Space (BAR0) at offset 0 of STC3 in MB at
slot 21
>>>a.r(21, log3, 0, mem)
0x0
# write 0x12345678 to Memory Space (BAR0) at offset 0x400 of STC3
of MB at slot 21
>>>a.w(21, log3, 0x400, 0x12345678, mem)
# read from the Memory Space (BAR0) at offset 0x400 of STC3 of
MB at slot 21
>>>a.r(21, log3, 0x400, mem)
0x12345678
# read from the Memory Space (BAR1) at offset 0 of STC3 of MB at
slot 21
>>>a.r(21, log3, 0, mem, bar1)
0xffffffff
```

set the Memory Space (BAR0) of Local Control at slot 21 MB as a destination

>>>a.setDestination(21, lc, mem, bar0)

read from the Memory Space at offset 0 of the Local Control

>>>'0x%x'%a.fr(0)

0x0

write 0x12345678 to the Memory Space at offset 0 of the Local Control

>>>a.fw(0, 0x12345678)

read from the Configuration Space at offset 0x10 of the Local Control (PCI bus and device number 0 and 4 respectively) of slot 21 MB using "direct" UniverseII access method

>>>a.csr(21, 0, 4, 0x10)

write 0x1234000 to the Configuration Space at offset 0x10 of the Local Control (PCI bus and device number 0 and 4 respectively) of slot 21 MB using "direct" UniverseII access method

>>>a.csw(21, 0, 4, 0x10, 0x1234000)

4 Appendix A: Format of the PCI Device Configuration File

A PCI device can be initialized by invoking the **CCrate::execConfigFile(const char* fileName)** method, where "fileName" is the path/name of a configuration file. The configuration file must be accessible by the VME IOC. The Python Client also allows to invoke that method by using the corresponding method of the "pciAccess" class.

A Configuration file is a text file which consists of a set of commands described below. If a literal from a Configuration file represents a number, it is parsed as a hexadecimal number if it has "0x" as a prefix, and as the decimal number if it does not have such a prefix.

- Comment: a line which start with symbol "#". Example:

  # This is a comment

- Definition of a variable:

define NAME_OF_VARIABLE ”value”

where ”value” is a string. A defined variable can be used throughout the configuration file (including the definition of another variable) by using the name of the variable with ”$” as a prefix. The following C language style bitwise operators are supported: ”&”, ”|”, ”^”, ”<<” and ”>>”. Examples:

define DEVICE ”21 log1”

define ADDRESS ”0xa000”

define DATA1 ”0xFB0”

define DATA2 ”2”

define DATA3 ”$DATA1|$DATA2”

define DATA4 ”1<<5”

define INIT ”$DEVICE $ADDRESS $DATA3|$DATA4”

- Write command (”long” format):

  write slot=slotNo device=devAlias offset=Offs data=Data memory=memType bar=Bar

  The format of the parameters is the same as the one used in the ”pciAccess::write” method of the client. The last two parameters are optional and defaulted to ”conf” and ”bar0”, respectively. Examples:

  write slot=21 device=lc offset=0xb00 data=255 memory=io bar=bar1

  write slot=21 device=log3 offset=$ADDRESS data=$DATA3|$DATA4 memory=mem

  write $DEVICE offset=0 data=1

- Write command (”short” format):

  w slotNo devAlias Offs Data memType Bar

  The arguments are the same as the ones of the previous command. Examples:

  w 21 log3 0xb00 255 mem bar0

  w 21 lc $ADDRESS $DATA3|$DATA4

- Destination block (”long” format):

destination slot=slotNo device=devAlias mem=memType bar=Bar

offset=Offs1 data=Data1

offset=Offs2 data=Data2

...

end

The Destination block structure allows to write a set of data to a particular region of PCI Address Space region allocated for a PCI device as it is specified by the first line of a Destination block. The format of the parameters is the similar to the one used by the "write" command. The last two parameters of the first line of a Destination block are optional and are defaulted to "conf" and "bar0", respectively. Example:

destination slot=21 device=log3 memory=mem

offset=0x10 data=0xffff

offset=255 data=16

offset=$ADDRESS data=$DATA1

end

- Destination block ("short" format):

  dest slotNo devAlias memType Bar

  Addr1 Data1

  Addr2 Data2

  ...

  end

  The format of the parameters is the same as the one of the previous command. Example:

  dest 5 mip1

  0x38 0x1c1

  0x38 0x1c0

  end

- Configuration file load command:

load "fileName"

where "fileName" is the path/name of a Configuration file. The "load" command allows to execute commands from the Configuration file "fileName". The file must be accessible by the VME IOC. Example:

load "/home/user/pci_config/stc_init.dat"

- "C language header" file load command:

  load_C "fileName"

  where "fileName" is the path/name of a "C language header" file. The file must be accessible by the VME IOC. Example:

  load_C "/home/user/mbsp/src/lrb_symb.hh"

  A "C language header" file can contain the C language preprocessor "#define" command. A line which does not start with the "#define" string is ignored. The following C language style bitwise operators are supported: "&", "|", "^", "<<" and ">>". The bitwise operator precedence is "from left to right". Parentheses are ignored and nested parenthesis are not allowed. Defined variables cannot be a part of a "#define" statement. Here is an example of a "C language header" file:

  #define LRB_CC_ENA_0 (1<<16) // enable channel 0

  #define LRB_CC_ENA_1 (1<<17) /* enable channel 1 */

  #define LRB_CC_ENA_2 (1<<18) // enable channel 2

  #define TEST (0x10|0x100)

- "Intel Hex" file load command:

  load_IHex "fileName"

  where "fileName" is the path/name of a file which contains offset/data in the Intel Hex format.[7] The file must be accessible by the VME IOC. The "load_IHex" command must be always invoked within the body of a Destination block. The IHex file data is written to the allocated region of a PCI address space of a device which is specified in the first line of the corresponding Destination block. Example:

  dest 21 log3 mem

load_IHex "/home/user/pci_config/stc_lookup_table.216.hex"

end

A Python script "adp2ihex.py", which converts a text file containing offset/data pairs to the corresponding file in the IHex format, is provided. It is available d0cvs server as a part of the "stt_mbsp" package or from [5] (subdirectory "client"). The script can be invoked from a command line and it takes two arguments: the path/name of an input text file containing offset/data pairs, and the path/name of the output IHex file. The input file must contain one pair of the offset and data words per line. The offset and data words must be separated by at least one empty space. The offset and data words are considered to be in the decimal format if they do not contain the "0x" prefix, and in the hexadecimal format if they contain the prefix. Here is an example of the input file:

0x0 0x12345678

4 0x9abcdef0

0x8 255

Here is the corresponding output IHex file (the second line corresponds to the end of the file):

:09000000123456789ABCDEF0000000FFC0

:00000001FF

References

[1] http://ohm.bu.edu/~hazen/my_d0/mb9u/

[2] http://focus.ti.com/docs/prod/productfolder.jhtml?genericPartNumber =PCI2031

[3] http://www.tundra.com/page.cfm?TREE_ID=100361

[4] http://d0server1.fnal.gov/www/online_computing/projects/controls/ epics_python/epics_python.html

[5] http://physics.bu.edu/~popkov/stt/mb/index.html

[6] PCI Local Bus Specification 2.2

[7] http://www.filespecs.com/formats/encoding/hex/content.jsp

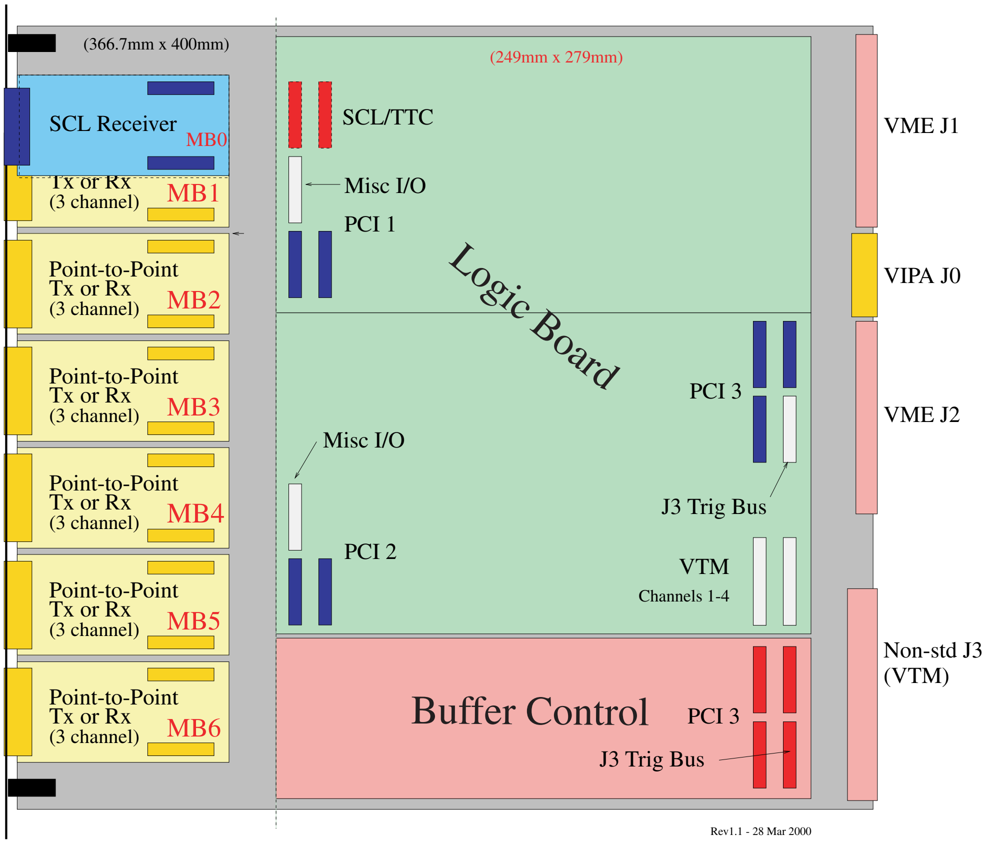

Figure 1: Layout of the motherboard