SSCL-N-800

Writing Device Drivers for the VxWorks Operating System

Case Study: MVME147-CAMAC and MVME167-CAMAC Device Drivers

C. Erbas, M. Botlo, and A. Fry

Superconducting Super Collider Laboratory*
2550 Beckleymeade Ave.
Dallas, TX 75237

May 1992

*Operated by the Universities Research Association, Inc., for the U.S. Department of Energy under Contract No. DE-AC35-89ER40486.

1. INTRODUCTION

This report is based on our efforts to port a CAMAC device driver [7] from the Sun Sparc workstation platform to two different VME modules, MVME147 [3] and MVME167 [10], which run the VxWorks real-time operating system. The report describes how to write a device driver and connect it to a VxWorks system, by providing examples from the MVME147-CAMAC and MVME167-CAMAC drivers. The differences between the I/O systems of UNIX (SUN OS 4.1.2) and VxWorks are also mentioned from the viewpoint of the device drivers.

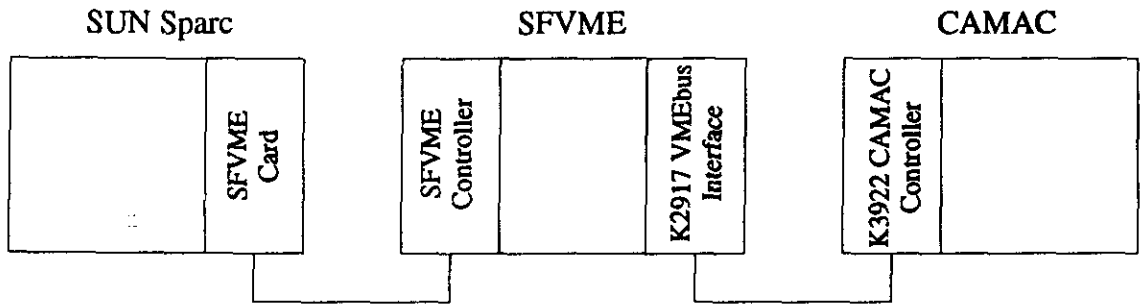

Figure 1. Configuration for Sun Sparc-CAMAC Device Driver .

The system configuration assumed by the Sun Sparc-CAMAC device driver is depicted in Figure 1. This figure illustrates the three components of the system: a Sun Sparc workstation, a Solflower SFVME crate, and a CAMAC crate. The workstation is connected to the Solflower crate with a SFVME controller. Further, Solflower is connected to the CAMAC crate with a K2917 VMEbus Interface module [1] (in the SFVME side) and K3922 CAMAC crate controller [2] (in the CAMAC side).

The MVME147S (or MVME167)-CAMAC device driver assumes the configuration shown in Figure 2. Here, the VME crate contains a MVME147S (or MVME167) module acting as the master. Similar to the previous configuration, the CAMAC crate is connected to the VME crate with K2917 and K3922 modules.

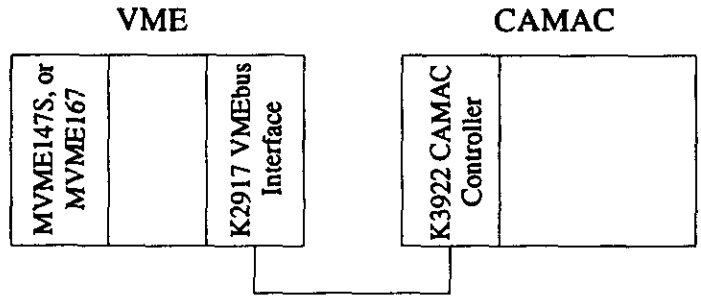

Figure 2. Configuration for MVME 147/167-CAMAC Device Driver .

2. AN OVERVIEW OF THE VxWORKS I/O SYSTEM

This section gives an overview of the implementation of the VxWorks I/O system from the viewpoint of the device drivers [4,5]. A device driver provides the software interface to an I/O device. The interface between a device driver and the VxWorks kernel consists of a set of entry points, which are called *driver routines*. When the VxWorks kernel recognizes that a particular action is required from an I/O device, it calls the appropriate driver routine. A driver for a non-block device provides the following seven driver routines: *creat(), delete(), open(), close(), read(), write()* and *ioctl()*. A device driver does not have to provide all the driver routines. The set of routines included in a particular device depends on the nature of the device. In addition to the above mentioned seven routines, interrupt handler routines can also be connected to the driver.

VxWorks DRIVER TABLE

	create	delete	open	close	read	write	ioctl
0							
1							
2							
3							
4							
5							
6							
7							
8	0	0	ccOpen	ccClose	ccRead	ccWrite	ccIoctl

ccdrvnum = iosDrvInstall (0, 0, ccOpen, ccClose, ccRead, ccWrite, ccIoctl);

Figure 3. Installing the CAMAC driver to the VxWorks driver table.

2.1 Driver Table and Device List

I/O requests of the user programs are routed to the driver routines by means of the *driver table*, which stores the start addresses of the driver routines. Drivers are added to this table by calling the *iosDrvInstall()* function, which is defined in the driver level I/O system library iosLib. This function returns the driver number of the new driver. As an example, the installation of the CAMAC device driver is illustrated in Figure 3.

It is apparent from Figure 3 that the CAMAC device driver does not provide the create() and the delete() driver routines. The function iosDrvInstall() returns 8 as the driver number. The contents of the driver table can be listed by calling the iosDrvShow() function. As an example, the content of the driver table, after the installation of the CAMAC driver, is given below.

```
DAQ1>iosDrvShow
drv  create   delete     open     close     read      write     ioctl
1    1a44     0        1a44      0        17a24     17960     1a50
2    0        0        1a942     0        1a94e     1a99a     1a9f0
3    0        0        0         129f4    1323c     12f8a     28b00
4    18cb4    18daa    18f32     18fde    198e4     19a2c     19b7a
5    1ac94    0        1ac94     1aca6    1acda     1ad5c     1adb4
6    1ac88    0        1ac88     1acc2    1acf4     1ad76     1adce
7    1a0f2    1a128    1a15c     1a2e2    1a348     1a4a8     1a636
8    0        0        fcafe6    fcb02c   fcb03c    fcb052    fcb1d2
value = 80 = 0x50 = 'P'
```

Sometimes, a driver serves many instances of the same device type. For example, as can be seen below, the first driver in our system serves four different devices, /tyCo/0, /tyCo/1, /tyCo/2, and /tyCo/3. In VxWorks, the devices are defined by a structure called *device header* (DEV_HDR), which is defined in iosLib.h, as follows.

```
typedef struct {
        NODE  node;
        short drvNum;
        char *name;
} DEV_HDR;
```

The device header of each device is stored in the device list. New devices are added to the device list by calling the iosDevAdd() function. As an example, the addition of the CAMAC device "/cc0" to the device list is given below.

```
pccDevHdr = &ccDevHdr;
strcpy(ccDevName, "/cc0");
iosDevAdd(pccDevHdr, ccDevName, ccDrvNum);
```

The list of the devices defined in the system can be displayed using the iosDevShow() function. As an example, the list of the devices in our system after the addition of "/cc0" is shown below.

```
DAQ1>iosDevShow
drv name
0 /null
1 /tyCo/0
1 /tyCo/1
1 /tyCo/2
1 /tyCo/3
4 scruff:
5 /pty/rlogin.S
6 /pty/rlogin.M
```

```
5 /pty/telnet.S
6 /pty/telnet.M
7 /usr5
7 /home
7 /daq1
8 /cc0
value = 0 = 0x0
```

2.2 Driver Routines and Interrupt Handling

The list of the seven driver routines were given before. This section summarizes the function of each driver routine. Further, how these routines are called from the user program, and how they are defined in the kernel level are also described.

1. open() routine: Whenever a user process makes an open() system call, VxWorks directs the call to the corresponding driver's open routine (for instance, ccOpen()). In the user level, all the I/O operations (read(), write(), ioctl()) require an fd. open() system call returns an fd, which will be used in the subsequent I/O operations on the same device. In the kernel level, basically, this routine is used to prepare the device for the subsequent I/O requests.

```
USER CALL:      fd = open ("name", flag, mode);
DRIVER CALL:    ccOpen (pccDevHdr, name, flags, mode)
                DEV_HDR *pccDevHdr;
                char *name;
                int flags;
                int mode;
```

2. close() routine: Whenever a user process issues a close() system call, VxWorks directs the call to the corresponding drivers close routine (for instance, ccClose()). This routine is used to close the file specified by the file descriptor.

```
USER CALL:      close (fd);
DRIVER CALL:    close ();
```

3. read() routine: Whenever a user process makes a read() system call, VxWorks routes the call to the corresponding drivers read routine (for instance, ccRead()). The user call gives a pointer to the buffer to store the data, and the number of bytes to read. In the kernel level, the driver routine is supposed to return the number of bytes read.

```
USER CALL:      nBytes = read (fd, &buffer, maxBytes);
DRIVER CALL:    ccRead (pccDevHdr, buf, len);
                DEV_HDR *pccDevHdr;
                char *buf;
                int len;
```

4. write() routine: Whenever a user process issues a write() system call, VxWorks routes the call to the corresponding drivers write routine (for instance, ccWrite()). The user call gives a pointer to the buffer containing the data to be written to the device, and the number of bytes to be output. In the kernel level, the driver routine is supposed to return the number of bytes written.

USER CALL: actualBytes = write (fd, &buffer, nBytes);

DRIVER CALL: ccWrite (pccDevHdr, buffer, nbytes)
DEV_HDR *pccDevHdr;
char *buffer;
int nbytes;

5. ioctl() routine: Whenever a user process makes a ioctl() system call, VxWorks directs the call to the corresponding drivers ioctl routine (for instance, ccIoctl()). This mechanism allows the user to perform any I/O function that cannot be performed using the other six basic I/O calls.

USER CALL: result = ioctl (fd, function, arg);

DRIVER CALL: ccIoctl (pccDevHdr, function, arg)
DEV_HDR *pccDevHdr;
int function;
int arg;

6. creat() routine: The creat() routine creates a new file on the device and returns a file descriptor for it. The difference between create() and open() is that the latter takes the name of an existing file as the parameter, although the former a new file. This routine is not provided in the CAMAC device driver.

USER CALL: fd = creat ("name", flag);

7. delete() routine: The delete() routine deletes a named file. This routine is not provided in CAMAC device driver.

USER CALL: delete ("name");

8. interrupt() routine: In VxWorks interrupt routines are connected to any interrupt using the *intConnect()* function. In the CAMAC device driver, the interrupt handling routine ccIntr() is connected to the interrupt vector using the following call.

```
intConnect(INUM_TO_IVEC(INT_VEC_LAM), ccIntr, 0);
```

INUM_TO_IVEC is defined in the header file $VWROOT/h/68k/iv.h, as follows:

```
#define IVEC_TO_INUM(intVec)      ((int) (intVec) >> 2)
```

There are two iv.h files, one in $VWROOT/h and one in $VWROOT/h/68k. In order to let the compiler to find the correct header file, the driver should be compiled with the -DCPU _FAMILY=MC680X0 switch.

2.3 Differences Between UNIX (Sun OS) and VxWorks Driver Routines

The parameters of the UNIX (Sun OS 4.1.2) [8,9] driver routines are slightly different than in the VxWorks. It is useful to have the list of the differences between the parameters passed to the driver routines. The following table provides the differences for open(), read(), write(), and ioctl().

VxWorks	Unix (Sun OS 4.1.2)
ccOpen(pccDevHdr, name, flags, mode)	ccopen(dev, flags)
DEV_HDR *pccDevHdr;	dev_t dev;
int flags;	int flags;
int mode;	

```
ccRead(pccDevHdr, buf, len)
DEV_HDR *pccDevHdr;
char *buf;
int len;
```

```
ccread(dev, uio)
dev_t dev;
struct uio *uio;
```

```
ccWrite(pccDevHdr, buffer, nbytes)
DEV_HDR *pccDevHdr;
char *buffer;
int nbytes;
```

```
ccwrite(dev, uio)
dev_t dev;
struct uio *uio;
```

```
ccIoctl(pccDevHdr, function, arg)
DEV_HDR *pccDevHdr;
int function;
int arg;
```

```
ccioctl(dev, cmd, data, flag)
dev_t dev;
int cmd;
short *data;
int flag;
```

2.4 Differences Between Sun OS and VxWorks Kernel Support Routines

The Sun Sparc-CAMAC device driver uses some of the SUN OS kernel support routines, which are not provided by the VxWorks operating system. The list of those routines, and their functions are given below.

1. **copyin():** moves data from the user address space to the kernel address space.
2. **copyout():** moves data from the kernel address space to the user address space.
3. **gsignal():** sends a signal to all the processes in a process group.
4. **iodone():** indicates that an I/O operation has been completed.
5. **hat_getkpfnum():** returns the associated page frame number of a virtual address.
6. **MBI_ADDR():** returns the address of the buffer in DVMA space.
7. **mb_mapalloc():** allocates address in DVMA space for I/O transfers.
8. **mb_mapfree():** deallocates the buffer in DVMA space.
9. **peek():** checks whether an address location exists or not by reading.
10. **physio():** is a service routine for block I/O operations.
11. **poke():** checks whether an address location exists or not by writing.
12. **sleep():** puts the calling process to sleep until a wakeup call issued.
13. **splr():** raises the priority level.
14. **timeout():** waits for a predefined interval and then calls a function.
15. **uprintf():** is an interruptible kernel printf function.
16. **untimeout():** cancels a prior timeout request..

In UNIX, the kernel address space is different than the user address space. Some of the kernel support routines given above, such as, copyin() and copyout(), are used to deal with this distinction. VxWorks does not separate the kernel space from the user space. Hence, those routines are simply discarded when porting the driver. Similarly,in VxWorks, it is not necessary

to allocate a separate DVMA space for DMA transfer. So the routines MBI_ADDR(), mb_mapalloc() and mb_mapfree() are not used in the VxWorks version of the driver. Further, some other routines, such as uprintf() and iodone() are not necessary and have not been used in the device driver.

The routines that are provided for the synchronization of the processes, such as gsignal(), sleep(), timeout(), untimeout() are replaced by the VxWorks functions given in the sigLib and wdLib libraries. The list of those VxWorks functions are given below.

1. **sigInit():** is used to initilize the signal facilities provided in sigLib.
2. **sigvec():** is used to connect a specific signal to a signal handler.
3. **pause():** blocks the execution of the process until a signal occured. This routine corresponds to the sleep() routine in UNIX.
4. **kill():** sends a signal to a specific process. This signal corresponds to the gsignal() routine in UNIX.
5. **wdCreate():** is used to create a watchdog timer.
6. **wdStart():** starts a watchdog timer and attaches the routine which will be called after a specified time. This routine corresponds to the timeout() routine in UNIX.
7. **wdCancel():** cancels a currently running watchdog timer. This routine corresponds to the untimeout() routine in UNIX.

3. VMEbus - CAMAC INTERFACE

As illustrated in Figure 2, the connection between the VMEbus and the CAMAC crates are handled by the K2917 VMEbus Interface w/DMA module [1], which provides an interface between a VMEbus system and up to eight CAMAC crates using a K3922 Parallel Bus Crate Controller [2]. In this section, we discuss the issues concerned with the interface between the VMEbus and CAMAC. The registers of K2917 and K3922 and their role in the interface are also described. The schematic representation of this connection is given in Figure 4.

3.1 Accessing the Registers of the K2917

K2917 VMEbus Interface contains three sets of registers: DMA Controller Registers, Bus Interrupt Registers, and K2917 On-Board Registers. The complete list of registers and their D16 offsets are given below. The description of each register can be found in [1].

1. DMA Controller Registers	D16 Offset
1.1 Channel Status/Error Register	$00
1.2 Device/Operation Control Register	$04
1.3 Sequence/Channel Control Register	$06
1.4 Memory Transfer Count Register	$0A
1.5 Memory Address Counter (High) Register	$0C
1.6 Memory Address Counter (Low) Register	$0E

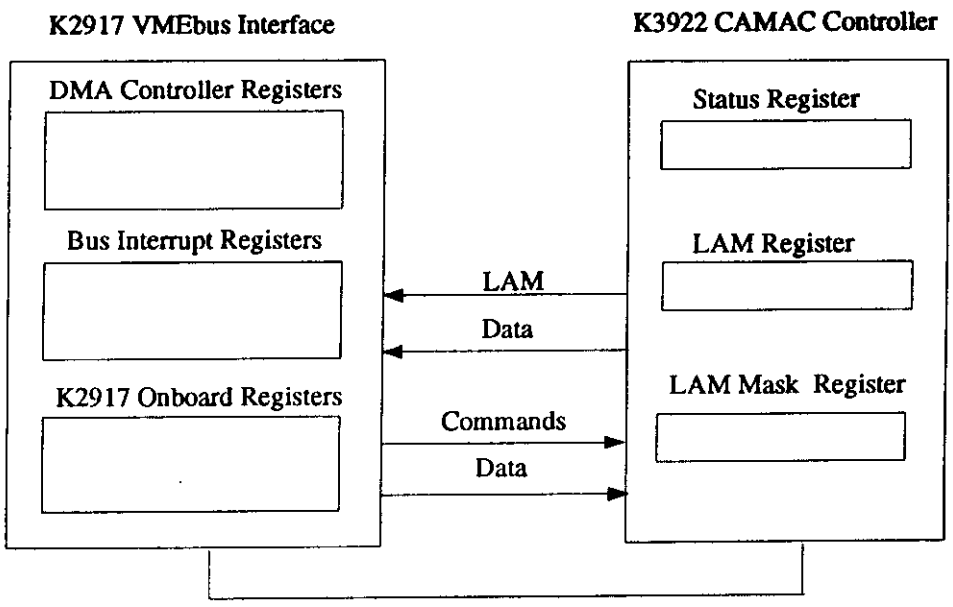

Figure 4. VME-CAMAC Interface Modules.

2. Bus Interrupt Registers	**D16 Offset**
2.1 Interrupt Control Registers	
2.1.a LAM	$40
2.1.b DONE	$42
2.1.c DMA BUFFER EMPTY	$44
2.1.d LIST ABORT	$46
2.2 Interrupt Vector Registers	
2.2.a LAM	$48
2.2.b DONE	$4A
2.2.c DMA BUFFER EMPTY	$4C
2.2.d LIST ABORT	$4E

3. K2917 On-Board Registers	**D16 Offset**
3.1 Address Modifier Register	$60
3.2 Command Memory Register	$62
3.3 Command Memory Address Register	$64
3.4 Command Word Count Register	$66
3.5 Service Request Register	$68
3.6 Data (Low) Register	$6A
3.7 Data (High) Register	$6C
3.8 Control Status Register	$6E

The corresponding structure (KREG = CAMAC) for the registers of K2917 is defined in the header file k2917.h. The K2917 resides in the short I/O address space of the VMEbus. The base address of the K2917 is determined by the switches SA15, SA14, ...,SA08 as described in [1]. In our driver, the base address is equal to 0x00FF. The following code segment maps the registers of the k2917 to KREG.

```
#define KREG_BASE 0xFF00
uint16   kbase = KREG_BASE;
struct    CAMAC *k = ((struct CAMAC *)0);
uint32    kreg_addr();
k = (struct CAMAC *) kreg_addr(kbase);

uint32 kreg_addr(k2917_addr)
faddr_t k2917_addr;
{
    faddr_t local_addr;
    STATUS s;

    s = sysBusToLocalAdrs(VME_AM_USR_SHORT_IO, k2917_addr, &local_addr);
    if (s == OK) return((uint32) local_addr);
    else printf("sysBusToLocal failed...\n");
}
```

3.2 Handling Interrupts

The driver provides the necessary mechanisms to forward an interrupt request to the MVME147/167, whenever a LAM signal is initiated by a module in the CAMAC crate. The LAM request is transmitted to MV147/167 in three steps. In the first step, K3922 is informed about the LAM. The K3922 provides two registers, namely a LAM register and a LAM mask register, to control the LAM signals coming from the CAMAC modules. The LAM register contains 24 bits each of which corresponds to one station in the crate. Every bit indicates the current status of the LAM requests of the corresponding station. The LAM requests of the CAMAC modules can be masked using the LAM mask register. When the K3922 controller receives a LAM request the SLP LED becomes on.

In the second step, K3922 transmits the LAM signal to the K2917. As it is described in Section 3.1, the K2917 contains a LAM interrupt control register (LAMC) and a LAM interrupt vector register (LAMV). LAMC is used to enable the LAM requests, and to determine its interrupt request level. LAMV contains the interrupt vector of the LAM requests.

In the third step, K2917 sends an interrupt signal to MVME 147 or 167. In order to let the processor receive the interrupt signal, the interrupts should be enabled. The addresses and the contents of the interrupt control registers are different for MVME147 and MVME167 [4,7,11]. It should also be noted that the registers in MVME 147 are accessed in 16 bits. However, in MVME167, the accesses to the registers are in 32 bits. In our device driver, the interrupts are enabled as follows:

```
#ifdef MVME147
        p = (u_short *) 0xfffe200e;  /* Interrupt Handler Mask Register of MVME147 */
        *p = (u_short) (0xff02);
#endif
#ifdef MVME167
        p = (int *) 0xfff4002c; /* Local Bus Interrupter Enable Register of MVME167 */
        *p = (int)(0x40000001);
#endif
```

4. OPERATING MODES OF THE K2917

K2917 provides four different types of operating modes: slave single transfer, slave block transfer, DMA single transfer, and DMA block transfer mode. A transfer operation may be 16 bits or 24 bits. Further, it may be from CAMAC to VME (read), or from VME to CAMAC (write). In this section, we describe how these transfer modes are implemented in the device driver.

4.1 Slave Single Transfer Mode

In this mode, during a CAMAC transfer operation, the DONE bit of the Command Status Register (CSR) is not set until the data transfer is completed. The RDY bit of the CSR indicates whether the data can be read from the data registers, or it can be written to data registers. In our device driver, slave single transfer mode is handled by the ***camac_s()*** procedure. The details of this procedure is described below.

Step 1. Initialize memory pointer, and load the Command Memory with the command list.

```
k->cma = CMA_INIT;
k->cmr = mode | (cur_crate << 8);
k->cmr = naf;
k->cmr = HALT;
k->cma = CMA_INIT;
```

Step 2. Set the DIR and GO bits of the CSR.

```
if CAMAC read operation k->csr &= ~WRITEC;
else if CAMAC write operation k->csr |= WRITEC;
k->csr |= GO;
```

Step 3. Wait until the RDY or the ERR bit of the CSR is set to one.

```
counter = 0;
while ((k->csr & (RDY | ERR)) == 0 && counter < TIMEOUT_COUNT_S)
    counter++ ;
```

Step 4. Read or write the data registers according to the operation type.

```
if CAMAC read operation
    if ((k->csr & RDY) != 0) {
        if ((mode & BIT16) == 0) {
```

```
            *dat = k->dhr & 0x00FF;
            *(dat+1) = k->dlr;
        }
        else {
            *(dat+1) = k->dlr;
        }
    }
else if CAMAC write operation
    if ((k->csr & RDY) != 0) {
        if ((mode & BIT16) == 0) {
            k->dhr = *dat;
            k->dlr = *(dat+1);
        }
        else {
            k->dlr = *(dat+1);
        }
    }
```

Step 5. Wait until the DONE or the ERR bit of the CSR is set to 1.

```
while ((k->csr & (DONE | ERR)) == 0 && counter < TIMEOUT_COUNT_S)
    counter++ ;
```

4.2 Slave Block Transfer Mode

K2917 provides four different types of block transfers: Q-STOP, Q-IGNORE, Q-REPEAT, and Q-SCAN. The RDY bit of the CSR indicates whether the data can be read from data registers, or it can be written into data registers. In our device driver, slave block transfer mode is handled by the ***camac_b()*** procedure. The details of this procedure is described below.

Step 1. Initialize memory pointer, and load the Command Memory with the command list.

```
k->cma = CMA_INIT;
k->cmr = mode | (cur_crate << 8);
k->cmr = naf;
k->cmr = -(len & 0xFFFF);
k->cmr = 0xFFFF;
k->cmr = HALT;
k->cma = CMA_INIT;
```

Step 2. Set the DIR and GO bits of the CSR.

```
if CAMAC read operation k->csr &= ~WRITEC;
else if CAMAC write operation k->csr |= WRITEC;
k->csr |= GO;
```

Step 3. Until the DONE or the ERR bit of the CSR is set to one, perform Step 4 and 5..

```
while ((k->csr & (ERR|DONE)) == 0) {
    { Step 4 }
    { Step 5 }
}
```

Step 4. Wait until the RDY or the ERR bit of the CSR is set to one.

```
while ((k->csr & (RDY | ERR)) == 0 && counter < TIMEOUT_COUNT_S)
    counter++;
```

Step 5. Read or write the data registers according to the operation type. (Only read operation is provided for slave block mode).

```
if ((k->csr & RDY) != 0) {
    if ((mode & BIT16) == 0) {
        *dat = k->dhr & 0x00FF;
        *(dat+1) = k->dlr;
    }
    else {
        *(dat+1) = k->dlr;
    }
}
```

4.3 DMA Block Transfer Mode

In DMA block transfer mode is similar to the slave block transfer mode, however, this time the K2917 becomes the VME bus master, and manages all the hand shaking operations. The DMA controller of 2917 is limited to transfer 64k words. So, if the transfer block size is larger than 64k, then the DMA operation should be repeated for every 64k words. In our device driver, DMA block transfer mode is handled by the ***camac_mb()*** procedure. The details of this procedure is described below.

Step 1. Initialize memory pointer, and load the Command Memory with the command list.

```
k->cma = CMA_INIT;
k->cmr = mode | (cur_crate << 8);
k->cmr = naf;
k->cmr = -(len & 0xFFFF);
k->cmr = 0xFFFF;
k->cmr = HALT;
k->cma = CMA_INIT;
```

Step 2. Load the MACHI and MACLO with the starting address of the data buffer.

```
k->maclo = dma_addr & 0xFFFF;
k->machi = dma_addr >> 16;
```

Step 3. Set the AMR register to the appropriate address modifier code; load the MTC with the number of memory counts; and clear CSER.

```
k->amr = AMR_INIT;
k->mtc = wc;
k->cser = DMA_RESET;
```

Step 4. Initialize the DOCR register; and in order to start DMA controller set SCCR with 0x80.

```
if CAMAC read operation k->docr = DOCR_INIT | DMA_READ;
```

```
else if CAMAC write operation k->docr = DOCR_INIT | DMA_WRITE;
k->sccr |= DMA;
```

Step 5. Set the DMA, DIR and GO bits of the CSR.

```
if CAMAC read operation k->csr &= WRITEC;
else if CAMAC write operation k->csr = WRITEC;
k->csr = DMA;
k->csr |= GO;
```

Step 6. Wait until the DONE or the ERR bit of the CSR is set to one.

```
while ((k->csr & (DONE | ERR)) == 0 && counter < TIMEOUT_COUNT_B) {
    for (i = 0; i < 1000; i++) j += i;
    counter++;
}
```

5. LIBRARY ROUTINES AND USER EXAMPLES

In order to facilitate the use of driver routines a set of high level procedures are provided in the CAMLIB library. These procedures allow the user to write programs using driver functions without knowing the details of the driver. In this section, we provide three example test programs for single action transfer, interrupts, and block transfer modes. The block transfer test program is capable of performing both slave block and DMA block transfers. For every test program, we will explain the functions of the library routines.

5.1 Single Action Test Program

This program uses the slave single transfer mode. The following CAMLIB library routoines are called from this program: CAMOPN(), CGENC(), CGENZ(), CSETI(), CREMI(), CAMAC(), CAMCLS(), and NAF(). The routines CAMOPN() and CAMCLS are used to open and close the camac driver. CGENC() and CGENZ() generate clear and initialize signals to the CAMAC crate, respectively. The routines CSETI() sets the inhibit signal, and CREMI()removes the inhibit signal. The routine CAMAC() makes a slave single transfer based on its input NAF(n, a, f). Here, n, a, and f correspond to the CAMAC crate slot number, subaddress, and the function code, respectively.

```
#include  <stdio.h>
#include  "camlib.h"

cam1_main(argc, argv)
int argc;
char *argv[];
{
  int loop;
  int n, a, f, q, x, dat;

  if (CAMOPN()) {
        perror("Open error: ");
```

```
    exit(1);
  }
  CGENC();
  CGENZ();
  CSETI();
  CREMI();
  while (loop-- > 0) {
    printf("Input n a f (data)>");
    scanf("%d %d %d*", &n, &a, &f);
    if ((f & 0x1C) == 0x10) scanf("%d*", &dat);
    CAMAC(NAF(n, a, f), &dat, &q, &x);
    printf("  N=%d A=%d F=%d Q=%d X=%d  Data:%06X(Hex) %08d(Dec)\n\n", n, a, f, q, x, dat, dat);
  }
  CAMCLS();
}
```

5.2 Interrupt Test Program

This program is written to test the interrupts. The mechanism of the interrupt handling is presented in Section 3.2. In addition to the CAMLIB library routines given in the single action test program, the interrupt test program uses the following routines: CSETBR(), CSETCR (), CELAM(), CWLAM(), and CDLAM(). The routines CSETBR() and CSETCR() sets the branch and crate numbers.CELAM() enables the LAM signal, and CDLAM disables the LAM signal. CWLAM waits until the occurence of a LAM signal or a timeout. The following program assumes that there is a NIM interrupt module in CAMAC slot 23.

```
#include <stdio.h>
#include <sys/fcntl.h>
#include <sys/file.h>
#include "camlib.h"

#define DEFN      23
#define DEFLOP    10
#define DEFTIMEOUT  100

cam2_main(argc, argv)
int argc;
char *argv[];
{
  int i, j, n, mask, status;
  int nafenalam, nafdislam, nafclrlam, nafchklam;
  int dat, q, x;
  int loop, timeo;

  n    = DEFN;
  loop = DEFLOP;
  timeo = DEFTIMEOUT;
```

```
    mask = 1 << (n-1);
    nafenalam = NAF(n, 0, 26);
    nafdislam =  NAF(n, 0, 24);
    nafclrlam =  NAF(n, 0, 10);

    CAMOPN();
    CSETBR(1);
    CSETCR(0);
    CGENC();
    CGENZ();
    CREMI();

    CAMAC(nafenalam, &dat, &q, &x);
    CELAM(mask);
    for (i = 0; i < loop; i++) {
      if (CWLAM(timeo)) printf("Timeout: ");
      printf("Interrupted !! count = %d\n, i+1);
      CAMAC(nafclrlam, &dat, &q, &x);
    }
    CDLAM();
    CAMCLS()
}
```

5.3 Block Transfer Test Program

This program tests both the slave block and the DMA block transfer modes. The mechanism of the block transfer mode is given in Section 4.2 and 4.3. Block transfers are handled by the CAMACB() procedure. In the following program the variable *len* gives the length of the block transfer.

```
#include   <stdio.h>
#include   "camlib.h"

camlb_main(argc, argv)
int argc;
char *argv[];
{
    int loop, len, i;
    int n, a, f, q, x;
    u_short dat[2048];

    if (CAMOPN()) {
          perror("Open error: ");
      exit(1);
    }
```

```
    CGENC();
    CGENZ();
    CSETI();
    CREMI();
    while (loop-- > 0) {
      printf("Input n a f >");
      scanf("%d %d %d*", &n, &a, &f);
      printf("Input len >");
      scanf("%d", &len);
      for (i=0; i < len; i++)  dat[i] = 0;
      CAMACB(NAF(n,a,f), dat, &q, &x, len);
      for (i = 0; i < len; i++) printf("dat[%d] = %X\n", i, dat[i]);
    }
}
```

---

6. PERFORMANCE

In this section, we provide the data transfer speed of our device driver for single block transfer mode, and DMA block transfer mode. The results are obtained empirically. Our test results demonstrate that the data transfer time depends on several factors, such as the types of the CAMAC modules, their relative positions, the number of the empty slots, the state of the system, etc. However, the following framework can be used to calculate the approximate run time of the single and DMA transfer modes.

6.1 Performance of the Single Block Transfer Mode

The data transfer time (T_d) for single block transfer mode is given with the following formula:

$$T_d = T_i + (N_c * T_c) + T_t$$

where, T_i is the initialization time, N_c is the number of channels, T_c is the transfer time per channel, and T_t is the termination time.

According to our test results the initialization time (T_i), and the termination time (T_t) for single block transfer mode takes 25 microseconds, and 7 microseconds, respectively. In scan mode, the transfer time per channel (T_c) is approximately 6.6 microsecond. Therefore, the data transfer time can be given as

$$T_d = (6.6 * N_c) + 32.$$

Assuming an infinite number of channels and 16-bit data for each channel, the maximum data transfer rate in single block transfer mode is calculated as 1/3.3 MB/sec = 330 KB/sec.

6.2 Performance of the DMA Block Transfer Mode

Similar to the single block transfer, in DMA block transfer mode, the data transfer time (T_d) is given with the following formula:

$$T_d = T_i + (N_c * T_c) + T_t$$

where, T_i is the initialization time, N_c is the number of channels, T_c is the transfer time per channel, and T_t is the termination time.

According to our test results the initialization time (T_i), and the termination time (T_t) for DMA block transfer mode takes 27 microseconds, and 52 microseconds, respectively. In scan mode, the tranfer time per channel (T_c) is approximately 2.7 microsecond. Therefore, the data transfer time can be given as

$$T_d = (2.7 * N_c) + 79.$$

Assuming an infinite number of channels and 16-bit data for each channel, the maximum data transfer rate in single block transfer mode is calculated as 1/1.35 MB/sec = 740 KB/sec.

The 16-bit transfer time for ***scan mode*** and the overhead of different transfer modes and the interrupt latency of the driver is given in the following table.

	Overhead	16-bit Transfer Time
Slave Single Transfer	70 µsec	(*)
Slave Block Transfer	32 µsec	6.6 µsec
DMA Block Transfer	79 µsec	2.7 µsec
Interrupt Latency	40 µsec	----

ACKNOWLEDGEMENTS

We are thankful to Y. Takeuchi from Tokyo Institute of Technology and Y. Nomachi from KEK, who developed and provided the Sun OS version of the CAMAC device driver. We also thank to Micheal Jacgielski for helpful discussions during the development of the VxWorks version of the driver.

REFERENCES

[1] Model 2917-Z1A, VMEbus Interface w/DMA Instruction Manual, Kinetic Sytems, (1991).

[2] Model 3922-Z1B, Parallel Bus Crate Controller, Instruction Manual, Kinetic Sytems, (1991).

[3] MVME147S MPU VMEmodule User's Manual, Motorola, (1990).

[4] VxWorks 5.0 Programmer's Guide, Wind River Systems.

[5] VxWorks 5.0 Reference Manual, Wind River Systems.

[6] VxWorks Target-Specific Documentation, Motorola MVME-147, Wind River Systems (1989).

[7] Y. Takeuchi, Development of a fast UNIX-VME Data Acquisition System, Master Thesis (in Japaneese), Tokyo Institute of Technology, (1992).

[8] Egan, J.I., and Teixeira, T.J., Writing a Unix Device Driver, Wiley, 1988.

[9] Sun OS Writing Device Drivers, 1990.

[10] MVME 167/MVME187 Single Board Computers Programmer's Reference Guide, Motorola, (1991).

APPENDIX

PROGRAM LISTING OF THE DEVICE DRIVER

```
#-------------------------------------------------------------#
#                                                             #
#   Makefile      May 20, 1992          Cengiz Erbas  (SSCL)   #
#                                                             #
#-------------------------------------------------------------#
#
# inputs:  environment variables
#          VWROOT where VxWorks is e.g. /usr5/vw for mvme147
#                                    or /usr5/vw40 for mvme167
#          CPU  (147 or 167) CPU model #
#
VX50_PATH = $(VWROOT)

VX50_INCL = -I$(VX50_PATH)/h -I$(VX50_PATH)/h/drv -I$(VX50_PATH)/config/mv$(CPU) -I$(VX50_
PATH)/config/all

CC = cc68k

CFLAGS = -DCPU_FAMILY=MC680X0 -DMVME$(CPU)

OBJ = ccmain.o camlib.o ccdriver.o cam1.o cam1b.o cam2.o

all: $(OBJ)

ccmain.o: ccmain.c
        cc68k -c $(CFLAGS) $(VX50_INCL) -O ccmain.c -o ccmain.o

camlib.o: camlib.c
        cc68k -c $(CFLAGS) $(VX50_INCL) -O camlib.c -o camlib.o

ccdriver.o: ccmain.o camlib.o
        ld68k -o ccdriver.o -r ccmain.o camlib.o

cam1.o: cam1.c
        cc68k -c $(CFLAGS) $(VX50_INCL) -O cam1.c -o cam1.o

cam1b.o: cam1b.c
        cc68k -c $(CFLAGS) $(VX50_INCL) -O $(?) -o cam1b.o

cam2.o: cam2.c
        cc68k -c $(CFLAGS) $(VX50_INCL) -O cam2.c -o cam2.o
```

```
/*-------------------------------------------------------------*/
/*    ccmain.c     May 20, 1992          Cengiz Erbas  (SSCL)   */
/*                                                             */
/*      for MVME147 K2917 K3922 or MVME167 K2917 K3922         */
/*                                                             */
/* original by Y.Takeuchi (TIT) for SUN-SPARC SF-VME K2917 K3922 */
/*-------------------------------------------------------------*/

#include <stdio.h>
#include "vxWorks.h"
#include "iv.h"
#include "ioLib.h"
#include "iosLib.h"
#include "config.h"
#include "sigLib.h"
#include "wdLib.h"
#include "vme.h"

#ifdef MVME167
#include "mv167.h"
#endif

#ifdef MVME147
#include "mv147.h"
#endif

#include "cc.h"
#include "k2917.h"

LOCAL int ccDrvNum;
LOCAL DEV_HDR ccDevHdr;

LOCAL int ccOpen();
LOCAL int ccClose();
LOCAL int ccRead();
LOCAL int ccWrite();
LOCAL int ccIoctl();
LOCAL int ccIntr();
LOCAL void cctimeout();
LOCAL VOID sigHandler();
LOCAL VOID clkHandler();

#define TIMEOUT_COUNT_S 1000
#define TIMEOUT_COUNT_B 100000
#define INT_VEC_LAM     VME_INT_VECTOR

typedef unsigned char *faddr_t;
typedef unsigned short uint16;
typedef unsigned long  uint32;

u_short camac_qx;
uint16 kbase = KREG_BASE;
struct CAMAC *k = ((struct CAMAC *) 0);
struct cc_device ccdevice[NCC];
int task_id, clk_count;
WDOG_ID wid;
SIGVEC pVec, pOvec;

/*-------------------------------------------------------------*/
/*                                                             */
/* ccDrv(): Installs the cc driver functions to driver         */
/*          table, and adds the device to the device list      */
/*                                                             */
/*-------------------------------------------------------------*/
```

```
STATUS ccDrv()
{
    char *ccDevName;
    STATUS status;
    DEV_HDR *pccDevHdr;
    register struct cc_device *cc = &ccdevice[0];
    uint32 kreg_addr();
    int vector;

    k = (struct CAMAC*) kreg_addr(kbase);
    cc->max_branch = CC_K_MAX_BRANCH;
    cc->cur_branch = 0;
    cc->cur_crate  = 0;

    intConnect(INUM_TO_IVEC(INT_VEC_LAM), ccIntr, 0);
    ccDrvNum = iosDrvInstall((FUNCPTR) NULL, (FUNCPTR) NULL, ccOpen,
       ccClose, ccRead, ccWrite, ccIoctl);
    printf("DrvNum = %d\n", ccDrvNum);

    pccDevHdr = &ccDevHdr;
    strcpy(ccDevName, "/cc0");
    iosDevAdd(pccDevHdr, ccDevName, ccDrvNum);
    if (status == ERROR) {
       printf("Error in iosDevAdd()... \n");
       exit(1);
    }
    sigInit();
    clk_count = 0;
    sysAuxClkConnect(clkHandler,0);
    sysClkRateSet(1000);
    sysAuxClkEnable();
    return(OK);
}

/*------------------------------------------------------------*/
/*                                                            */
/* ccOpen(): CAMAC open procedure.                            */
/*                                                            */
/*------------------------------------------------------------*/
LOCAL int ccOpen(pccDevHdr, name, flags, mode)
DEV_HDR *pccDevHdr;
char *name;
int flags;
int mode;
{
   register struct cc_device *cc = &ccdevice[0];
   u_short  *q;
   int *p;

   wid = wdCreate();

   k->lamc = K2917_ENABLE;
#ifdef MVME147
   q = (u_short *)0xfffe200e;    /* Interrupt Handler Mask */
   *q = (u_short)(0xff02);       /* Register of MVME147 */
   q = (u_short *)0xfffe101d;    /* Watchdog Timer Control */
   *q = (u_short)(0x00ff);       /* Register of MVME147 */
#endif

#ifdef MVME167
   p = (int *)0xfff4006c;        /* Local Bus Interrupter */
   *p = (int)(0x40000001);       /* Enable register */
   p = (int *)0xfff40000;
   *p = (int)(0xffff0000);
```

```
    p = (int *)(0xfff40008);      /* DMA Map Decoder */
    *p = (int)(0x00000000);       /* Initialization  */
    p = (int *)(0xfff40010);
    *p = (int)(0x00000391);
#endif

    if (cc->cc_busy == CC_BUSY) {
       printf("Busy ...\n");
       return(16);   /* EBUSY */
    }
    cc->cc_busy = CC_BUSY;
    pVec.sv_handler = sigHandler;
    pVec.sv_mask  = 0xffff;
    pVec.sv_flags = 0;
    sigvec(SIGUSR1, &pVec, &pOvec);
    task_id = taskIdSelf();
    k->lamv = INT_VEC_LAM;
    return 0;
}

/*-------------------------------------------------------------*/
/*                                                             */
/* ccClose(): CAMAC close procedure.                           */
/*                                                             */
/*-------------------------------------------------------------*/
LOCAL int ccClose()
{
    register struct cc_device *cc = &ccdevice[0];

    cc->cc_busy = 0;
    return 0;
}

/*-------------------------------------------------------------*/
/*                                                             */
/* ccRead(): CAMAC read procedure.                             */
/*                                                             */
/*-------------------------------------------------------------*/
LOCAL int ccRead(pccDevHdr, buf, len)
DEV_HDR *pccDevHdr;
char *buf;
int len;
{
    return (ccWrite(pccDevHdr, buf, len));
}

/*-------------------------------------------------------------*/
/*                                                             */
/* ccWrite(): CAMAC write procedure.                           */
/*                                                             */
/*-------------------------------------------------------------*/
LOCAL int ccWrite(pccDevHdr, buffer, nbytes)
DEV_HDR *pccDevHdr;
char *buffer;
int nbytes;
{
    struct message *pm;
    u_short mssg_command, mssg_mode, mssg_naf, mssg_dat, mssg_blocksize;
    register struct cc_device *cc = &ccdevice[0];
    register u_short mode, naf;

    pm = (struct message*)buffer;

    cc->status = 0;
```

```
    cc->mode = mode = pm->mode;
    cc->naf  = naf  = pm->naf;

    mssg_command = pm->command;
    mssg_mode = pm->mode;
    mssg_naf = pm->naf;
    mssg_dat = *(pm->ptr_data);
    mssg_blocksize = pm->block_size;

    switch (mssg_command) {
       case CC_CMD_DOSINGLE:
          switch (mssg_naf & 18) {
             case 0x0000:                    /* single CAMAC read */
                camac_s(mssg_mode, mssg_naf, pm->ptr_data);
                break;
             case 0x0010:                    /* single CAMAC write */
                camac_s(mssg_mode, mssg_naf, pm->ptr_data);
                break;
             default:
                camac_s(mssg_mode, mssg_naf, pm->ptr_data);
                break;
          }
          break;
       case CC_CMD_DOBLOCK:
          switch (mssg_naf & 18) {
             case 0x0000:                    /* block CAMAC read */
                camac_mb(mssg_mode, mssg_naf, pm->ptr_data, mssg_blocksize);
                /* camac_b(mssg_mode, mssg_naf, pm->ptr_data, mssg_blocksize); */
                break;
             case 0x0010:                    /* block CAMAC write */
                camac_mb(mssg_mode, mssg_naf, pm->ptr_data, mssg_blocksize);
                /*camac_mb(mssg_mode, mssg_naf, pm->ptr_data, mssg_blocksize);*/
                break;
             default:
                camac_b(mssg_mode, mssg_naf, pm->ptr_data, mssg_blocksize);
                break;
          }
          break;
       case CC_CMD_LOADLIST:

       case CC_CMD_LOADDOLIST:

       case CC_CMD_DOLIST:

       default:
                break;
    }
    return 0;
}

/*-------------------------------------------------------------*/
/*                                                             */
/* cctimeout(): CAMAC timeout procedure.                       */
/*                                                             */
/*-------------------------------------------------------------*/
LOCAL void cctimeout(param)
int param;
{
    register struct cc_device *cc = &ccdevice[0];

    cc->interrupt |= CC_K_INT_TIMEOUT;
    kill(task_id,SIGUSR1);
}
```

```
LOCAL VOID sigHandler(sig, code, sigContext)
int sig;
int code;
SIGCONTEXT *sigContext;
{

}

LOCAL VOID clkHandler()
{
   clk_count++;
}

/*-------------------------------------------------------------*/
/*                                                             */
/* ccIntr(): CAMAC interrupt handler.                          */
/*                                                             */
/*-------------------------------------------------------------*/
LOCAL int ccIntr(param)
int param;
{
   register struct cc_device *cc = &ccdevice[0];
   register struct CAMAC *CAMAC;

   CAMAC = (struct CAMAC *)k;
   if ((CAMAC->csr & LAM) != 0) {
      cc->interrupt |= CC_K_INT_LAM;
   }
   if ((CAMAC->csr & DONE) != 0) {
      cc->interrupt |= CC_K_INT_DONE;
   }
   if ((CAMAC->empc & INT_ENABLE) == 0) {
      cc->interrupt |= CC_K_INT_EMPTY;
   }
   if ((CAMAC->aboc & INT_ENABLE) == 0) {
      cc->interrupt |= CC_K_INT_ABORT;
   }
   /* ------ untimeout & wakeup */
   wdCancel(wid);
   kill(task_id,SIGUSR1);
   return 0;
}

/*-------------------------------------------------------------*/
/*                                                             */
/* ccIoctl(): CAMAC ioctl procedure.                           */
/*                                                             */
/*-------------------------------------------------------------*/
LOCAL int ccIoctl(pccDevHdr, function, arg)
DEV_HDR *pccDevHdr;
int function;
int arg;
{
   register struct cc_device *cc = &ccdevice[0];
   register struct CAMAC *CAMAC;
   int *data;
   int  cmd;
   int  sdat;
   int  idat, i;

   CAMAC = (struct CAMAC *)k;
   cmd = function;
   data = &arg;
```

```
switch (cmd) {
   case CCIOC_CHK_BRANCH:
            *(int *)data = cc->max_branch;
            break;
   case CCIOC_SET_BRANCH:
            if (*(int *)data >= 0 && *(int *)data < cc->max_branch)
               cc->cur_branch = *(int *)data;
            break;
   case CCIOC_SET_CRATE:
            if (*(int *)data >= 0 && *(int *)data < MAX_CRATE)
               cc->cur_crate = *(int *)data;
            break;
   case CCIOC_WAIT_LAM:
            /*task_id = taskIdSelf();*/
            /*CAMAC->lamc = K2917_ENABLE;*/
            /*for (i=0; i<100; i++) {*/
            if (cc->interrupt & CC_K_INT_LAM) {
               cc->interrupt &= ~CC_K_INT_LAM;
               CAMAC->lamc = K2917_ENABLE;
               return 0;
            }
            /*}*/
            /* ----- Timeout & Sleep ------*/
            wdStart(wid, arg, (VOIDFUNCPTR)cctimeout, 0);
            pause();
            cc->interrupt |= CC_K_INT_TIMEOUT;
            if (cc->interrupt & CC_K_INT_LAM) {
               cc->interrupt &= ~CC_K_INT_LAM;
            }
            else if (cc->interrupt & CC_K_INT_TIMEOUT) {
               cc->interrupt &= ~CC_K_INT_TIMEOUT;
               return CC_STA_TIMEOUT;
            }
            return 0;
            break;
   case CCIOC_ENB_LAM:
            camac_s((u_short)BIT24, (u_short)NAF(30,13,17), (int *)data);
            camac_s((u_short)BIT16, (u_short)NAF(30,0,1), &sdat);
            sdat |= 0x100;
            camac_s((u_short)BIT16, (u_short)NAF(30, 0, 17), &sdat);
            CAMAC->lamc = K2917_ENABLE;
            break;
   case CCIOC_DSB_LAM:
            CAMAC->lamc = K2917_DISABLE;
            idat = 0;
            camac_s((u_short)BIT24, (u_short)NAF(30,13,17), &idat); /*write mask */
            camac_s((u_short)BIT16, (u_short)NAF(30,0,1), &sdat);
            sdat &= ~0x100;
            camac_s((u_short)BIT16, NAF(30,0,17), &sdat);
            break;
   case CCIOC_GENZ:

   case CCIOC_GENC:

   case CCIOC_SETI:

   case CCIOC_REMI:

   case CCIOC_WAIT_SIG:

   case CCIOC_SET_DEBUG:

   case CCIOC_READ_STATUS:
```

```
        default:
                break;
    }
    return 0;

}

/*-------------------------------------------------------------*/
/*                                                             */
/*              OTHER  PROCEDURES                              */
/*                                                             */
/*-------------------------------------------------------------*/
uint32 kreg_addr(k2917_addr)
faddr_t k2917_addr;
{
    faddr_t local_addr;
    STATUS s;

    s = sysBusToLocalAdrs(VME_AM_USR_SHORT_IO, k2917_addr, &local_addr);
    if (s == OK) printf("s = %d, localAddr = 0x%x ", s, local_addr);
    else printf("sysBusToLocalAdrs failed...\n");
    return ((uint32) local_addr);
}

/*-------------------------------------------------------------*/
/*                                                             */
/*            slave single transfer mode                       */
/*                                                             */
/*-------------------------------------------------------------*/
camac_s(mode, naf, dat)
u_short mode, naf, *dat;
{
    register int counter;
    int cur_crate = 0;
    int status, temp;

    k->cma = CMA_INIT;                      /* Initialize memory pointer */
    k->cmr = mode | (cur_crate << 8);
    k->cmr = naf;
    k->cmr = HALT;
    k->cma = CMA_INIT;                      /* Reset memory pointer */

    counter = 0;
    switch (naf & 0x0018) {
      case 0x0000:                          /* CAMAC read */
        k->csr &= ~WRITEC;
        k->csr |= GO;                       /* Go! */
        while ((k->csr & (RDY | ERR)) == 0 && counter < TIMEOUT_COUNT_S)
            counter++;
        if ((k->csr & RDY) != 0) {
            if ((mode & BIT16) == 0) {
                *dat = k->dhr & 0x00FF;
                *(dat+1) = k->dlr;
            }
            else {
                *(dat+1) = k->dlr;
                *dat = (u_short)0;
            }
        }
        break;

      case 0x0010:                          /* CAMAC write */
        k->csr |= WRITEC;
        k->csr |= GO;                        /* Go! */
```

```
        while ((k->csr & (RDY | ERR)) == 0 && counter < TIMEOUT_COUNT_S)
            counter++;
        if ((k->csr & RDY) != 0) {
            if ((mode & BIT16) == 0) {
                k->dhr = *dat;
                k->dlr = *(dat+1);
            }
            else {
                k->dlr = *(dat+1);
             }
        }
        break;

      default:                                  /* NDT */
        k->csr |= GO;                           /* Go! */
        break;
    }

    while ((k->csr & (DONE | ERR)) == 0 && counter < TIMEOUT_COUNT_S)
        counter++;

    camac_qx = k->csr;

    if (counter >= TIMEOUT_COUNT_S) {
        status = CC_CAMAC_TIMEOUT;
        k->csr = RST;                          /***** RESET *****/
    }

    return ((status | camac_qx) >> 12);
}


/*-------------------------------------------------------------*/
/*                                                             */
/*            slave block transfer mode                        */
/*                                                             */
/*-------------------------------------------------------------*/
camac_b(mode, naf, dat, len)
u_short mode, naf, *dat, len;
{
    register int counter;
    int cur_crate = 0;
    int status, i;
    int temp = 0;

    k->cma = CMA_INIT;                          /* Initialize memory pointer */
    k->cmr = mode | (cur_crate << 8);
    k->cmr = naf;
    k->cmr = -(len & 0xFFFF);
    k->cmr = 0xFFFF;
    k->cmr = HALT;
    k->cma = CMA_INIT;                          /* Reset memory pointer */

    counter = 0;
    switch (naf & 0x0018) {
      case 0x0000:                              /* CAMAC read */
        k->csr &= ~WRITEC;
        k->csr |= GO;                           /* Go! */
        while ((k->csr & (ERR | DONE)) == 0) {
           temp++;
           while ((k->csr & (RDY | DONE | ERR)) == 0 && counter < TIMEOUT_COUNT_S)
               counter++;
           if ((k->csr & RDY) != 0) {
               if ((mode & BIT16) == 0) {
```

```
                *dat = k->dhr & 0x00FF;
                *(dat+1) = k->dlr;
                dat += 2;
            }
            else {
                *dat = k->dlr;
                dat++;
            }
        }
    }
    break;

  case 0x0010:                          /* CAMAC write */
    /* Not provided yet */
    break;

  default:                              /* NDT */
    k->csr |= GO;                       /* Go! */
    break;
}
while ((k->csr & (DONE | ERR)) == 0 && counter < TIMEOUT_COUNT_S)
    counter++;

camac_qx = k->csr;

if (counter >= TIMEOUT_COUNT_S) {
    status = CC_CAMAC_TIMEOUT;
    k->csr = RST;                      /***** RESET *****/
}

return ((status | camac_qx) >> 12);
}


/*-------------------------------------------------------------*/
/*                                                             */
/*            master block transfer mode                       */
/*                                                             */
/*-------------------------------------------------------------*/
camac_mb(mode, naf, dat, len)
u_short mode, naf, *dat, len;
{
register struct cc_device *cc = &ccdevice[0];
register int wc = (mode & BIT16) ? len : len*2;
register u_long dma_addr;
register int counter;
int cur_crate = 0;
int i,j;
int temp;

dma_addr = (u_long)dat;

k->cma = CMA_INIT;                      /* Initialize memory pointer */
k->cmr = mode | (cur_crate << 8);
k->cmr = naf;
k->cmr = -(len & 0xFFFF);               /* Max length = 1MWord */
k->cmr = 0xFFFF;
k->cmr = HALT;
k->cma = CMA_INIT;                      /* Reset memory pointer */
k->maclo = dma_addr & 0xFFFF;           /* Set DMA base address */
k->machi = dma_addr >> 16;
k->amr = AMR_INIT;                      /* Set VME AM code */
k->mtc = wc;
k->cser = DMA_RESET;                    /* DMA reset */
```

```
    switch (naf & 0x0018) {
       case 0x0000:                          /* CAMAC read */
          k->docr = DOCR_INIT | DMA_READ;
          k->sccr = DMA_START;
          k->csr |= DMA;                     /* DMA mode */
          k->csr &= ~WRITEC;
          break;

       case 0x0010:
          k->docr = DOCR_INIT | DMA_WRITE;
          k->sccr = DMA_START;
          k->csr |= DMA;                     /* DMA mode */
          k->csr |= WRITEC;
          break;

       default:
          return;                            /* ERROR */
          break;
    }

    k->csr |= GO;

    counter = 0;
    while ((k->csr & (ERR | DONE)) == 0 && counter < TIMEOUT_COUNT_B) {
        for (i = 0; i < 1000; i++)           /* wait */
            j += i;
        counter++;
    }
    if (counter >= TIMEOUT_COUNT_B) cc->status = CC_CAMAC_TIMEOUT;

    cc->camac_qx = k->csr;

    if ((k->csr & ERR) == 0) {
       cc->retlen = (cc->mode & BIT16) ?
          len - k->mtc : cc->len - k->mtc / 2;
       cc->ptr_udata += wc;
       cc->len_udata += wc;
    }

    k->csr = RST;

    return;
}
```

```
/*-------------------------------------------------------------*/
/*    camlib.c     May 20, 1992          Cengiz Erbas  (SSCL)   */
/*                                                             */
/*      for MVME147 K2917 K3922 or MVME167 K2917 K3922         */
/*                                                             */
/* original by Y.Takeuchi (TIT) for SUN-SPARC SF-VME K2917 K3922 */
/*-------------------------------------------------------------*/

#include <stdio.h>
#include <errno.h>
#include <sys/fcntl.h>
#include <sys/types.h>
#include <sys/file.h>
#include <sys/mman.h>
#include "vxWorks.h"
#include "vme.h"

#ifdef MVME167
#include "mv167.h"
#endif

#ifdef MVME147
#include "mv147.h"
#endif

#include "camlib.h"

#define W   unsigned short
#define B   unsigned char
#define CC_CAMAC_TIMEOUT 0x1000
#define KBASE     0x0000FF00;

typedef unsigned char *faddr_t;
typedef unsigned short uint16;
typedef unsigned long uint32;

static struct message message;

extern int camac_s();
extern u_short camac_qx;
int cc_path;

CAMOPN()
{
   int flags = 0;
   int mode  = 0;
   cc_path = open("/cc0", flags, mode);
   printf("cc_path = %d", cc_path);
   return 0;
}

CAMCLS()
{
   return close(cc_path);
}

CSETCR(crate)
int crate;
{
   return ioctl(cc_path, CCIOC_SET_CRATE, crate);
}

CSETBR(branch)
int branch;
```

```
{
    return ioctl(cc_path, CCIOC_SET_BRANCH, branch);
}

CGENZ()
{
    int status, i, q, x;

    status = CAMACW(NAF(30, 0, 1), &i, &q, &x);
    if (status != 0) return status;
    i |= 1;
    status = CAMACW(NAF(30, 0, 17), &i, &q, &x);
    return(status);
}

CGENC()
{
    int status, i, q, x;

    status = CAMACW(NAF(30, 0, 1), &i, &q, &x);
    if (status != 0) return status;
    i |= 2;
    status = CAMACW(NAF(30, 0, 17), &i, &q, &x);
    return(status);
}

CSETI()
{
    int status, i, q, x;

    status = CAMACW(NAF(30, 0, 1), &i, &q, &x);
    if (status != 0) return status;
    i |= 4;
    status = CAMACW(NAF(30, 0, 17), &i, &q, &x);
    return(status);
}

CREMI()
{
    int status, i, q, x;

    status = CAMACW(NAF(30, 0, 1), &i, &q, &x);
    if (status != 0) return status;
    i &= ~4;
    status = CAMACW(NAF(30, 0, 17), &i, &q, &x);
    return(status);
}

CAMACW(naf, dat, q, x)
int naf, *dat, *q, *x;
{
    char *buffer;
    register int status;
    u_short qx;
    int len;

    *(u_short *)dat = 0;
    message.command = (u_short)CC_CMD_DOSINGLE;
    message.mode = (u_short)BIT16;
    message.naf = (u_short)naf;
    message.ptr_data = (u_short *)dat;
    message.ptr_qx = (u_short *)&qx;
    buffer = (char *)&message;
    len = sizeof(struct message);
```

```
    write(cc_path, buffer, len);

    *x = ((camac_qx & NOX) == 0) ? 1 : 0;
    *q = ((camac_qx & NOQ) == 0) ? 1 : 0;

    return(camac_qx >> 12);
}


CAMAC(naf, dat, q, x)
int naf, *dat, *q, *x;
{
    char *buffer;
    register int status;
    u_short qx;
    int len;

    *(u_short *)dat = 0;
    camac_s((u_short)BIT24, (u_short)naf, (u_short *)dat);
    message.command = (u_short)CC_CMD_DOSINGLE;
    message.mode = (u_short)BIT24;
    message.naf = (u_short)naf;
    message.ptr_data = (u_short *)dat;
    message.ptr_qx = (u_short *)&qx;
    buffer = (char *)&message;
    len = sizeof(struct message);
    write(cc_path, buffer, len);

    *x = ((camac_qx & NOX) == 0) ? 1 : 0;
    *q = ((camac_qx & NOQ) == 0) ? 1 : 0;

    return(camac_qx >> 12);

}

CELAM(mask)
int mask;
{
    return ioctl(cc_path, CCIOC_ENB_LAM, mask);
}

CDLAM()
{
    int dummy;

    return ioctl(cc_path, CCIOC_DSB_LAM, dummy);
}

CWLAM(timeout)
int timeout;
{
    return ioctl(cc_path, CCIOC_WAIT_LAM, timeout);
}


CAMACB(naf, dat, q, x, len)
int naf, /**dat,*/ *q, *x, len;
u_short *dat;
{
    char *buffer;
    register int status;
    u_short qx;
    int length;
```

```
    message.command = (u_short)CC_CMD_DOBLOCK;
    message.mode = (u_short) (QSCAN|BLOCK|BIT16);
    message.naf = (u_short)naf;
    message.ptr_data = (u_short *)dat;
    message.ptr_qx = (u_short *)&qx;
    message.block_size = (u_short)len;
    buffer = (char *)&message;
    length = sizeof(struct message);
    write(cc_path, buffer, length);

    *x = ((camac_qx & NOX) == 0) ? 1 : 0;
    *q = ((camac_qx & NOQ) == 0) ? 1 : 0;

    return(camac_qx >> 12);

}
```

```
/*************************************************************
 * cc.h         20-FEB-1992            Y.Takeuchi (T.I.T.)   *
 *                                                           *
 * original: cc.h on SUN4 by Y.Yasu (KEK)                    *
 *                                                           *
 * for SUN-SPARC SF-VME K2917 K3922                          *
 *************************************************************/

/**************** User Definitions *****************/

#define DEBUG
#define DEBUG_LEVEL 0
/*
#define DEBUG_LIST
#define LIST_ERR_ABORT
*/
#define LIST_EXTENSION

#define CC_SIGNAL               SIGUSR1
#define CC_LIST_MAX_STEP        1000000
#define CC_LIST_MAX_LENGTH      0x20000 /* max len of list  64kw */
#define CC_LIST_MAX_DATA        0x20000 /* max len of data  64kw */
#define CC_KLIST_MAX_DIR        256     /* max number of K2917 lists */
#define CC_DMA_WAIT_LOOP        1000    /* DMA done poling rate for IPC */
/************* End of User Definitions *************/


#include <sys/types.h>
#include <sys/ioccom.h>

#define NCC     1
#define CC_MINPHYS_SIZE 0x10000         /* max word count = 64k */
#define CC_K_MAX_BRANCH 8
#define MAX_CRATE       8

/* command for message structure */
#define CC_CMD_DOSINGLE         1
#define CC_CMD_DOBLOCK          2
#define CC_CMD_LOADLIST         3
#define CC_CMD_LOADDOLIST       4
#define CC_CMD_DOLIST           5
#define CC_CMD_ADDKLIST         6
#define CC_CMD_DELKLIST         7
#define CC_CMD_EXEKLIST         8
#define CC_CMD_SAVEKLIST        9
#define CC_CMD_LOADKLIST       10

/* ioctl command */
#define CCIOC_CHK_BRANCH        _IOR(c,0,int) /* check branch # */
#define CCIOC_SET_BRANCH        _IOW(c,1,int) /* set branch # */
#define CCIOC_SET_CRATE         _IOW(c,2,int) /* set crate # */
#define CCIOC_WAIT_LAM          _IOW(c,3,int) /* wait LAM interrupt */
#define CCIOC_ENB_LAM           _IOW(c,4,int) /* enable LAM interrupt */
#define CCIOC_DSB_LAM           _IOW(c,5,int) /* disable LAM interrupt */
#define CCIOC_GENZ              _IOW(c,6,int)
#define CCIOC_GENC              _IOW(c,7,int)
#define CCIOC_SETI              _IOW(c,8,int)
#define CCIOC_REMI              _IOW(c,9,int)
#define CCIOC_WAIT_SIG          _IOW(c,10,int)
#define CCIOC_SET_DEBUG         _IOW(c,11,int)
#define CCIOC_READ_STATUS       _IOR(c,12,int)
#define CCIOC_KINIT             _IOW(c,13,int)

#define CC_K_INT_LAM            1
```

```
#define CC_K_INT_DONE           2
#define CC_K_INT_EMPTY          4
#define CC_K_INT_ABORT          8
#define CC_K_INT_TIMEOUT        0x8000

#define CC_K_TIME_TIMEOUT       100     /* 100 * 10 msec */
#define CC_DMA_TIMEOUT          200     /* 200 * 10 msec */
#define CC_CAMAC_TIMEOUT        0x1000  /* for cc->status cc->qx */

#define CC_WRITE_FAIL_COUNT     10      /* for Interrupt registers */
#define CC_WRITE_FAIL           0x0800  /* for cc->status */

#define CC_LIST_STOP            0x0010  /* not equal 0 */
#define CC_LIST_EMPTY           2
#define CC_LIST_TIMEOUT         0x8000

#define CC_KLIST_YES            1
#define CC_KLIST_NO             0
#define CC_KLIST_CMAINIT        100
#define CC_KLIST_SAMENAME       1
#define CC_KLIST_MEMORYFULL     2
#define CC_KLIST_NONAME         3

#define CC_STA_TIMEOUT          2

#define CC_BUSY                 1       /* for cc->busy */


struct iosb {
    int status;
    int ret_length;
    int s_reg;
};

struct message {
    u_short command;
    u_short mode;
    u_short naf;
    u_short *ptr_data;
    u_short *ptr_qx;
    u_short block_size;     /* added by C. Erbas */
    struct iosb *ptr_iosb;
    char *klname;
};

/* #ifdef KERNEL */

struct cc_device {
    int status;         /* current system status */
    int cc_busy;        /* true if the device is busy */
    struct mb_device *md; /* current pointer of md */
    struct CAMAC *CAMAC; /* current pointer of CAMAC */
    u_short *ptr_list;  /* current pointer to list_buffer */
    int len_list;       /* actual length processed */
    u_short *ptr_udata_s; /* first pointer to user's buffer address */
    u_short *ptr_udata; /* current pointer to user's buffer address */
    int len_udata_t;    /* total length of user's buffer */
    int len_udata;      /* actual length of user's buffer */
    u_short *ptr_kdata; /* current pointer to kernel's buffer address */
    int len_kdata;      /* actual length of kernel's buffer processed */
    int s_reg;
    int t_reg;
    int a_reg;
    int event_count;
```

```
        u_short camac_qx;
        int branch_flag;
        int max_branch;
        int cur_branch;
        int cur_crate;
        int interrupt;
        u_short mode;
        u_short naf;
        int len;
        int retlen;
        struct uio *uio;
        dev_t dev;
        int klist;
};

/* #endif */
```

```
/*******************************************************
 * k2917.h   Kinetic K2917 registers                   *
 *                                                     *
 *   20-FEB-1992 Y.Takeuchi                            *
 *******************************************************/

#define KREG_BASE       0xFF00
#define VME_INT_LEVEL   1                   /* CAMAC mo kaeru */
#define VME_INT_VECTOR  0xff


#define NAF(n, a, f)    (((n) << 9) + ((a) << 5) + (f))

/***** DMA *****/
#define DMA_DONE        0x8000
#define DMA_ERR         0x1000
#define DMA_ACT         0x0800
#define DOCR_INIT       0x8010
#define DMA_READ        0x0080
#define DMA_WRITE       0x0000
#define DMA_START       0x0080
#define DMA_ABORT       0x0010
#define DMA_INT_ENABLE  0x0008
#define DMA_RESET       0x0000

/***** Interrupt *****/
#define INTR_INIT       0x0000
#define TEST_FLAG       0x0080
#define FLAG_AUTO_CLEAR 0x0040
#define INT_ENABLE      0x0010
#define INT_AUTO_CLEAR  0x0008
#define IRQ1            0x0001
#define IRQ2            0x0002
#define IRQ3            0x0003
#define IRQ4            0x0004
#define IRQ5            0x0005
#define IRQ6            0x0006
#define IRQ7            0x0007

/***** Main register *****/
#define AMR_INIT        0x007D
#define CMA_INIT        0x0000

#define AD              0x0001
#define INLINE          0x0060
#define BLOCK           0x0020
#define BIT24           0x0000
#define BIT16           0x0002
#define QSTOP           0x0000
#define QIGNO           0x0008
#define QREPE           0x0010
#define QSCAN           0x0018
#define HALT            0x0080

#define ERR             0x8000
#define ABT             0x4000
#define TMO             0x2000
#define RST             0x1000
#define LAM             0x0200
#define RDY             0x0100
#define DONE            0x0080
#define DMA             0x0040
#define WRITEC          0x0020
#define NOX             0x0004
```

```
#define NOQ             0x0002
#define GO              0x0001

/* for cc device driver */
#define K2917_ENABLE    (0xf8 | VME_INT_LEVEL)
#define K2917_DISABLE   (0xe8 | VME_INT_LEVEL)
#define K2917_VECTOR    VME_INT_VECTOR

#define W   unsigned short

#define K_REG CAMAC                     /* for cc.c */

struct K_REG {
    W cser;
    W dummy02;
    W docr;
    W sccr;
    W dummy05;
    W mtc;
    W machi;
    W maclo;

    W dummy11;          /* 0x10 - 0x1F */
    W dummy12;
    W dummy13;
    W dummy14;
    W dummy15;
    W dummy16;
    W dummy17;
    W dummy18;

    W dummy21;          /* 0x20 - 0x2F */
    W dummy22;
    W dummy23;
    W dummy24;
    W dummy25;
    W dummy26;
    W dummy27;
    W dummy28;

    W dummy31;          /* 0x30 - 0x3F */
    W dummy32;
    W dummy33;
    W dummy34;
    W dummy35;
    W dummy36;
    W dummy37;
    W dummy38;

    W lamc;
    W donc;
    W empc;
    W aboc;
    W lamv;
    W donv;
    W empv;
    W abov;

    W dummy51;          /* 0x50 - 0x5F */
    W dummy52;
    W dummy53;
    W dummy54;
    W dummy55;
    W dummy56;
```

```
        W dummy57;
        W dummy58;

        W amr;
        W cmr;
        W cma;
        W cwc;
        W srr;
        W dlr;
        W dhr;
        W csr;
};

struct klist {
    char name[9];
    W cma;
    W wc;
};
```