6 Nov 2011 /TBS

FYS4220 / 9220

RT-lab no 3 - 2011

MIDAS M5000 SBC and VME

1 Scope of the exercise

This purpose of RT-lab 3 twofold:

1) A first order introduction to a Single Board Computer, the MIDAS M5000;
2) An example of communication with a VME module from the M5000.

For FYS9220 students an additional task is added, see end of document.

1.1 MIDAS M5000

The M5000 is a Single Board Computer (SBC) in VME format with the PPC440 processor. The MIDAS M5000 family is a product from the company VMETRO for the professional signal processing market, in particular military, which is also reflected by the price. VMETRO was bought in 2009 by the US Company Curtiss Wright[1] . See also http://www.cwcembedded.com

Figure 1 The MIDAS M5000 – VME version. Two PMCs can be mounted on the 2x4 PCI connectors.

[1] Yes, the name Wright is related to the brothers who invented and build the world's first successful airplane and making the first controlled, powered and sustained heavier-than-air human flight, on December 17, 1903

1.1 Main features:

- A PowerPC processor subsystem (440GX from IBM) with up to 256MiB local DDR-SDRAM memory, 32MiB of FLASH memory, four Ethernet ports (two 10/100/1000Mbps and two 10/100Mbps) and two serial ports.
- Two standard PPMC sites located on separate PCI segments (64-bit, 33/66MHz PCI, 66/100/133MHz PCI-X). In addition, PMC#1 can be MONARCH
- Dual 2Gib Fibre Channel I/O Controller (ISP2312 from QLOGIC) (Optical)
- Single 10/100 Ethernet port (front panel connector)
- Dual serial ports configurable as two RS-232 ports or one RS-232 and one RS-422
- PCI-to-VME bridge (Universe IID from Tundra)
- Three PCI(X)-to-PCI(X) bridges which connects the different PCI(-X) segments together (PCI6540 from PLX)
- Optional PCI-to-RACEway bridge (PXB++ from Mercury)
- Optional I/O Spacer extension via a built-in connector, thus adding up to 3 Ethernet ports (One Fast Ethernet 10/100 SMII and two Gigabit Ethernet 10/100/1000 RGMII) and an I2C bus. Note: Adding more than one Gigabit Ethernet port will reduce the number of Fibre Channel ports
- Optional mezzanine extension via a built-in connector, thus adding 3 PMC sites to the Quaternary PCI segment. These PMC sites are 64-bit, 33MHz PCI, 5V signaling.
- One PIM I/O board slot, in accordance to VITA36.

FIGURE 1-1 *Component Overview*

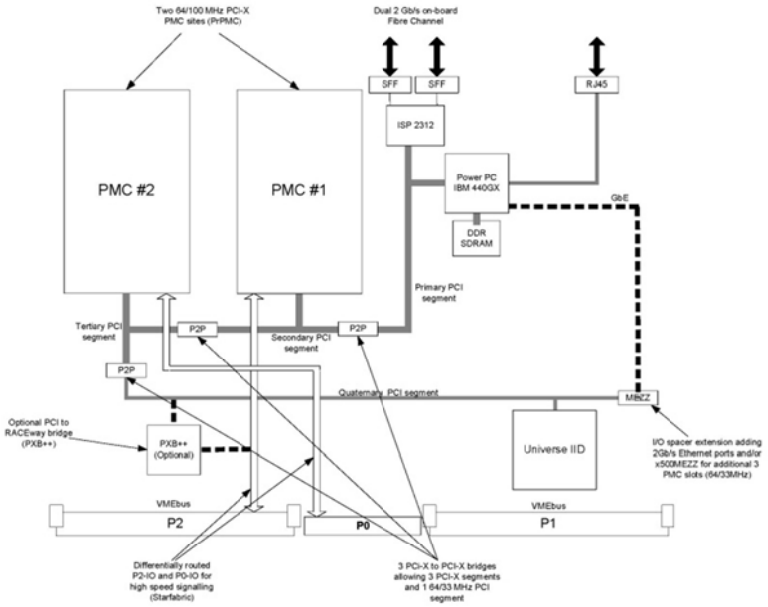


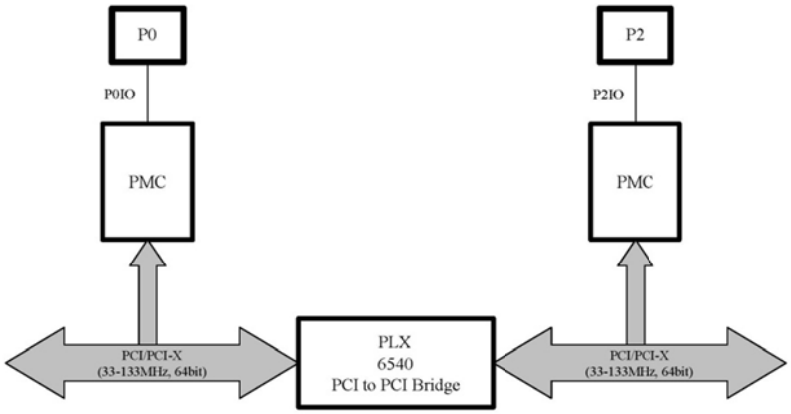
Figure 2 MIDAS M5000 main features

The on-board interconnection busses are via four PCI segments. The Universe chip bridges one PCI bus to VMEbus.

4.1 Introduction

The PMC subsystem is composed of two 3.3V signalling PMC sites and is located on two PCI(-X) segments called the Secondary and Tertiary PCI Segments. The connection between the two PCI(-X) segments is provided by a PCI-X to PCI-X bridge.

FIGURE 4-1 *PMC Subsystem Block Diagram*



The factory settings of the board switch configuration allows the bus frequency of the PCI-X segments to be automatically configured to the frequency used by a PMC card insterted into it, and by the limitation set with dip-switches 1 and 4. See "PCI-X Capability Selection for PMC Slots" on page 17.

Figure 3 MIDAS M5000 PMC system

1.1.1 The Universe PCI-to-VMEbus Bridge

The block diagram is shown in Figure 4 and the data flow diagram in Figure 5.

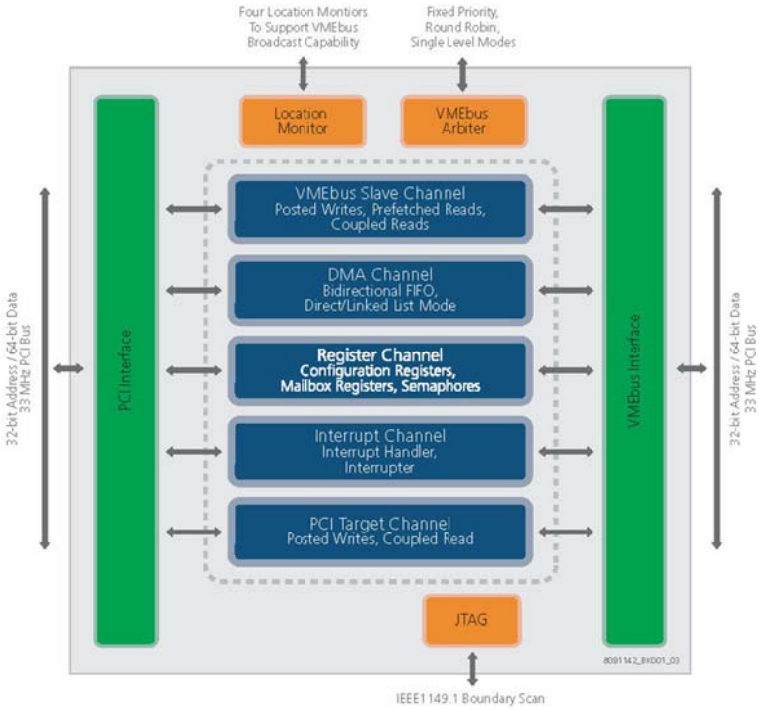


Figure 4 Block diagram of Universe PCI-VMEbus bridge

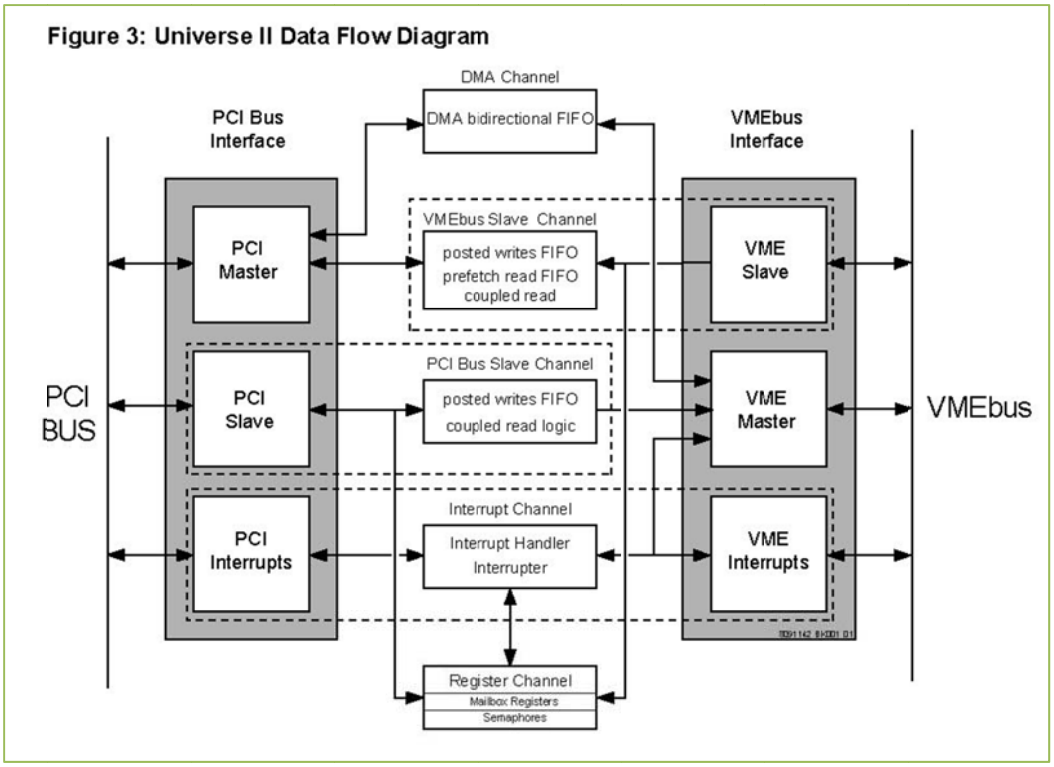


Figure 5 Data Flow diagram

Altogether 8 VME Slave images and 8 PCI slave images can be defined. Each VME Slave image opens a window to the resources of the PCI bus and, through its specific attributes, allows the user to control the type of access to those resources. Correspondingly, a PCI Slave image opens a window for VMEbus Master operations.

For more information on the Universe registers it is referred to the documentation.

2 *The VxWorks BSP for MIDAS M5000*

The M5000 BSP uses the PPC440 MMU (Memory Management Unit) for virtual addressing. The address map is shown in Figure 6 .

At boot time some default mapping is set up; see Figure 7 (from RTlab-1).

The BSP contains a large number of system calls, only those relevant for VME access is commented here. For a deep pleasure of understanding the complete spectrum of BSP functions it is referred to the documentation.

1.3 M5xxx Address Maps and Address Space Mapping

The address map layouts (CPU and PCI) for the M5xxx BSP implementation are as follows. These maps are shown as supported with the default PCI auto-configuration. Manual PCI configuration is not currently supported by the M5xxx BSP. A detailed look at PCI address space assignment is given in the section of PCI bus layout.

TABLE 1-2. M5xxx BSP 32-bit Effective (Virtual) Address Map with PCI Auto-config (default)

Address range	Resource Mapped	Mapped by
0x00000000-0x0dffffff[b]	Cached System SDRAM access	MMU TLB Entry
0x0e000000-0x0fffffff[ab]	Non-cached System SDRAM access	MMU TLB Entry
0x10000000-0xbfffffff[c]	PCI outbound translation window	MMU TLB Entry with prefetch
0xc0000000-0xefffffff[c]	PCI outbound translation window	MMU TLB Entry without prefetch
0xf0000000-0xf0ffffff	Internal CPU Peripherals	MMU TLB Entry
0xf1000000-0xf1ffffff	I2O	MMU TLB Entry
0xf2000000-0xf2ffffff	SRAM	MMU TLB Entry
0xf3000000-0xf4ffffff	FLASH memory (cached)	MMU TLB Entry
0xf5000000-0xf5ffffff	PLD	MMU TLB Entry
0xf8000000-0xfbffffff	PCI I/O outbound	MMU TLB Entry
0xfc000000-0xfdffffff	Not mapped-available	-
0xfd000000-0xfdffffff	PCI-X bridge	MMU TLB Entry
0xfe000000-0xffffffff	FLASH memory (non-cached)	MMU TLBEntry

a. The cached and non-cached regions access the same physical SDRAM.
b. User configurable through NONCACHEABLE_MEMORY_SIZE
c. User configurable through PCI_MASTER_PREFETCHABLE_POOL_SIZE

Regions marked "Not mapped--available" can provide addressing to PCIbus resources. To enable access to these regions, the PPC440GX MMU must be initialized appropriately. This is done by adding entries to the sysStaticTlbDesc[] array found in sysLib.c. See the sysStaticTlbDesc[] array in "sysLib.c" for more details.

Figure 6 M5xxx Address Map

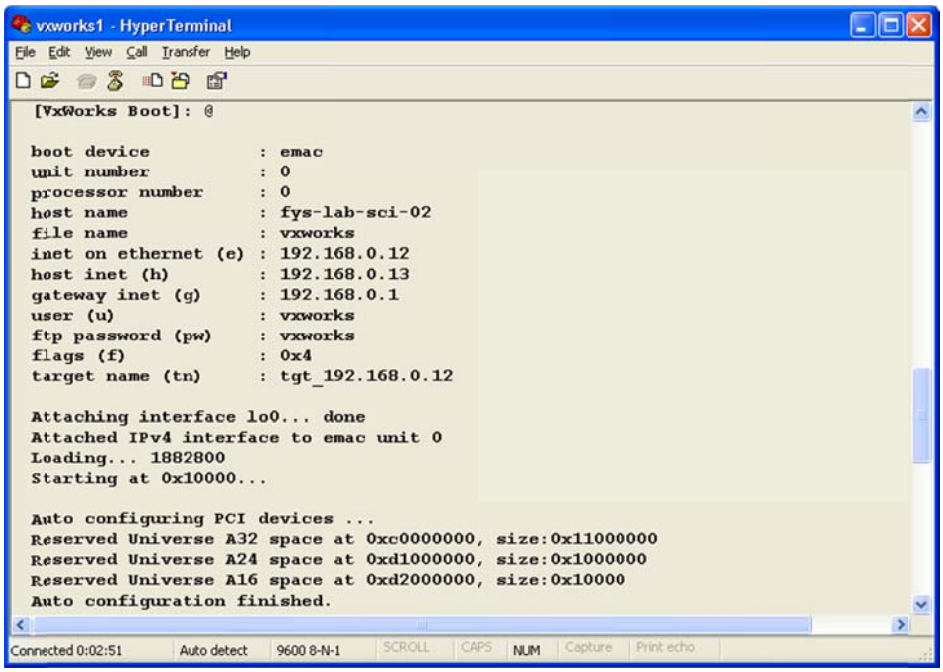
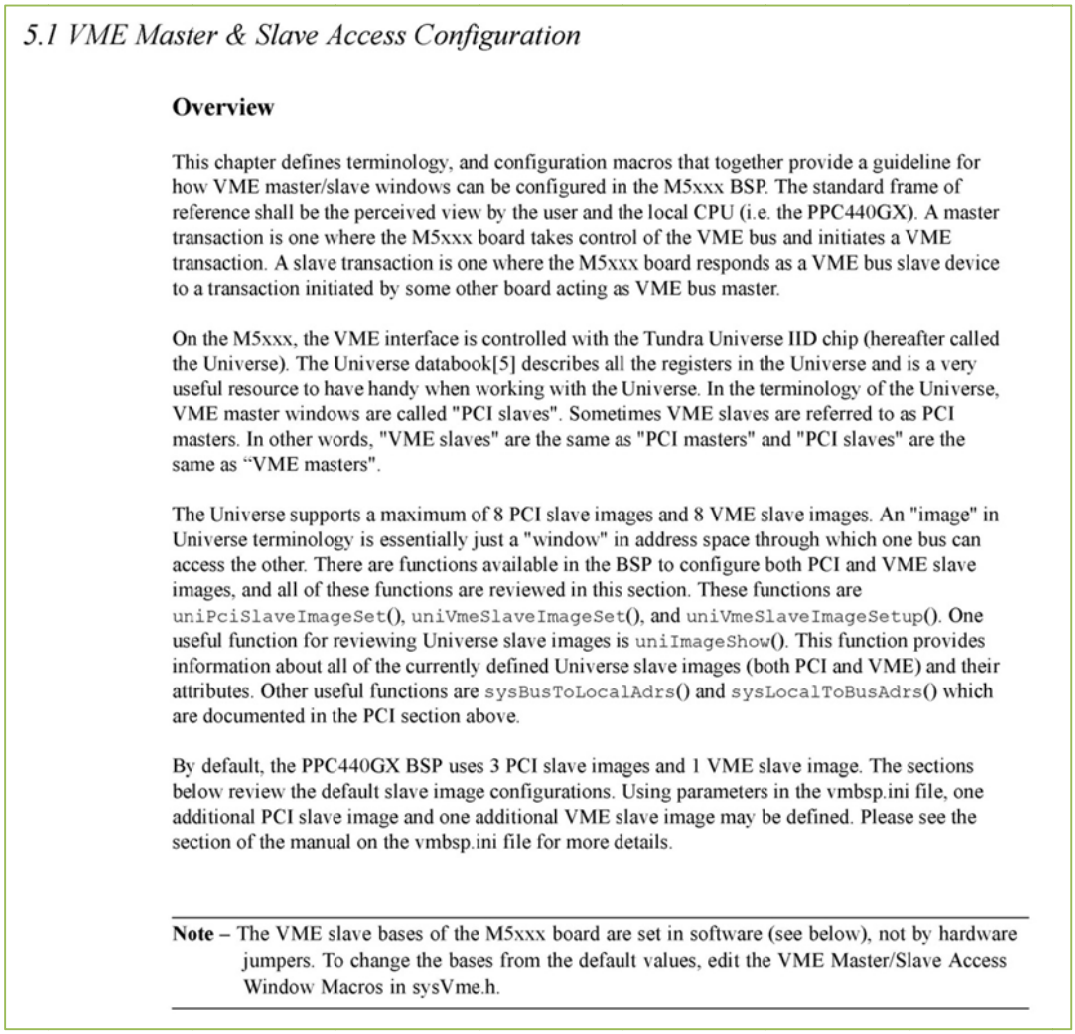


Figure 7 Universe bridge VME A32/A23/A16 configuring at boot time

An overview of the access of VME address space is shown in Figure 8**.**



5.1 VME Master & Slave Access Configuration

Overview

This chapter defines terminology, and configuration macros that together provide a guideline for how VME master/slave windows can be configured in the M5xxx BSP. The standard frame of reference shall be the perceived view by the user and the local CPU (i.e. the PPC440GX). A master transaction is one where the M5xxx board takes control of the VME bus and initiates a VME transaction. A slave transaction is one where the M5xxx board responds as a VME bus slave device to a transaction initiated by some other board acting as VME bus master.

On the M5xxx, the VME interface is controlled with the Tundra Universe IID chip (hereafter called the Universe). The Universe databook[5] describes all the registers in the Universe and is a very useful resource to have handy when working with the Universe. In the terminology of the Universe, VME master windows are called "PCI slaves". Sometimes VME slaves are referred to as PCI masters. In other words, "VME slaves" are the same as "PCI masters" and "PCI slaves" are the same as "VME masters".

The Universe supports a maximum of 8 PCI slave images and 8 VME slave images. An "image" in Universe terminology is essentially just a "window" in address space through which one bus can access the other. There are functions available in the BSP to configure both PCI and VME slave images, and all of these functions are reviewed in this section. These functions are `uniPciSlaveImageSet()`, `uniVmeSlaveImageSet()`, and `uniVmeSlaveImageSetup()`. One useful function for reviewing Universe slave images is `uniImageShow()`. This function provides information about all of the currently defined Universe slave images (both PCI and VME) and their attributes. Other useful functions are `sysBusToLocalAdrs()` and `sysLocalToBusAdrs()` which are documented in the PCI section above.

By default, the PPC440GX BSP uses 3 PCI slave images and 1 VME slave image. The sections below review the default slave image configurations. Using parameters in the vmbsp.ini file, one additional PCI slave image and one additional VME slave image may be defined. Please see the section of the manual on the vmbsp.ini file for more details.

Note – The VME slave bases of the M5xxx board are set in software (see below), not by hardware jumpers. To change the bases from the default values, edit the VME Master/Slave Access Window Macros in sysVme.h.

Figure 8 VME Access Configuration

To access a VMEbus location there are three actions to be taken:

1. Set up a PCI Slave image with **uniPciSlaveImageSet** , Figure 9.
2. Map to the VME address space with **sysBusToLocalAdrs** , Figure 10.
3. By now one shall have a pointer to the VME base address, and a VME address is accessed by a pointer operation.

In other words, a piece of cake!

Instead of accessing individual VME addresses one can set up a DMA transfer. By using the linked list feature of Universe a sequence of DMA transfers can be defined.

uniPciSlaveImageSet

Synopsis

```
STATUS uniPciSlaveImageSet
        (
         int    image,
         UINT32 pciBase,
         UINT32 vmeBase,
         UINT32 size,
         UINT32 pciAddrSpace,
         UINT32 vmeAmCode,
         UINT32 vmeDataWidth,
         BOOL   postedWrites
         )
```

`image` - the Universe PCI slave image number, from 0 - 7

`pciBase` - the PCI base address of the window

`vmeBase` - the VME base address of the window

`size` - the size of the window in bytes

`pciAddrSpace` - the PCI address space. The value can be either `UNI_PCI_MEMORY_SPACE (0)`, `UNI_PCI_IO_SPACE (1)`, or `UNI_PCI_CFG_SPACE (2)`.

`vmeAmCode` - the VME AM code; specifying a "block" type AM code also implies that the similar AM code corresponding to "single" cycles will also be supported by the window.

`vmeDataWidth` - the data width supported by the window. The value can be either `UNI_VMEBUS_DATAWIDTH_8 (0)`, `UNI_VMEBUS_DATAWIDTH_16 (1)`, `UNI_VMEBUS_DATAWIDTH_32 (2)`, or `UNI_VMEBUS_DATAWIDTH_64 (3)`

`postedWrites` - whether the window allows posted (cached) writes. The value should be either TRUE (1) or FALSE (0).

Description This function is used to configure a Universe PCI slave image. This allows the M5xxx to act as a VME master and read/write to other VMEbus devices configured as VME slaves.

Figure 9 Setting up a PCI slave image for VME access

sysBusToLocalAdrs

Synopsis

```
STATUS sysBusToLocalAdrs (
int adrsSpace,
char *busAdrs,
char **pLocalAdrs
)
```

`adrsSpace` - Represents the bus address space in which busAdrs resides. The value can be one of the following:`PCI_SPACE_IO_PRI (0x40)` - 32-bit PCI I/O Space
`PCI_SPACE_MEMIO_PRI (0x41)` - Non-cacheable PCI Memory Space
`PCI_SPACE_MEM_PRI (0x42)` - Cacheable PCI Memory Space
`PCI_SPACE_IO16_PRI (0x43)` - 16-bit PCI I/O Space
A supported VMEbus AM code (see section on VMEbus).

`busAdrs` - the bus address to be converted to a local address

`pLocalAdrs` - holds the returned local address equivalent of the busAdrs if it exists

Description This function converts a bus address to a local address.The function can be used with both PCI and VME address spaces. If the given bus address can be converted to a local address, the local address is placed in pLocalAdrs and the function returns OK. Otherwise, ERROR is returned. Note that an adrsSpace value of PCI_SPACE_CFG_PRI is not supported. In other words, sysBusToLocalAdrs() cannot be used to determine the local address space equivalent for PCI Config Space because there is no such direct address mapping between local and PCI configuration space.

Returns `OK`, or `ERROR`.

Figure 10 Mapping to a VME bus address

How does one specify the actual PCI Slave Image for **sysBusToLocalAdrs** ? Good question, because it is not described in the documentation as far as I can see.

The answer is that the routine selects the first Slave Image, starting from no. 0, which matches the VMEbus AM code! Therefore one must configure a PCI Slave Image if none of the default settings do not match. From the printout in RTlab-3 one observes that PCI Slave Images 0, 1 and 2 are set up at boot time with data width D32 and AM (Address Modifier) code 2d, 3d and 0d, respectively.

3 RTlab-3.c

The source code is written for a module (TSVME) which is an interface for the HP-IB instrumentation bus. However, what is relevant for this exercise is that the module contains two memory blocks, a ROM and a RAM.

The base address bits 23-16 is set up by switches to 0xF00000.

The code should be self-explanatory. The module is accessed by A24 and D8.

Disclaimer: do not use this source; download it from the FYS4220 web.

```
/* RTlab-3.c  - FYS4220 2011 */
/* B. Skaali copyright */

#define VXWORKS
#define M5000

#include   <vxworks.h>
#include   <taskLib.h>
#include   <pci.h>
#include   <sysLib.h>
#include   <MidasPciLib.h>
#include   <pciConfigLib.h>
#include   <uniLib.h>
#include   <sysVme.h>
#include   <vme.h>
#include   <uniDmaLib.h>
#include   "stdio.h"

void        uniImageShow();

/* Pci slave image 1 default mapping */
UINT32              pciBase1d = 0xD1000000,
                    vmeBase1d = 0x00000000,
                    size1d =    0x01000000,
                    pciAddrSpace1d = UNI_PCI_MEMORY_SPACE,
                    vmeAmCode1d = VME_AM_STD_SUP_DATA,
                    vmeDataWidth1d = UNI_VMEBUS_DATAWIDTH_32;
BOOL      postedWrites1d = TRUE;

typedef struct VME_PCISLAVE_INFO {
  UINT       pci_base_adr;
  UINT       vme_base_adr;
  UINT       vme_size;
  UINT       pci_addr_space;
  UINT       vme_am_code;
  UINT       vme_data_width;
  BOOL        posted_writes;
} VME_PCISLAVE_INFO;


/* some handy routines */
UINT32    swapendian (UINT32 val)
{
          return (((0xff000000 & val)>>24) + ((0x00ff0000 & val)>>8)
                              + ((0x0000ff00 & val)<<8)
```

```
                                + ((0x000000ff & val)<<24));
}

UINT32    bitfield (UINT32 val, UINT32 mask, int shift)
{
        return ((swapendian(val) & mask) >> shift);
}

/* PciSlave stuff */

STATUS  PciSlaveImage1Set( VME_PCISLAVE_INFO   *info)
{
        int image = 1;
        if (uniPciSlaveImageSet (image,
                        info->pci_base_adr,
                        info->vme_base_adr,
                         info->vme_size,
                         info->pci_addr_space,
                         info->vme_am_code,
                         info->vme_data_width,
                         info->posted_writes)
                         == ERROR) return ERROR;
        return OK;
}

STATUS  PciSlaveImage1Default()
{
        int image = 1;
        if (uniPciSlaveImageSet (image, pciBase1d, vmeBase1d, size1d, pciAddrSpace1d,
                        vmeAmCode1d, vmeDataWidth1d, postedWrites1d) == ERROR) return ERROR;
        return OK;
}

/* TSVME module base addresses for RAM, ROM and jumper setting */
#define TSVME_RAM       0x8000;
#define TSVME_ROM       0xC000;
#define TSVME_BADR      0x00F00000;

UINT32    TSVMEadroffRAM = TSVME_RAM;
UINT32  TSVMEadroffROM = TSVME_ROM;

STATUS  tsvme()
{
        uint32_t   vmePtr;
        uint32_t   vmeAdr;
        int                             off;

/* set up PCI Slave image for the TSVME module */
        VME_PCISLAVE_INFO pci_slave_image;
        VME_PCISLAVE_INFO *pinfo = &pci_slave_image;

        pci_slave_image.pci_base_adr =          0xD1000000;
        pci_slave_image.vme_base_adr =         0x00000000;
        pci_slave_image.vme_size =              0x01000000;
        pci_slave_image.pci_addr_space =        UNI_PCI_MEMORY_SPACE;
        pci_slave_image.vme_am_code =          VME_AM_STD_SUP_DATA;            /* 0x3d */
        pci_slave_image.vme_data_width =        UNI_VMEBUS_DATAWIDTH_8;
        pci_slave_image.posted_writes =         TRUE;

/* show PciSlave mapping before and after setup */
        printf("===>uniImageShow before remapping\n"),
        uniImageShow();
        printf("\n");

        if (PciSlaveImage1Set(pinfo) == ERROR) return ERROR;
        printf("===> uniImageShow after re-mapping of PciSlave 1\n");
        uniImageShow();
        printf("\n");

        /* map from TSVME A24 jumper base address with pointer arithmetic for RAM/ROM acccess */
        vmeAdr = TSVME_BADR;
        if (sysBusToLocalAdrs(VME_AM_STD_SUP_DATA, (char*)vmeAdr, (void**)&vmePtr) == ERROR)
        {
                printf("TSVME: could not translate the VME address, check AM value\n");
                PciSlaveImage1Default();
```

```
                return ERROR;
        }

        printf("TSVME base address 0x%x is mapped to local BSP address 0x%x\n", vmeAdr, vmePtr);

        printf("\nDumping start of TSVME ROM\n");
        for(off=0; off<0x40; ++off)
        {
                if (off%16 == 0) printf("0x%6x", vmeAdr + TSVMEadroffROM + off);
                printf(" %02x", *((unsigned char*)vmePtr + TSVMEadroffROM + off));
                if (off%16==15) printf("\n");
        }

        printf("\nWriting to TSVME RAM with readback\n");

        ---- your job to fill in this code ----

        /* set PciSlaveImage 1 back to default setting */
        if (PciSlaveImage1Default() == ERROR) return ERROR;
        printf("\n");
        return OK;
}
```

4 M5000 target for Workbench

A VxWorks target "**tgt_192.168.0.12**" is defined on both lab PCs. However, the M5000 must be booted via COM1 terminal on PC-2, see RTlab-1.

Note, only one Workbench user can be connected at a time. If your neighbor insists on using the target then disconnect and wish her good luck.

A project must be built for PPC440sfgnu.

4.1 Additional header files for M5000

These files are not contained in the default header directory. This is signaled as shown in Figure 11.

Use the Add Folder command to include the directories shown in Figure 12, and move the three additional paths **up** to the top, see Figure 13.

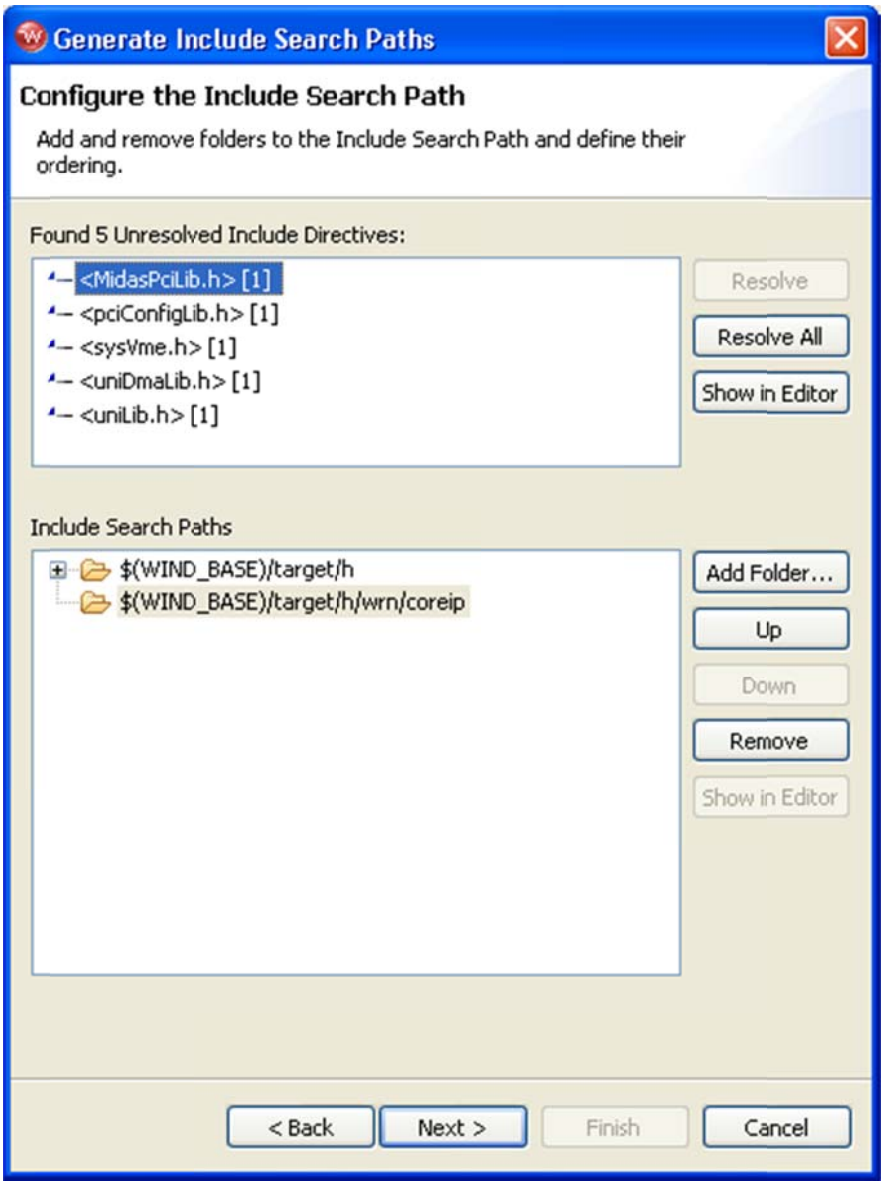


Figure 11 Include Search Paths for MIDAS M5000 BSP

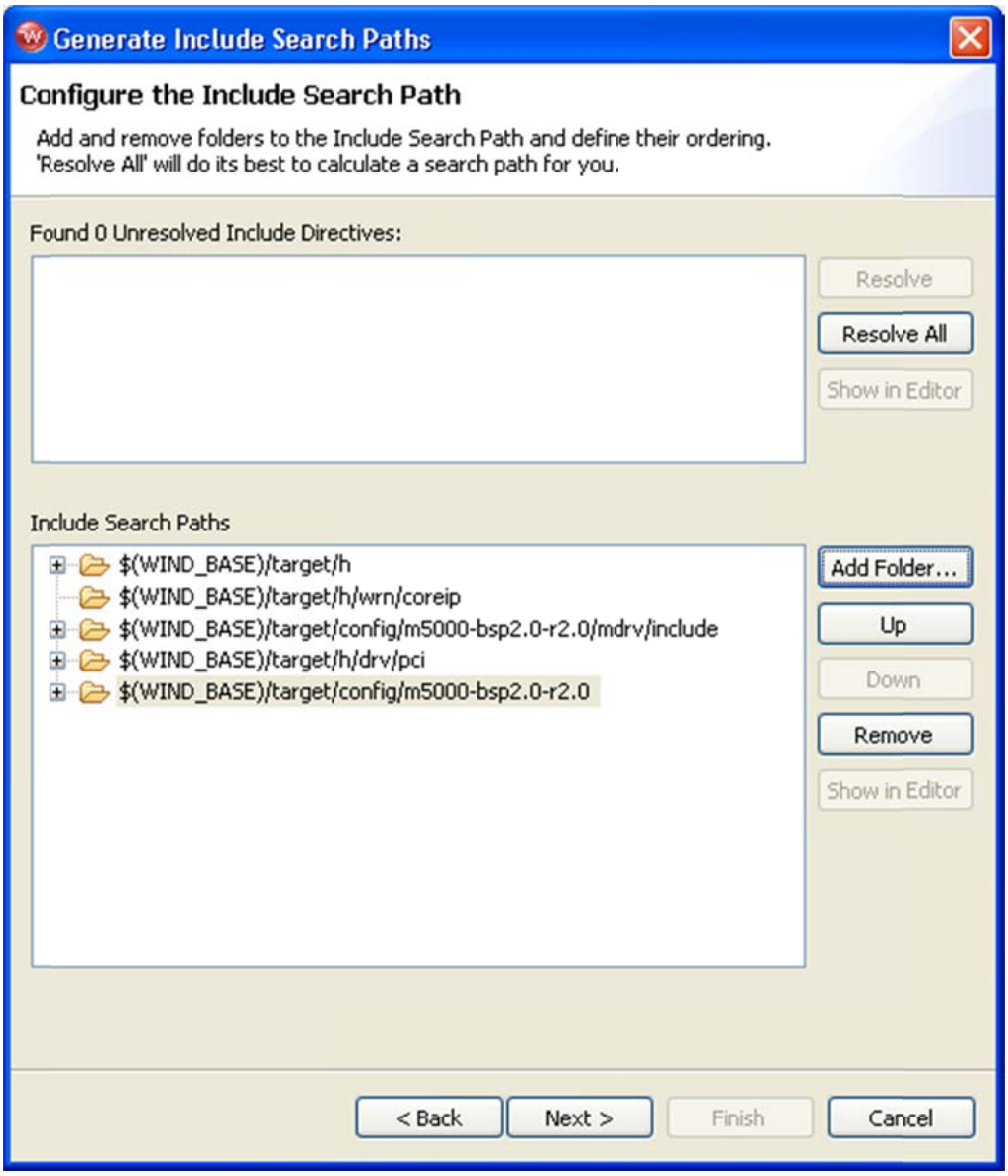


Figure 12 Add Include folders

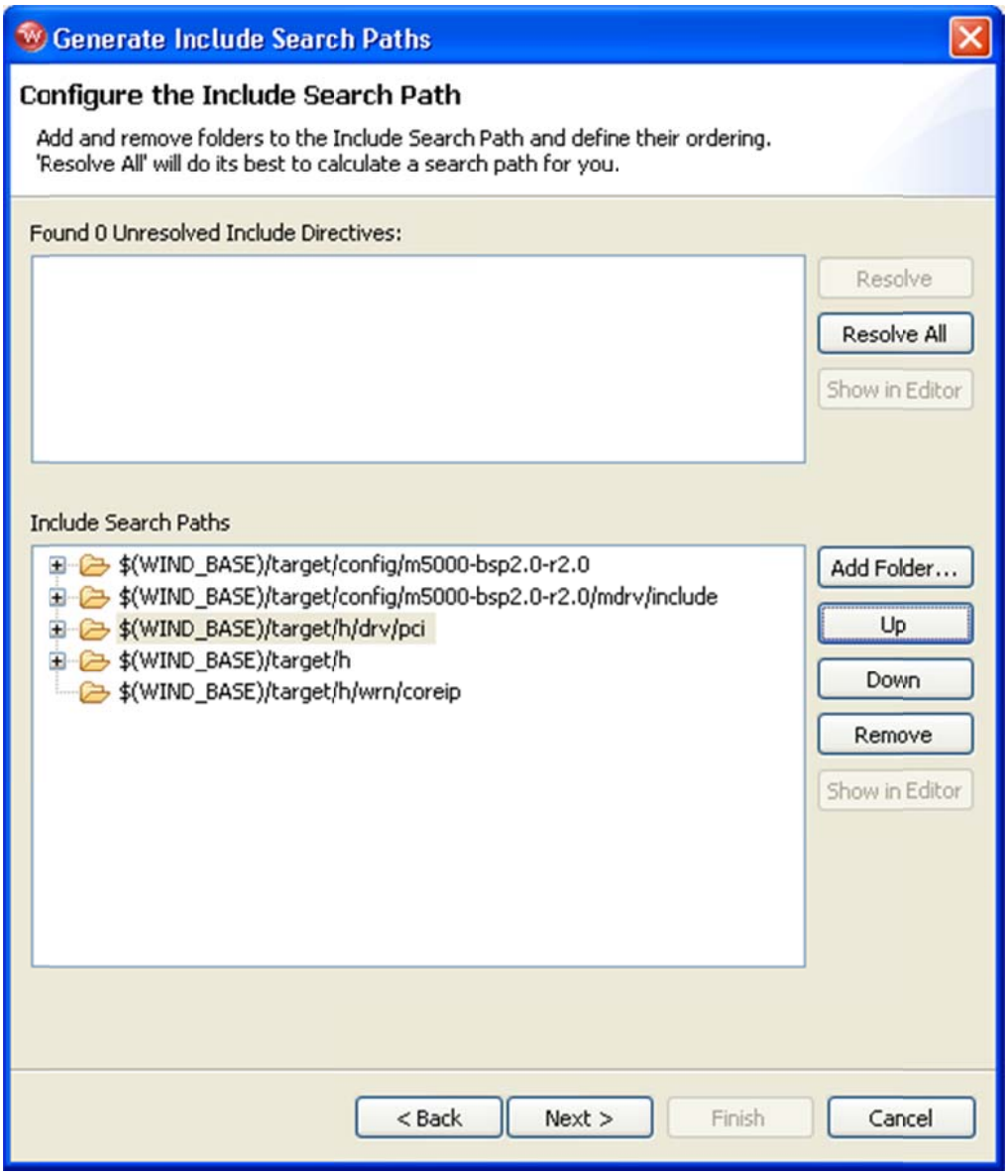


Figure 13 Include Search Paths after re-organization

4.2 Success!

A working RTlab-3 program should present output something like Figure 14.

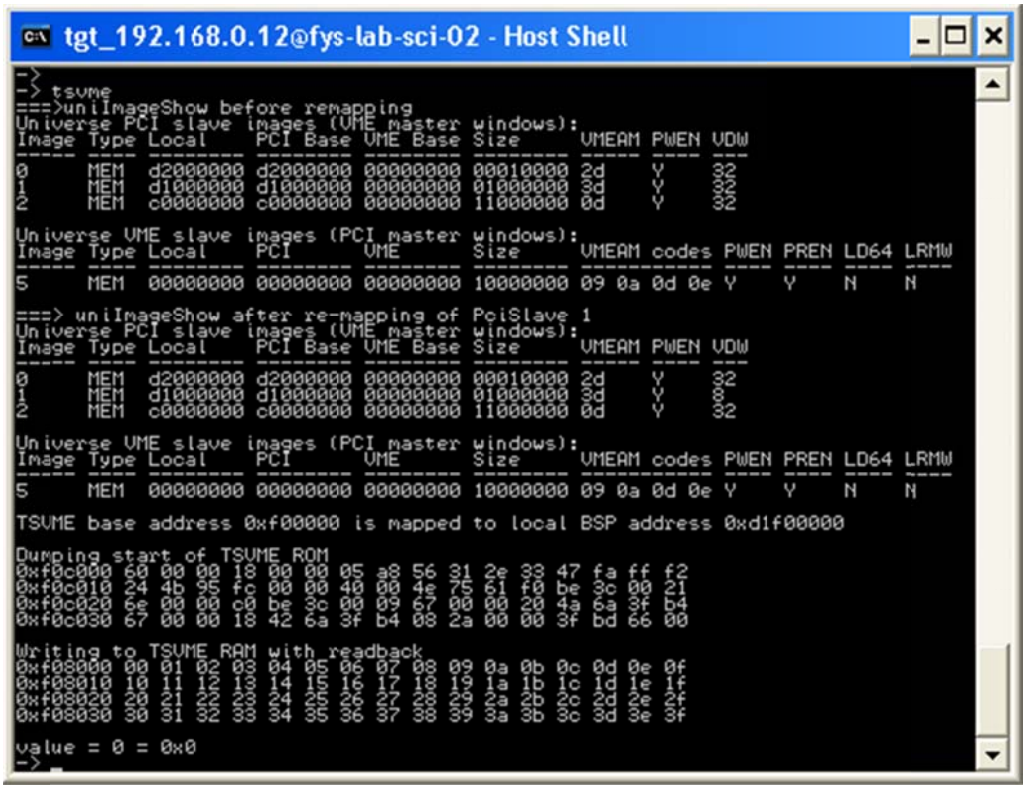


Figure 14 Output from RTlab-3.c with RAM write and read back implemented

4.3 Logic state analyzer

The activity on the VMEbus can be displayed by means of a connected LSA in timing mode, see instructions on the LSA. Bus analyzers are very useful for debugging.

5 Add-on for FYS9220 students – DMA transfers

Using RTlab-3.c as basis, allocate a PCI address area with **cacheDmaMalloc(size),** fill it with some intelligent information, DMA transfer it to the TSVME RAM, read it back and compare.

STATUS uniDmaLibInit()	seems that it shall be called without parameters
STATUS uniDmaDirect(….)	execute a DMA transfer according to the parameters. PCI and VME start addresses are mapped with **sysBusToLocalAdrs()**. The PCI address area is allocated with **cacheDmaMalloc(size),** which defines a cache safe safe buffer, see BSP User Guide p. 29.

One can also set up a chained DMA list between the same memories using the BSP functions **uniDmaChainCmdPktCreate()** and **uniDmaChain().**

Note!

DMA functions use the library **uniDmaLib.o.** Download the library to the target from

C:/WindRiver/vxworks-6.2/target/config/m5000-bsp2.0-r2.0/mdrv/lib